Faculté des Sciences et Ingénierie

Ajout d'un méchanisme d'éviction dans le cache noyau BMC

Rapport de stage M1

Baptiste Pires, Leif Henriksen Larez

Encadrant : Julien Sopena, Référent : Jonathan Lejeune

11 octobre 2021

Nous tenions à remercier M. Julien Sopena pour
l'opportunité qu'il nous a offert
ainsi que M. Yoann Ghigoff pour nous avoir accompagné.

Table des matières

1 Introduction

Memcached est un système de gestion de mémoires caches distribuées. Il est open source et gratuit. Il est souvent utilisé par les sites internet pour améliorer leur temps de réponse. Son fonctionnement repose sur un système de clé-valeur qui permet de stocker des données résultantes d'appels à une base de données ou d'une API par exemple. On comprend donc que Memcached doit être performant et doit répondre rapidement aux applications lui faisant des requêtes. Nous allons maintenant présenter BMC [1] (Berkeley Memcached) qui est un cache situé dans le noyau Linux pour les requêtes de Memcached.

BMC étant situé dans le noyau de Linux, il a été développé avec eBPF(Extended Berkeley Packer Filter) pour assurer la sécurité du noyau. Il fonctionne comme Memcached, c'est un cache clé-valeur. Son objectif est d'améliorer la latence de Memcached en traitant les requêtes avant qu'elles n'atteignent la pile réseau de Linux où des embouteillages de requêtes ont lieux. Le fait qu'il a été développé avec eBPF le rend portable et son utilisation ne requiert aucune modification du noyau Linux ou de Memcached.

Le cache BMC étant de taille limitée, le placement des clés ainsi que leur éviction sont importants. En effet, des clés peu demandées par les applications (dites froides) ne doivent pas occuper une grande partie du cache au détriment des clés plus demandées (dites chaudes). La mission de notre stage a été d'améliorer les performances de BMC. Pour le faire, nous avons dans un premier temps modifié l'algorithme d'éviction en rajoutant un système de points de vie pour les emplacements du cache. Dans un second temps nous avons analysé et fait des simulations de BMC s'il était un cache associatif.

Dans ce rapport nous allons en premier lieu présenter et comparer les algorithmes d'éviction, en deuxième lieu nous allons analyser la distribution des clés dans les ensembles d'un cache associatif (en nous concentrons sur les clés chaudes), puis nous allons introduire un algorithme *Least Recently Used* (ainsi qu'une version avec un challenger) pour le cache associatif pour finalement conclure.

1. Yoann Ghigoff et al. « BMC : Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing ». In : *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, avr. 2021, p. 487-501. isbn : 978-1-939133-21-2. url : `https://www.usenix.org/conference/nsdi21/presentation/ghigoff`.

2 Les Algorithmes de BMC

L'algorithme de BMC actuel a un mécanisme d'éviction très simple qui consiste a mettre à jour le cache à chaque fois que Memcached envoie une réponse. Ce mécanisme est correct, mais il ne prend pas en compte le fait que certaines clefs sont plus sollicitées que d'autres, il va donc remplacer les clef chaudes (les plus sollicitées) par des clefs froides (les moins sollicités), diminuant ainsi la performance du cache. Pour éviter l'éviction des clefs chaudes nous introduisons l'algorithme dit challenger qui utilise un système de points de vie pour protéger les clefs chaudes. Avec les clef chaudes protégés leur nombre augmente et avec lui le taux de HIT.

Dans la suite nous allons présenter et comparer les deux algorithmes, l'algorithme actuel et l'algorithme challenger.

2.1 Algorithme actuel

L'algorithme actuel de BMC a deux chaînes, la "incoming chain" et la "outgoing chain". la incoming chain traite les opérations GET et SET provenant du réseau, et la outgoing chain traite les messages provenant de Memcached.

SET : L'entrée qui vient d'être modifiée est invalide et la requête et transmise a Memcached.

