Thrifty-malloc : un gestionnaire dynamique de mémoire pour systèmes embarqués multicoeurs avec mémoire transactionnelle matérielle

Thomas Carle * [1], Dimitra Papagiannopoulou[2], Tali Moreshet[3], Andrea Marongiu[4], Maurice Herlihy[2], and Iris Bahar[2]

[1]Université Paul Sabatier, Toulouse, France, {thomas.carle@irit.fr}
[2]Brown University, Providence, RI, USA, {prénom_nom@brown.edu}
[3]Boston University, Boston, USA, {talim@bu.edu}
[4]University of Bologna, Bologna, Italy, {a.marongiu@unibo.it}

Résumé
Cet article présente *thrifty-malloc* : un gestionnaire de mémoire dynamique compatible avec la mémoire transactionnelle matérielle, pour les systèmes embarqués multi-coeurs. Ce gestionnaire combine modularité, facilité d'utilisation et compatibilité avec la mémoire transactionnelle matérielle (HTM) dans un design léger et peu gourmand en mémoire. Thrifty-malloc est facile à déployer et à configurer pour des programmeurs non-experts. Il délivre de bonnes performances avec un faible surcoût en mémoire, pour des applications embarquées exhibant un niveau élevé de parallélisme exécutées sur des architectures many-coeurs. De plus, les mécanismes transparents qui permettent d'augmenter la résilience de ce gestionnaire aux situations dynamiques imprédictibles n'induisent qu'un faible surcoût temporel.

1. Introduction

Le nombre de coeurs de calcul disponibles dans les systèmes embarqués ne cesse d'augmenter, créant un besoin réel pour des interfaces de programmation (APIs) efficaces et sûres. Pour les systèmes embarqués à mémoire partagée, la *gestion dynamique de mémoire* est un problème crucial. Les systèmes embarqués étant souvent limités en mémoire, pour certaines applications il est impossible d'affecter la mémoire de façon statique et conservative. Au contraire, de tels systèmes nécessitent un gestionnaire dynamique de mémoire léger, et ne dépendant d'aucun système d'exploitation.

Historiquement, les *verrous* ont été utilisés pour synchroniser les applications multithreadées ou parallèles/distribuées. Cependant, les verrous séquentialisent l'exécution des sections critiques, limitant les bénéfices de la parallélisation. De plus, leur utilisation est dangereuse à cause des possibles interblocages, particulièrement désastreux pour les applications embarquées. La mémoire transactionnelle matérielle (HTM) [10] est une alternative de synchronisation plus simple à mettre en oeuvre pour les programmeurs. Cette méthode autorise toutes les sections critiques à s'exécuter en parallèle et fournit des mécanismes de secours en cas de détection de conflits sur les données. Malheureusement, les gestionnaires dynamiques de mémoire classiques [4] ne sont en général pas compatibles avec les mécanismes HTM.

Afin de régler ces problèmes, nous proposons une méthode générique rendant la gestion dynamique de mémoire compatible à la fois avec les systèmes embarqués et la HTM : le programmeur peut appeler *malloc()* et *free()* dans ou en dehors des transactions, et la méthode elle-même

∗. Ce travail a été en partie supporté par les bourses NSF CNS-1319095, CNS-1519576 et CNS-1301924.

est basée sur la synchronisation par HTM. Cette méthode permet à la fois le passage à l'échelle des performances et une utilisation simple. Pour de meilleures performances, cette méthode minimise le nombre d'opérations de synchronisation en provisionnant une petite zone de mémoire privée pour chaque thread. Dans le cas nominal, un thread alloue et retourne la mémoire directement dans sa zone privée sans synchronisation. Dans le cas particulier où un thread vide sa zone, il lance un mécanisme transparent (décrit dans cet article) qui permet de la recharger. Ce service est implémenté avec des transactions dédiées à ce seul objectif, et qui sont clairement séparées des transactions spécifiées par le programmeur dans son application.

Nous appelons notre gestionnaire de mémoire efficace *Thrifty malloc*. Il est conçu avec l'objectif de combiner performance et compatibilité avec l'utilisation de verrous ou de HTM, ce qui le rend particulièrement adapté aux systèmes embarqués multicoeurs.

