Compas'2023 : Parallélisme/ Architecture/ Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

Taenite : Gestion transparente de la mémoire persistante en Rust

Louis Boulanger, Frédéric Wagner, Yves Denneulin

Université Grenoble Alpes, Inria, CNRS, LIG
Laboratoire LIG - Bâtiment IMAG - 700 Avenue Centrale
38400 Saint-Martin-d'Hères - France
louis.boulanger@univ-grenoble-alpes.fr

Résumé
Nous présentons *Taenite*, une bibliothèque transactionnelle de programmation pour la mémoire persistante. Basée sur le principe de *copy-on-write*, elle permet d'outre-passer le besoin de journaux et confirme, à la fin d'une transaction, toutes les modifications atomiquement. La cohérence des données est garantie à tout moment et les reprises après pannes sont instantanées. Son interface de programmation, proche de celle de la bibliothèque standard de Rust, permet une utilisation simple et transparente. Nous détaillons son principe de fonctionnement et, plus particulièrement, son allocateur mémoire. À partir de briques élémentaires nous mettons en place la construction de structures persistantes complexes telles des structures arborescentes comme les arbres auto-stoppeurs ou des DAG.

Mots-clés : Rust, Mémoire persistante, NVRAM

---

1. Introduction

La hiérarchie des mémoires d'ordinateurs est en constante évolution depuis le début de l'informatique : des cartes perforées aux *SSDs*, l'état de l'art concernant le stockage de données n'a cessé d'évoluer. Deux grandes catégories de mémoires existent : les mémoires persistantes (disques durs, mémoire flash, ...) qui conservent les données, et volatiles (caches CPU, RAM, ...), qui sont addressables à l'octet. Ces dernières années ont vu l'apparition et le développement de mémoires réunissant les caractéristiques de ces deux catégories : adressage à l'octet en non au secteur, et persistence ; ce sont les mémoires non-volatiles, NVRAM [7]. Développer des applications ou des systèmes utilisant ces mémoires est rendu compliqué par l'obligation de la prise en compte de la hiérarchie mémoire : pour l'application, chaque écriture est vue comme définitive alors qu'elle n'est physiquement réalisée que dans la mémoire cache du processeur qui l'a effectuée. Pour garantir la persistence des écritures, le plus simple est de réaliser une opération de synchronisation des caches après chaque écriture, ce qui est très coûteux et donc irréalisable en pratique. Une approche permettant de conserver de meilleures performances consiste à regrouper ces opérations d'écriture en des entités dont la réalisation se déroule de manière atomique, soit toutes les modifications sont reportées en mémoire persistente soit aucune.

2. État de l'art

Le concept de bibliothèque de gestion de NVRAM n'est pas nouveau. Intel en propose une pour ses propres composants[7, 11], et de nombreuses équipes en ont proposées. Les bibliothèques transactionnelles sont fréquentes : celle d'Intel[11] en est une. Plusieurs types de journaux ont été explorés, comme l'*undo log* avec Romulus[3] et le *redo log* avec Romulus et Pisces[5]. Les journaux ont souvent été identifiés comme coûteux[9], et des bibliothèques comme Clobber-NVM[14] s'en défont au profit de passes compilateur. ArchTM[13] a la particularité d'utiliser le principe de *Copy-on-Write* pour éviter les journaux mais utilisent du *garbage-collection* pour libérer leurs ressources. Corundum[6] est développée en Rust, et propose une méthode pour filtrer les types autorisés en NVRAM ou non, mais se base encore sur des journaux.
D'autres bibliothèques choisissent de ne pas utiliser de transaction et se basent alors sur des sauvegardes à intervalles réguliers : ResPCT[8] utilise une méthode de gestion du cache CPU[1], et Montage[12] limite la fréquence des checkpoints pour augmenter les performances des opérations.
Le langage fortement typé Rust[10] a pour objectif de fournir un modèle sûr d'accès à la mémoire même dans le cadre de la programmation concurrente tout en fournissant un niveau de performances du code généré proche des meilleurs langages. Une de ses caractéristiques fondamentales est le principe de propriété des objets : chaque variable ne peut être accédé que par un flux d'exécution à la fois. Le partage éventuel ne peut se faire que sur des références aux variables, ces références pouvant permettre de lire ou d'écrire l'objet suivant le principe du lecteurs multiples/ rédacteur unique. Ces vérifications sont effectuées statiquement par le compilateur.
Nous pensons que la finesse de l'accès aux données que fournit le système de typage de Rust permet une maitrise des accès mémoire qui rend superflue la notion de transactions. Dans cet article nous présentons donc notre proposition d'une bibliothèque de gestion de données pour la NVRAM, *Taenite*, qui se base sur le concept du *copy-on-write* couplé à une gestion fine des accès aux données afin que l'ensemble des valeurs de variable d'un programme soit toujours dans un état cohérent. De plus, en ne recopiant que ce qui est strictement nécessaire, *Taenite* fonctionne avec un surcoût limité en ce qui concerne les opérations de cohérence des données persistantes.