GET : Si l'entre sollicité est présent dans le cache, alors elle est renvoyé, sinon, la requête est transmise a Memcached.

UPDATE : L'opération UPDATE a lieu a chaque fois que Memcached renvoie une réponse, dans l'UPDATE BMC met a jour le cache en copiant la réponse qui a été envoyé pas Memcached dans le cache.

Algorithme 1 : BMC actuel

```
   input : Context ctx
   // Incoming chain
 1 case GET do
 2 |   hash ← get_hash(ctx.key);
 3 |   entry ← cache[hash];
 4 |   if entry.key == ctx.key AND entry.valid == true then
   |   |   // If there is a hit
 5 |   |   send_reply(entry);
 6 |   end
 7 end
 8 case SET do
 9 |   hash ← get_hash(ctx.key);
10 |   entry ← cache[hash];
11 |   entry.valid ← false;
12 end
   // Outgoing chain
13 case UPDATE do
14 |   hash ← get_hash(ctx.key);
15 |   entry ← cache[hash];
16 |   if entry.key == ctx.key AND entry.valid == true then
   |   |   // The cache is up to date, no need to update.
17 |   |   return ;
18 |   else
   |   |   // Either the cache is invalid or we need to replace the entry.
19 |   |   entry.key ← ctx.key;
20 |   |   entry.data ← ctx.data;
21 |   |   entry.valid ← true;
22 |   end
23 end
```

2.2 Algorithme challenger

L'algorithme challenger est très similaire à l'algorithme actuel, la différence la plus importante est que, pour mettre à jour une entrée elle doit être soit invalide, soit morte (points de vie <= 0). L'idée est que, plus une clef est chaude plus elle est difficile à tuer, nous allons donc augmenter le nombre de clefs chaudes dans le cache et par conséquent augmenter le taux de HIT.

SET : Idem que pour BMC actuel.

GET : Idem que pour BMC actuel, mais, s'il y a un HIT les points de vie sont mis au maximum et le challenger est enlevé.

UPDATE : Le cache est mis à jour seulement si l'entrée touchée est soit morte (points de vie <= 0), soit invalide. Si l'entrée est valide et vivante, soit nous mettons à jour son challenger, soit nous réduisons ses points de vie.

Algorithme 2 : BMC challenger

input : Context ctx

```
// Incoming chain
case GET do
    hash ← get_hash(ctx.key);
    entry ← cache[hash];
    if entry.key = ctx.key AND entry.valid = true then
        // If there is a hit
        entry.hp ← MAX_HP;
        entry.challenger ← NULL;
        send_reply(entry);
    end
end
case SET do
    hash ← get_hash(ctx.key);
    entry ← cache[hash];
    entry.valid ← false;
end
// Outgoing chain
case UPDATE do
    hash ← get_hash(ctx.key);
    hash2 ← get_hash2(ctx.key);
    entry ← cache[hash];
    if entry.key = ctx.key AND entry.valid = true then
        // The cache is up to date, no need to update.
        return ;
    end
    // Either the cache entry is invalid or we need to replace the entry.
    if entry.valid = true AND entry.challenger ≠ hash2 AND entry.hp > 0 then
        // New challenger.
        entry.challenger ← hash2;
        return ;
    else if entry.valid = true AND entry.challenger = hash2 AND entry.hp > 0 then
        // I am the challenger, I can decrease entry's hp.
        entry.hp ← entry.hp − 1;
        if entry.hp > 0 then
            // entry is not dead yet, do not update.
            return ;
        end
    end
    // Either the entry is invalid or it is dead.
    // In both cases update the cache.
    entry.key ← ctx.key;
    entry.data ← ctx.data;
    entry.valid ← true;
    entry.hp ← MAX_HP;
    entry.challenger ← NULL;
end
```

2.3 Résultats

Pour comparer les deux algorithmes nous avons créé un test qui enregistre l'état du cache et le taux de HIT toutes les n requêtes. Les données enregistrées nous permettront d'analyser comment le cache est rempli dans le temps et aussi de voir l'évolution du taux de HIT.