Le reste de cet article est organisé de la façon suivante : la section 2 contient une présentation des travaux connexes, la section 3 présente notre méthode, ses algorithmes et l'architecture que nous avons visée. La section 4 présente les résultats obtenus par notre méthode sur des benchmarks représentatifs, et enfin la section 5 conclue.

2. Etat de l'art

2.1. Gestion dynamique de mémoire

Les systèmes de gestion dynamique de mémoire utilisent une structure de données partagée par tous les threads, qui représente les blocs de mémoire disponibles [20]. Chaque fois qu'un thread alloue ou retourne de la mémoire, il doit synchroniser ses accès à la structure partagée avec les autres threads concurrents. Ces (potentiellement) fréquentes synchronisations peuvent détériorer la performance, en particulier lorsque les threads utilisent beaucoup le gestionnaire de mémoire.

Une technique courante permettant de réduire ces pénalités de synchronisation consiste à diviser la mémoire en zones logiques séparées, ce qui permet de servir en parallèle les requêtes à des zones différentes [4, 8, 15]. Les systèmes embarqués étant généralement limités en mémoire, le nombre et la taille des zones mémoire est le fruit d'un compromis entre la quantité de mémoire disponible dans le système, le degré de parallélisme visé et la simplicité de design du gestionnaire de mémoire. Thrifty malloc a été conçu en prenant ces contraintes en compte.

2.2. Mémoire transactionnelle matérielle

Les architectures multicoeurs à mémoire partagée nécessitent des primitives pour synchroniser les accès concurrents aux structures de données partagées. C'est traditionnellement le rôle des verrous. Cependant, les verrous peuvent réduire la performance et consommer une énergie excessive, car ils utilisent des opérations de lecture-modification-écriture gourmandes en énergie et qui traversent toute la hierarchie mémoire. De plus, les verrous doivent être déployés de façon conservative à chaque endroit où un conflit est susceptible de se produire, même si celui-ci est peu probable. À l'opposé, les mécanismes de synchronisation *spéculative* détectent les conflits dynamiquement, rembobinent, puis retentent l'exécution uniquement dans le cas où des conflits se produisent effectivement.

La mémoire transactionnelle [10, 19] est une technique spéculative qui permet aux sections critiques de s'exécuter en parallèle. Si un conflit de données se produit, il est détecté et un ou plusieurs des threads en conflit sont rembobinés et relancés. L'implémentation de mémoire transactionnelle matérielle requiert trois composantes : le suivi transactionnel, le contrôle de version des données et la gestion/résolution des conflits. Le suivi transactionnel surveille les transactions effectuant des lectures et des écritures. Pour chaque ligne de données, plusieurs transactions peuvent lire cette ligne, ou alors une seule transaction peut y écrire. Lorsqu'une transaction essaye d'accéder (i.e. de lire ou écrire) à une ligne de données qui a été modifiée (i.e. écrite) par une autre transaction, un conflit de données apparaît. Le contrôle de version garde la

trace des versions spéculatives et non-spéculatives des données : si une transaction modifie une donnée, l'ancienne version non-spéculative doit être copiée quelque part afin d'être restaurée en cas de conflit. Finalement le schema de gestion et de récupération de conflit est responsable de l'initiation du processus de récupération, choisit les transactions à redémarrer, restaure leurs données originales et relance leur exécution depuis leur début. La limitation du taux de conflits est une question cruciale car des redémarrages répétés peuvent avoir un coût important en termes de performances temporelle et énergétique.

Dans les architectures conventionnelles, les verrous sont des adresses mémoire manipulées par des opérations particulières (e.g. test-and-set). Dans l'architecture utilisée pour nos travaux, les verrous sont situés dans une zone mémoire dédiée possédant une sémantique test-and-set, et séparée du reste de la mémoire. Dans tous les cas, il est dangereux ou potentiellement problématique d'utiliser les verrous à l'intérieur de transactions. Dans les architectures existantes, comme la famille de processeurs Intel Haswell, les sections critiques peuvent être remplacées automatiquement par des transactions grâce au processus de Hardware Lock Elision (HLE) [12], ce qui peut entrainer l'imbrication de transactions et ainsi réduire les performances en cas de conflits. Dans notre architecture, les mécanismes de redémarrage d'une transaction ne permettent pas de relacher un verrou acquis par la transaction, ce qui mènerait en pratique à des interblocages.