3. *Taenite*, une bibliothèque sans journal

Dans cette section, nous présentons les principes fondamentaux de *Taenite*. Son objectif principal est de proposer des accès sécurisés à la NVRAM à l'aide de transactions sans journaux.

3.1. Principe de fonctionnement

Le principe d'une transaction est de modifier des données en réalisant un ensemble d'opérations qui doivent toutes réussir afin d'obtenir un nouveau jeu de données valide. Prenons par exemple une structure de liste chaînée sur laquelle on souhaite ajouter une nouvelle cellule directement derrière la tête. Nous allons alors :

1. allouer une nouvelle cellule en mémoire persistante ;
2. y stocker la nouvelle valeur ;
3. mettre comme cellule suivante la seconde cellule de la liste ;
4. mettre notre nouvelle cellule comme nouvelle suivante de la tête de liste.

Pour obtenir une liste valide, l'ensemble des opérations doit donc réussir. En particulier, si le programme est interrompu juste après l'allocation la liste initiale est encore valide mais une fuite mémoire en mémoire persistante se produit.

Pour éviter l'utilisation de journaux, nous faisons en sorte que chaque opération ne réalise que des modifications temporaires qui seront toutes validées simultanément en sortie de transaction. En cas d'arrêt intempestif aucune modification n'ayant réellement eu lieu le système restera dans son état initial valide. C'est un avantage majeur sur les systèmes de transactions à base de journaux qui nécessitent de réaliser une correction de l'état de la mémoire lors du redémarrage[9].
Pour atteindre cet objectif, *Taenite* fonctionne en stockant deux jeux de données. Les vraies données de l'utilisateur sont stockées dans un objet racine contenant également l'allocateur mémoire. En entrée de transaction nous réalisons une copie de la racine et travaillons lors de la transaction sur la copie, laissant l'originale inchangée. Racine originale et copie sont stockées dans un tableau de 2 cases et nous disposons d'un booléen indiquant quel jeu de données est le jeu valide, sur un principe similaire à ArchTM[13]. En sortie de transaction un *flush* synchronise les données en NVRAM puis le booléen est inversé ce qui valide atomiquement toutes les modifications réalisées.
Pour de bonnes performances, il est nécessaire que les copies des données originales soient réalisées efficacement. Pour ce faire nous utilisons différentes techniques issues du paradigme de programmation fonctionnelle autour des *structures immuables*. Nous évitons un maximum de copies tout en réalisant celles qui sont nécessaires de manière paresseuse. Nous donnons différents exemples de ces structures section 4.