2.3. Allocation de mémoire dans des transactions

La combinaison des verrous et des transactions étant à éviter, les gestionnaires de mémoire classiques utilisant des verrous ne peuvent pas être appelés à l'intérieur de transactions. Cependant, des gestionnaires compatibles avec les transactions ont été développés.

Hudson et al. ont présenté *McRT-Malloc* [11]. Ce gestionnaire n'utilise pas de mémoire transactionnelle pour se synchroniser, mais s'exonère des verrous en utilisant des primitives de Compare-and-Swap (CaS) dans un algorithme non-bloquant, et donc compatible avec les mécanismes de mémoire transactionnelle. Cependant, cette compatibilité est limitée : le comportement correct d'applications faisant appel à McRT-Malloc n'est garanti qu'à la condition de prouver statiquement qu'aucun appel au gestionnaire depuis l'intérieur d'une transaction ne se produira au même moment qu'un appel concurrent réalisé en dehors d'une transaction, car le suivi transactionnel n'est pas activé en dehors des transactions. Cette condition est trop restrictive pour les applications que nous visons, et Thrifty-malloc a été conçu pour être totalement compatible avec les mécanismes de HTM.

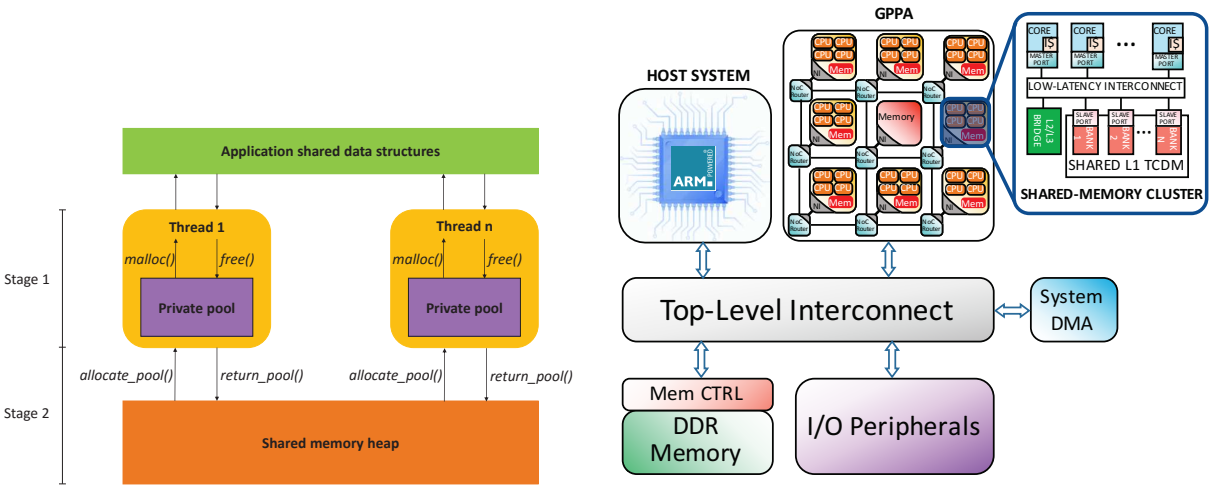
Mostly Lock Free Malloc ou LFMalloc [6] est un gestionnaire de mémoire similaire par bien des aspects avec McRT-Malloc : sa structure interne est conçue pour minimiser autant que possible les synchronisations. Dans sa version initiale, les synchronisations sont assurées par un verrou, et donc les limitations citées précédemment sont également présentes. Une version ultérieure [7] a été implémentée, dans laquelle ce verrou est remplacé par un mécanisme de HTM. Cependant, cela ne le rend pas pour autant totalement compatible avec l'utilisation de transactions au niveau de l'application, car là encore des transactions imbriquées peuvent apparaître. Encore une fois, Thrifty-malloc a été précisément conçu pour éviter ces écueils et offrir une compatibilité complète et efficace avec la HTM.