3.2. L'allocateur fondamental

Dans *Taenite* différents allocateurs sont disponibles mais l'un d'entre eux sert de base à tous les autres. L'allocateur fondamental permet d'allouer et libérer des blocs mémoire de taille fixe, sans journaux. Comme dans un allocateur classique nous stockons une liste chaînée des blocs libres. À la différence des listes classiques nous ne pourrons pas stocker ici les pointeurs à l'intérieur des blocs. En effet, il se peut qu'un bloc soit libre en début de transaction mais alloué dans la transaction. Tout bloc est donc potentiellement simultanément libre et utilisé. Nous acceptons donc de payer un surcout en mémoire en stockant les pointeurs juste après chaque bloc.
Néanmoins cela n'est pas suffisant. Un premier problème à gérer vient d'une libération d'un bloc ancien (alloué lors d'une transaction précédente). En effet, il n'est pas possible de ré-allouer ce bloc dans la transaction courante car son contenu pourrait alors être écrasé par l'utilisateur. Or ce bloc doit toujours être considéré comme alloué jusqu'à la bascule de fin de transaction. Nous n'avons donc pas le droit d'écrire à l'intérieur. Comme solution les blocs anciens libérés sont ré-insérés dans la liste mais en queue de liste, tandis que les blocs alloués sont pris en tête de liste. De cette manière l'écriture d'une donnée allouée ne peut avoir lieu avant l'épuisement complet de la mémoire. Les blocs alloués dans la transaction courante puis re-libérés sont eux réinsérés en tête de liste.
Un deuxième problème vient du fait qu'il est possible pour un bloc d'avoir deux successeurs. La figure 1 nous en donne un exemple. Nous considèrons ici trois blocs mémoire A, B, C. Initialement ils sont tous libres (liens en rouge, pointillés). Les opérations courantes sont ici représentées en vert (trait plein). Nous commençons par effectuer deux allocations, ce qui bouge la tête sur C. Puis, dans un second temps, on libère A. À ce stade le bloc A a donc deux blocs successeurs : B à la fin de la transaction précédente et C dans la transaction courante. À tout instant la liste rouge doit rester inchangée et il nous est donc nécessaire d'avoir deux pointeurs suivants pour chaque bloc.
Cet exemple conclut la présentation générale de l'allocateur. L'algorithme exact est en réalité un peu plus complexe mais nous ne rentrerons pas plus dans les détails ici.

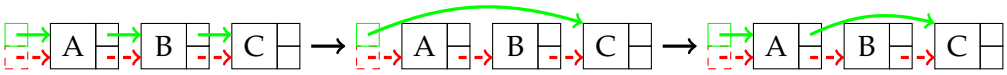


FIGURE 1 – Allocateur : malloc, malloc ; free(A)

4. Une API transparente

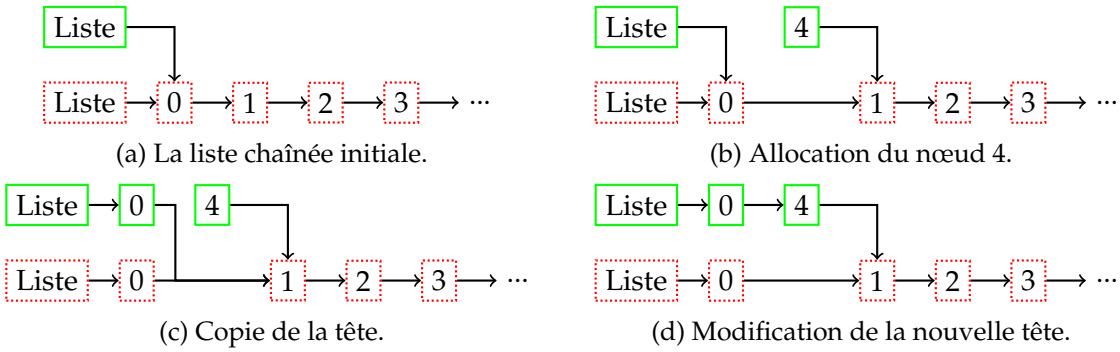
Cette section présente l'interface de programmation conçue pour *Taenite* et démontre les intérêts à ce qu'elle soit aussi proche que possible de celle de la bibliothèque standard de Rust.