3. Implémentation de Thrifty-malloc

Dans cette section nous présentons les principes et algorithmes sous-jacents à notre méthode.

3.1. Principes généraux d'allocation/désallocation

Notre gestionnaire de mémoire implémente un schéma a deux niveaux, tel que décrit dans la figure 1(a). Chaque thread reçoit une zone privée de mémoire allouée depuis le tas, à partir de laquelle il peut directement allouer et retourner des blocs de mémoire durant l'exécution du programme. Ce recours aux zones de mémoire réduit drastiquement le nombre de synchroni-



(a) Schéma de notre gestionnaire de mémoire

(b) Cluster à mémoire partagée

FIGURE 1 – Vue haut-niveau de notre gestionnaire et de notre plateforme d'évaluation

sations réalisées par les différents threads, et est donc adapté aux applications parallèles [20]. Les zones allouées initialement aux threads ne couvrent pas la totalité du tas, dont la majeure partie reste disponible pour recharger les threads si leurs zones se vident durant l'exécution. Cette approche préserve la dynamicité des applications, en permettant à chaque thread de demander de la mémoire si, et quand il en a besoin, plutôt que de diviser statiquement le tas entre tous les threads.

3.2. Mécanismes transparents de rechargement des zones de mémoire

Pour des raisons de simplicité et de robustesse, nous souhaitons que notre gestionnaire de mémoire soit capable de réapprovisioner des zones de mémoire privées vides via des mécanismes transparents pour le programmeur et l'utilisateur. Un thread ayant vidé sa zone de mémoire peut demander qu'une nouvelle zone lui soit allouée depuis le tas, et ce peu importe si l'appel à *malloc()* qui a entrainé cette requête a été fait dans ou en dehors d'une transaction. Le traitement de cette demande requiert des synchronisations que nous assurons par le biais de transactions.

3.2.1. *Malloc()* appelé en dehors des transactions

Lorsqu'un appel à *malloc()* par un thread ne peut être servi car la zone de mémoire privée de ce thread n'a pas assez de mémoire disponible, une *routine de repli* est exécutée : ce qu'il reste de mémoire dans la zone allouée au thread est retourné dans le tas, une nouvelle zone mémoire est allouée au thread, puis *malloc()* est rappelé et tente d'allouer depuis cette nouvelle zone de mémoire. Ces deux opérations (retourner le reste de la zone de mémoire et allouer une nouvelle zone) sont effectuées à l'aide de deux transactions dédiées, ce qui permet de bien séparer ces étapes et de n'en recommencer qu'une en cas de conflit avec un autre thread. Afin de limiter le nombre d'appels à la routine de repli, la nouvelle zone mémoire allouée a une taille égale au maximum de la taille initiale de la zone actuelle et du double de la taille demandée par le *malloc()* en cours. Lorsque le *malloc()* tente d'allouer de gros blocs de mémoire, cela évite de vider directement la nouvelle zone de mémoire allouée, et d'avoir à rappeler la routine de repli à la prochaine occurence de *malloc()*.

3.2.2. *Malloc()* appelé dans une transaction

Si l'appel à *malloc()* ne pouvant être servi a été initié à l'intérieur d'une transaction, l'exécution de la routine de repli directement depuis la transaction en cours mènerait à des transactions imbriquées. Pour éviter cela, nous complexifions notre méthode : lorsque l'appel à *malloc()* ne