4.1. Primitive d'allocation mémoire utilisateur avec la *PBox*

Afin que les modifications à effectuer pour adapter un code existant pour fonctionner en mémoire persistante avec *Taenite* soient moindres, il faut que nous donnions accès aux utilisateurs aux mêmes méthodes sur des types analogues à ceux de la bibliothèque standard.
La primitive d'allocation mémoire côté utilisateur de Rust est la "Boîte" (`Box<T>`). Générique sur un type `T`, la boîte alloue assez de mémoire pour stocker une seule instance de ce type sur le tas, utilisant un allocateur optionnellement défini par l'utilisateur. Nous proposons donc la "Boîte persistante" (`PBox<T>`), qui permet d'allouer une instance d'un type en NVRAM. Contrairement à celle de la bibliothèque standard, qui utilise comme allocateur par défaut `malloc`, notre allocateur par défaut est l'allocateur persistant décrit en section 3.2.
La mutabilité est un problème dans notre modèle actuel : si nous autorisons les accès mutables arbitraires, alors la mémoire qui se doit de rester immuable en cas de panne peut être modifiée par l'utilisateur. C'est pourquoi, tout comme décrit pour la racine et l'état de l'allocateur en section 3.1, toute structure allouant de la mémoire persistante (comme la "Boîte persistante" doit faire en sorte de copier ses données allouées lors d'une modification.
Nous implémentons cette fonctionnalité de copie grâce à un mécanisme de Rust appelé un *trait*. Proche des "interfaces" dans les langages orientés objets, certains traits permettent de modifier le comportement de certaines opérations basiques du langage. C'est le cas des traits `Deref` et `DereMut`, qui respectivement dictent le comportement du déréferencement immutable et du déréférencement mutable ; la méthode `deref_mut` du trait `DerefMut` est appelée à chaque déréferencement mutable. Implémenté sur notre type `PBox`, ce trait effectue une copie des données de la boîte et renvoie une référence vers ces nouvelles données, de façon totalement transparente à l'utilisateur.
La figure 3 illustre ce mécanisme en ré-utilisant l'exemple de la liste chaînée introduit en section 3.1 avec 4 phases. La figure 3a représente la liste chaînée originale. On remarque qu'il existe deux "listes" : étant dans une transaction, la liste actuelle, en vert, représente la copie de son état précédent, et pointe donc sur la même tête que l'état immuable du passé. Dans 3b, un nouveau nœud est alloué, et pointe lui-même sur le suivant de la tête ; cela correspond aux lignes 2 et 3 de la figure 2. Puis, à la ligne 4 la tête est modifiée, ce qui déclenche la copie des données de la tête comme illustré dans 3c. Cette copie de la tête est rattachée à la liste courante, étant donné que la liste du passé est immuable (en effet, si une panne se produit, nous devons pouvoir retrouver l'ancienne liste). Étant une copie conforme, cette nouvelle tête pointe toujours sur le même suivant que l'ancienne tête. La figure 3d montre la deuxième étape de la ligne 4 du code en figure 2, qui assigne le suivant de la nouvelle tête. Cette modification est autorisée, étant donné qu'elle s'effectue sur une copie de la tête originale.
Il est intéressant de noter que cet exemple fonctionnerait sans *Taenite* ; la seule chose qui change est le type de boîte : `Box` sans, et `PBox` avec. Avec un alias de type, il est possible de créer une

```rust
fn ajout_apres_tete(tete: &mut PBox<Noeud>) {
  let mut noeud = PBox::new(Noeud::nouveau(4));
  noeud.suivant = tete.suivant;
  tete.suivant = noeud;
};
```

FIGURE 2 – Code Rust avec *Taenite* qui insère un nœud après la tête d'une liste chaînée.



(a) La liste chaînée initiale.

(b) Allocation du nœud 4.

(c) Copie de la tête.

(d) Modification de la nouvelle tête.

FIGURE 3 – Illustration de l'ajout d'un nœud après la tête d'une liste chaînée persistante. L'apparence des boîtes reflète leur état : rouges et pointillées pour une allocation dans une transaction précédente, et vertes et pleines pour une allocation dans la transaction courante.

bibliothèque qui fonctionne dans les deux modes, comme illustré en section 4.2.
La *Boîte persistante* nous permet ainsi de modéliser de nombreux types de structures de données persistantes, à condition qu'elles soient arborescentes : c'est à dire qu'une valeur n'a qu'un et qu'un seul chemin de référence vers la racine ; notamment, les DAG ne sont pas possibles avec seulement la `PBox`.