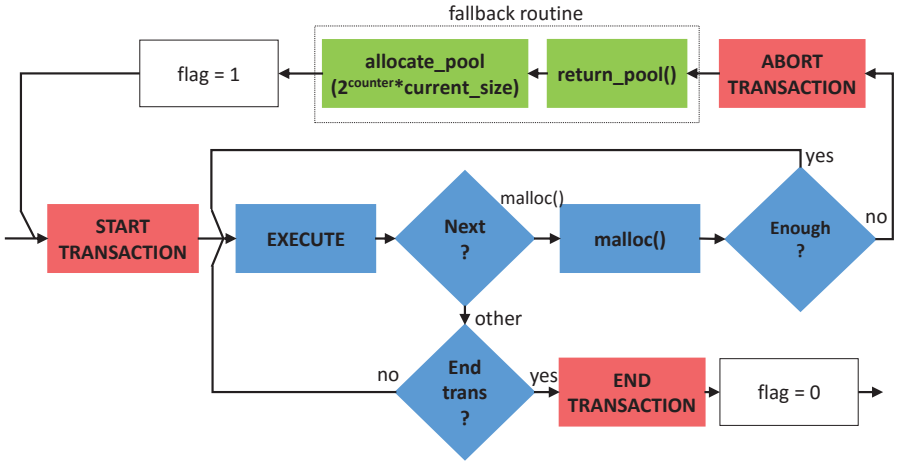


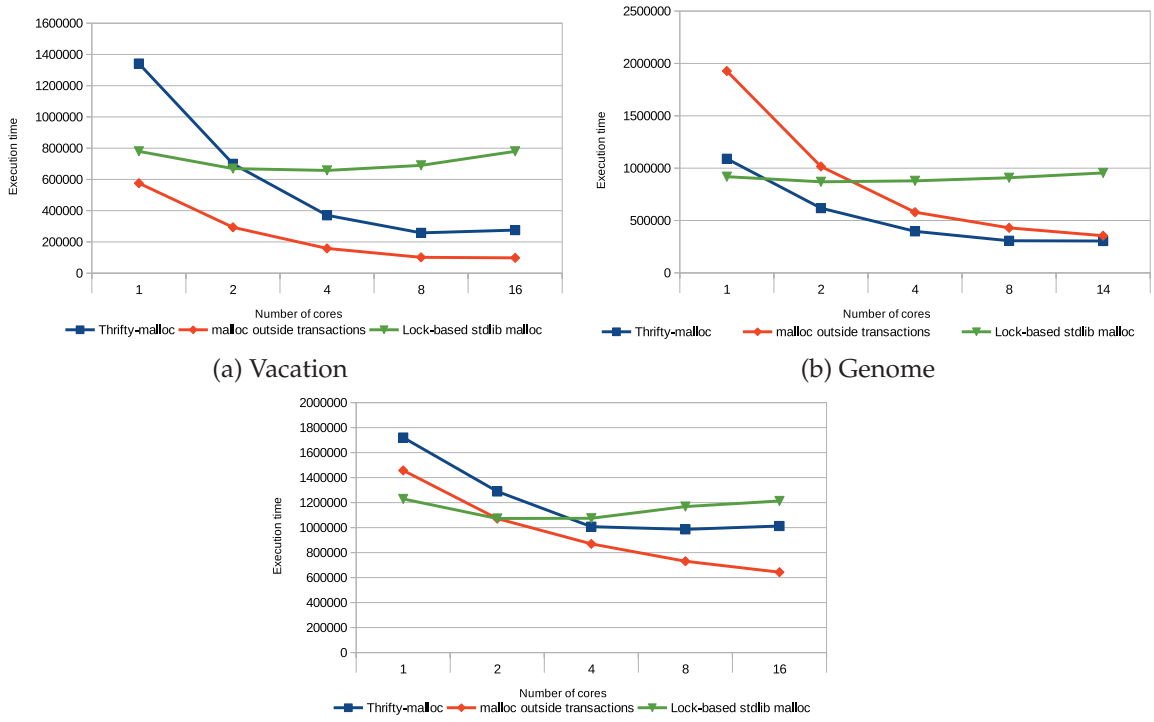
FIGURE 2 – Méthode de rechargement quand *malloc* est appelé dans une transaction

peut être servi, la transaction en cours est annulée de la même manière que pour un conflit. Les données modifiées par cette transaction sont restaurées à leur valeur d'avant le début de la transaction. Lorsque cela est fait, au lieu de redémarrer la transaction directement (comme ce serait le cas pour un conflit), on exécute la routine de repli et ses deux transactions dédiées, puis on relance la transaction initiale. Là encore, afin de minimiser les recours à cette routine coûteuse en temps, on essaye d'optimiser la taille de la nouvelle zone de mémoire allouée au thread. Cela est fait en utilisant un drapeau qui permet de savoir si la routine a déjà été appelée dans le cadre de la transaction en cours. Si c'est le cas, on double la taille de la zone mémoire retournée au thread, et ce jusqu'à ce qu'elle soit suffisante pour servir le *malloc()*. Ce schéma de rechargement de la zone de mémoire privée d'un thread est illustré dans la figure 2. L'intérêt principal de sortir de la transaction courante pour exécuter la routine de repli est qu'une fois la routine exécutée et sa deuxième transaction validée, les conflits susceptibles d'impacter la transaction courante n'annulent pas les modifications effectuées par la routine de repli, qui n'a donc pas à être réexécutée.

3.3. Plateforme d'évaluation

Les architectures many-coeurs sont composées de *clusters* fortement couplés, reliés entre-eux par un medium modulable tel qu'un réseau sur puce (NoC) [13, 14]. Les clusters sont composés d'un petit nombre d'unités de calcul (UC) simples et partageant de la mémoire et d'un medium d'interconnexion local à haute performance, comme illustré dans la figure 1(b). Dans notre architecture, chaque cluster comprend 16 UC RISC32, chacune équipée d'un cache d'instruction privé. Elles communiquent via une mémoire de données fortement couplée (TCDM) multi-bancs et multi-ports. Il s'agit d'un scratchpad partagé auquel les processeurs sont connectés par un interconnect logarithmique (garantissant une faible latence d'accès à la mémoire e.g. réseau omega).

Ce modèle architectural capture les caractéristiques clés des many-coeurs basés sur des clusters existants, tels que le STMicroelectronics STHORM [14], le Kalray MPPA [13], le TI Keystone II [2], l'Adapteva Parallela [1] ou le Toshiba Energy Efficient Many-Core [3], en termes d'organisation des coeurs, du nombre de clusters, du système d'interconnexion et de hiérarchie mémoire. Notre plateforme d'évaluation est un simulateur SystemC cycle-accurate, basé sur la plateforme de prototypage *VirtualSoC* [5]. Plus spécifiquement, nous nous concentrons sur un cluster de cette architecture. Le tas s'étend sur tous les bancs mémoire du TCDM de ce cluster, et les zones de mémoire allouées dynamiquement peuvent résider physiquement dans n'importe lesquels de ces bancs. Les processeurs se synchronisent par le biais d'opérations standard de lecture et d'écriture à certaines adresses particulières du TCDM, implémentant une sémantique



(a) Vacation

(b) Genome

(c) Patricia

FIGURE 3 – Temps d'exécution pour les 3 applications

test-and-set. Des mécanismes de verrou sont implémentés par dessus ces addresses. De plus, chaque cluster est équipé d'un support pour la mémoire transactionnelle matérielle (HTM). Nous implémentons pour cela le design défini par Papagiannopoulou *et al.* [18], qui vise la même classe de clusters à mémoire partagée que nos travaux. Ce système HTM se base sur une logique de détection et de résolution de conflits distribuée, implémentée dans les contrôleurs des bancs mémoire, ce qui permet une gestion des transactions rapide et modulable [17].

4. Evaluation

Nous évaluons Thrifty-malloc sur trois applications provenant des benchmarks STAMP [16] et MiBench [9] : *vacation*, *genome* et *patricia*. Nous les avons sélectionnées car elles sont bien connues dans la communauté de la mémoire transactionnelle et font appel à de la gestion dynamique de mémoire à l'intérieur de leurs transactions. Notre évaluation mesure d'une part la performance temporelle et la consommation de mémoire de ces applications dans trois cas caractéristiques : le cas où Thrifty-malloc est appelé à l'intérieur des transactions (labélisé "Thrifty"), le cas où les applications sont réécrites pour que Thrifty-malloc soit appelé seulement en dehors des transactions ("malloc outside"), et enfin le cas où les applications sont synchronisées avec des verrous, et où on utilise une version standard de malloc ("C_malloc"). On mesure d'abord les performances dans des cas ne nécessitant pas l'exécution de la routine de repli. Dans un second temps, on mesurera le coût temporel de cette exécution.