4.2. Une collection de données fonctionnelle : *L'arbre autostoppeur*

En se basant sur la *Boîte persistante*, ainsi que qu'une autre structure de données analogue, le *Vecteur persistant*, nous avons pu implémenter une structure de données fonctionnelle très adaptée au contexte de la NVRAM : l'*Arbre autostoppeur* ("Hitchhiker tree")[4]. Cette structure arborescente est basée sur le B+Tree[2], où l'arbre *m-aire* ne stocke que des copies de clefs dans ses nœuds internes. L'amélioration principale de l'arbre autostoppeur est l'addition d'un tampon de couples clef-valeur de taille fixe, sur chaque nœud intermédiaire.
L'intérêt de ces tampons est de limiter la distance à parcourir dans l'arbre pour trouver la destination d'un couple clef-valeur. S'il reste de la place dans le tampon d'un nœud, alors le couple y est inséré, sans parcourir le reste de l'arbre ; s'il est plein, alors le tampon se vide dans les enfants du nœud, remplissant leurs tampons, jusqu'à arriver à la feuille. Les données transitent alors d'insertion en insertion, d'où le nom "autostoppeur".
Cette méthode a un avantage dans le contexte de mémoire persistante, qui est plus coûteuse que la RAM en terme d'accès mémoire. Elle unifie l'écriture de données en $\mathcal{O}(1)$ quand le tampon n'est pas plein, et la complexité intrinsèque des recherches dans un B+Tree.
Nous avons implémenté cette structure de données de deux façons : une implémentation en RAM, utilisant seulement la bibliothèque standard de Rust ; et avec *Taenite*, en utilisants les types présentés précédemment. Grâce à notre interface de programmation qui imite la bibliothèque standard, nous n'avons en fait qu'un seul "code" d'implémentation ; nous pouvons décider

d'utiliser nos types, ou bien ceux de la bibliothèque standard, en modifiant seulement une ligne d'import. La logique de la structure de donnée est écrite exactement de la même façon dans les deux implémentations.

4.3. Au-delà de l'arborescence avec la mutabilité intérieure persistante

Comme expliqué en section 4.1, avec seulement la *Boîte persistante*, notre interface de programmation est limitée dans le type de structure de données possible à exprimer. Nous sommes en effet contraints de respecter une certaine arborescence à partir des types allouants, comme la *Boîte persistante* ou le *Vecteur persistant* : la racine de notre système doit pouvoir atteindre toute autre objet.

Rust introduit un type de structure de données qui permet d'outre-passer la vérification de mutabilité à la compilation. Le compilateur permet de vérifier statiquement et exhaustivement si les règles de mutabilité de Rust sont respectées ; c'est à dire que chaque *valeur* est soit référencée immutablement par une ou plusieurs variables, ou bien référencée mutablement par une et une seule variable, et jamais les deux à la fois. Cette règle, appelée *aliasing XOR mutability*, permet d'éliminer une grande catégorie de problèmes souvent présents dans d'autres langages systèmes.

Cependant, certaines structures de données ne peuvent pas prouver qu'elles respectent cette règle à la compilation. Rust propose donc alors des types à la *mutabilité intérieure* : au lieu de vérifier le bon fonctionnement à la compilation, d'autres moyens sont utilisés. La *Cellule* (`Cell`) ne permet de stocker que des types qui sont *copiables* ; la *Cellule à référence* (`RefCell`) vérifie la règle à l'exécution et termine le programme si une violation est détectée.

Nous pouvons implémenter une *Cellule* persistante, en se basant sur les *arbres autostoppeurs* détaillés en section 4.2. L'arbre, faisant parti de la racine de la mémoire persistante, aurait comme clef un identifiant unique, et comme valeur une *Boîte persistante* vers des données. Chaque *Cellule persistante* n'est donc qu'une copie de cet identifiant. Lors d'accès aux données d'une cellule, nous allons donc rechercher dans cet arbre le couple clef-valeur correspondant à l'identifiant de cette cellule. Nous gardons ainsi une arborescence nécéssaire pour garantir la cohérence des donneés persistante, tout en autorisant la mutabilité intérieure.