4.1. Performance brute

Les courbes de la figure 3 présentent les performances obtenues pour les 3 applications dans les 3 cas énoncés précédemment, sans que la routine de repli ne soit exécutée. La première observation que nous pouvons faire est que lorsque le nombre de coeurs augmente, l'utilisation de transactions (courbes rouges et bleues) offre toujours de meilleures performances que les verrous (courbes vertes), allant même jusqu'à plus de deux fois plus vite pour certaines appli-

cations. On remarque que pour *vacation* et *patricia*, les performances obtenues lorsque les appels à malloc sont effectués en dehors des transactions sont légèrement meilleures que lorsqu'on appelle malloc dans les transactions. En réalité ces résultats sont biaisés car lors de la réécriture de ces applications pour sortir les appels à malloc des transactions, nous les avons placés tout au début du programme dans la phase d'initialisation. Or, nous n'avons pas inclus cette phase dans la mesure des performances. On peut néanmoins constater que l'écart de performance est faible, et que le fait d'appeler malloc à l'intérieur des transactions consomme entre 20% et 80% moins de mémoire selon l'application (cf. figure 4). Notre approche offre donc le meilleur compromis performance/consommation de mémoire.

	Initial pool size of last core			
	1024 bytes	512 bytes	256 bytes	128 bytes
Vacation				
Thrifty-malloc	100%	100%	101%	109%
C malloc()	265%	265%	265%	265%
Genome				
Thrifty-malloc	100%	100%	100%	103%
C malloc()	281%	281%	281%	281%
Patricia				
Thrifty-malloc	100%	99%	112%	109%
C malloc()	120%	120%	120%	120%

TABLE 1 – Performances relatives à une exécution sans rechargement

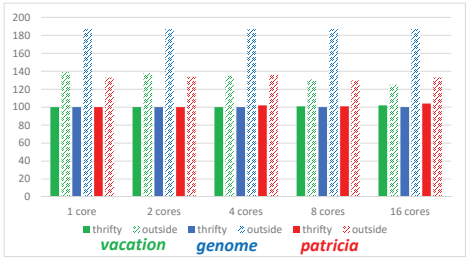


FIGURE 4 – Consommation de mémoire

4.2. Surcoût de rechargement des zones de mémoire privées

Le tableau 1 contient les mesures du surcoût nécessaire au rechargement d'une zone mémoire. Pour obtenir ces résultats, on a exécuté les trois applications sur 16 coeurs en réduisant volontairement la taille de la zone de mémoire allouée à l'un des threads pour le forcer à exécuter la routine de repli. Nous donnons les résultats pour plusieurs tailles initiales de zone de mémoire, en pourcentage du temps d'exécution brut (i.e. sans rechargement de zone de mémoire). Nous rajoutons pour comparaison le temps brut pris par la version synchronisée par verrous. On constate que dans le pire des cas, le surcoût temporel est de 12% du temps d'exécution par rapport à la version sans rechargement. Ces 12% restent inférieurs aux 20% de surcoût que l'on obtient si l'on synchronise l'application avec des verrous au lieu de HTM. Globalement, le coût de rechargement (y compris lorsque la routine de repli est exécutée à plusieurs reprises) est toujours négligeable par rapport aux performances obtenues en synchronisant avec des verrous. En conséquence, il est préférable d'utiliser des gestionnaires de mémoire compatibles avec les HTM (comme Thrifty-malloc) dans un contexte d'architecture multicoeur équipée d'HTM.

5. Conclusion

Nous avons présenté une méthode inédite de gestion dynamique de mémoire pour les systèmes embarqués multi/many-coeurs synchronisés par HTM. Nous avons évalué sa performance sur des applications de benchmark et montré que le faible surcoût lié au rechargement des zones de mémoire privées est un compromis acceptable aux vues de l'amélioration de l'utilisation de la mémoire, et de la flexibilité que Thrifty-malloc apporte au système. À l'avenir, nous prévoyons de poursuivre ces travaux dans les directions suivantes :

— Développer des techniques permettant de maintenir une répartition équitable de la mémoire libre entre les différentes zones privées.
— Étendre notre méthode de gestion de mémoire aux applications s'exécutant sur plusieurs clusters de calcul.
— Développer une technique de virtualisation de la mémoire compatible avec la HTM, basée sur l'utilisation d'un cache logiciel implémenté dans l'espace de mémoire transactionnelle, et communicant avec la mémoire L3 par des appels de DMA.

Bibliographie

1. Adapteva parallela. – http://www.adapteva.com/epiphany-multicore-intellectual-property/.
2. TI Keystone II. – http://www.ti.com/lit/ds/symlink/66ak2h12.pdf.
3. Toshiba energy efficient many-core. – http://www.aspdac.com/aspdac2014/technical_program/pdf/4A-4.pdf.
4. Berger (E. D.), McKinley (K. S.), Blumofe (R. D.) et Wilson (P. R.). – Hoard : A scalable memory allocator for multithreaded applications. *SIGOPS Oper. Syst. Rev.*, décembre 2000.
5. Bortolotti (D.), Pinto (C.), Marongiu (A.), Ruggiero (M.) et Benini (L.). – Virtualsoc : A full-system simulation environment for massively parallel heterogeneous system-on-chip. – In *ISPDP*, 2013.
6. Dice (D.) et Garthwaite (A.). – Mostly lock-free malloc. *SIGPLAN Not.*, juin 2002.
7. Dice (D.), Lev (Y.), Marathe (V. J.), Moir (M.), Nussbaum (D.) et Olszewski (M.). – Simplifying concurrent algorithms by exploiting hardware transactional memory. – In *SPAA*, 2010.
8. Ghemawat (S.) et Menage (P.). – TCMalloc : Thread-caching malloc. – http://goog-perftools.sourceforge.net/doc/tcmalloc.html.
9. Guthaus (M. R.), Ringenberg (J. S.), Ernst (D.), Austin (T. M.), Mudge (T.) et Brown (R. B.). – Mibench : A free, commercially representative embedded benchmark suite. – In *WWC-4*, 2001.
10. Herlihy (M.) et Moss (J. E. B.). – Transactional memory : Architectural support for lock-free data structures. – In *ISCA*, pp. 289–300, 1993.
11. Hudson (R. L.), Saha (B.), Adl-Tabatabai (A.-R.) et Hertzberg (B. C.). – McRT-Malloc : A scalable transactional memory allocator. – In *ISMM*, 2006.
12. Intel Corporation. – Hardware Lock Elision Overview. https://software.intel.com/en-us/node/683688.
13. Kalray. – MPPA 256. – www.kalray.eu/products/mppa-manycore/mppa-256/.
14. Melpignano (D.), Benini (L.) et Flamand (E.). – Platform 2012, a many-core computing accelerator for embedded SoCs : performance evaluation of visual analytics applications. – In *DAC*, 2012.
15. Michael (M. M.). – Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, juin 2004.
16. Minh (C. C.), Chung (J.), Kozyrakis (C.) et Olukotun (K.). – Stamp : Stanford transactional applications for multi-processing. – In *IISWC*, Sept 2008.
17. Papagiannopoulou (D.), Marongiu (A.), Moreshet (T.), Benini (L.), Herlihy (M.) et Bahar (I.). – Playing with fire : Transactional memory revisited for error-resilient and energy-efficient mpsoc execution. – In *GLSVLSI*, 2015.
18. Papagiannopoulou (D.), Moreshet (T.), Marongiu (A.), Benini (L.), Herlihy (M.) et Iris Bahar (R.). – Speculative synchronization for coherence-free embedded numa architectures. – In *SAMOS 2014*, July 2014.
19. Shavit (N.) et Touitou (D.). – Software transactional memory. – In *PODC*, 1995.
20. Wilson (P.), Johnstone (M.), Neely (M.) et Boles (D.). – Dynamic storage allocation : A survey and critical review. *In : Memory Management*. – Springer, 1995.