Grâce à ces structures mutables intérieurement, il est possible d'implémenter des *pointeurs à références comptées* (`Rc`) persistants. Nous l'implémentons en stockant ses valeurs de liens forts et faibles dans une cellule à mutabilité intérieure.

5. Conclusion

Les journaux transactionnels étant coûteux, nous avons conçu notre bibliothèque de gestion de mémoire persistante, *Taenite*, et illustré son principe de fonctionnement sans journal et son interface de programmation. Elle s'appuie sur les fonctionnalités intrinsèques du langage Rust pour rendre son utilisation facile et transparente, à la fois pour NVRAM et la bibliothèque standard de Rust. Initialement limitée aux structures de nature arborescente, de par son fonctionnement où chaque donnée est identifiée par son lien à la racine, nous avons transcendé cette limitation en nous appuyant sur le concept de *mutabilité intérieure*.

Nous prévoyons de continuer notre travail sur *Taenite* sur plusieurs aspects, tels que le parallélisme, l'adaptation d'une plus grande partie de l'interface standard de Rust pour la mémoire persistante, ainsi que l'élaboration et l'évaluation d'une application pouvant bénéficier de nos techniques.

Bibliographie

1. Cohen (N.), Aksun (D. T.), Avni (H.) et Larus (J. R.). – Fine-grain checkpointing with in-cache-line logging. – In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, avril 2019.
2. Comer (D.). – Ubiquitous b-tree. *ACM Computing Surveys*, vol. 11, n2, juin 1979, pp. 121–137.
3. Correia (A.), Felber (P.) et Ramalhete (P.). – Romulus. – In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM, juillet 2018.
4. Greenberg (D.), Wellbach (C.), Bánffy (R.), Meza (R.) et Boykis (D.). – Hitchhiker tree technical documentation. – `https://web.archive.org/web/20230419152540/https://cljdoc.org/d/io.replikativ/hitchhiker-tree/0.1.7/doc/hitchhiker-tree`. Accédé : 2023-04-19.
5. Gu (J.), Yu (Q.), Wang (X.), Wang (Z.), Zang (B.), Guan (H.) et Chen (H.). – Pisces : A scalable and efficient persistent transactional memory. – In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, *USENIX ATC '19*, USENIX ATC '19, p. 913928, USA, 2019. USENIX Association.
6. Hoseinzadeh (M.). – Nvsl/corundum, février 2021.
7. Intel. – Intel optane persistent memory. – `http://web.archive.org/web/20220530054900/https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`. Accédé : 2022-05-30.
8. Khorguani (A.), Ropars (T.) et Palma (N. D.). – ResPCT. – In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, mars 2022.
9. Marathe (V.), Mishra (A.), Trivedi (A.), Huang (Y.), Zaghloul (F.), Kashyap (S.), Seltzer (M.), Harris (T.), Byan (S.), Bridge (B.) et Dice (D.). – Persistent memory transactions, 2018.
10. Matsakis (N. D.) et Klock II (F. S.). – The rust language. – In *ACM SIGAda Ada Letters*. ACM, janvier 2014.
11. pmem.io. – Persistent memory development kit. – `https://github.com/pmem/pmdk`, 2022.
12. Wen (H.), Cai (W.), Du (M.), Jenkins (L.), Valpey (B.) et Scott (M. L.). – A fast, general system for buffered persistent data structures. – In *50th International Conference on Parallel Processing*. ACM, août 2021.
13. Wu (K.), Ren (J.), Peng (I.) et Li (D.). – ArchTM : Architecture-Aware, high performance transaction for persistent memory. – In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 141–153. USENIX Association, février 2021.
14. Xu (Y.), Izraelevitz (J.) et Swanson (S.). – Clobber-NVM : log less, re-execute more. – In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, avril 2021.