Informations sur le cache utilisé

- Taille du cache : 3250000 entrées
- Nombre de clefs : 30M

— Découpage tout les 10M requêtes, nous enregistrons l'état du cache tout les 10M de requêtes.
— Compilateur bpf : clang-9
— Compilateur c : gcc 8.4.0
— Linux kernel : 5.3

Informations sur les températures
— HOT : premier 5% des clefs
— WARM : les 10% suivantes (de 5 à 15 %)
— MILD : les 30% suivantes (de 15 à 45 %)
— COLD : le reste (de 45 à 100 %)

Nous avons lancé le test avec BMC actuel et puis avec BMC challenger avec 1, 2, 3, 4, et 100 points de vie (HP).

2.3.1 Test de base

Dans ce test nous avons envoyé 1000M de requêtes a un cache BMC qui était vide au départ. Les fréquences des clefs suivent la loi de Zipf pour reproduire une charge réaliste sur BMC. [2] Tout les 10M de requêtes un snapshot du cache est pris pour pouvoir l'analyser par la suite.

L'objective de ce test est de voir l'évolution de la distribution des clef dans le cache et du taux de HIT au fils du temps.

Analyse des résultats

Nous pouvons voir dans la figure 1a et 1b que les résultats avec l'algorithme actuel et le challenger avec un seul point de vie sont les mêmes. Le taux est de HIT de 76.5% et la distribution de clefs ne change pas.

Dans la figure 1c avec 2 points de vie, nous observons une amélioration du taux de HIT qui monte à 81.6 % et le nombre de clefs chaudes monte aussi. Avec 3 points de vie (figure 1d), le taux de HIT augmente encore à 82%. Avec 4 points de vie (figure 1e), le taux de HIT ne augmente plus et il tombe à 81.7%, finalement avec 100 points de vie (figure 1f), le taux de HIT tombe à 78.6%.

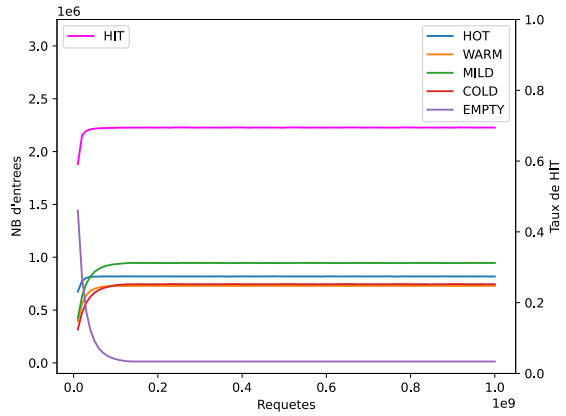
HP	Taux HIT	HOT	WARM
1	76.5	818K	729K
2	81.6	996K	725K
3	82	1141K	803K
4	81.7	1098K	710K
100	78.6	820K	728K

TABLE 1 – Resultats BMC challenger

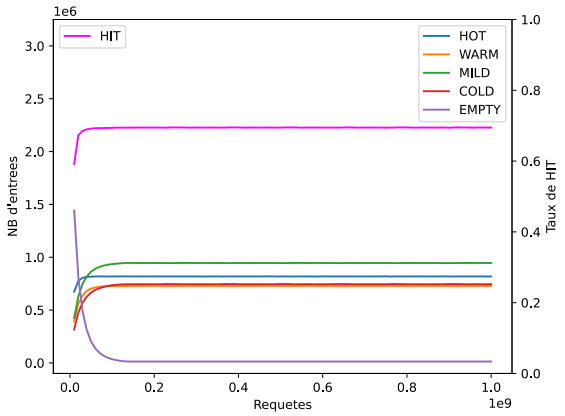
Dans la table 1 nous pouvons remarquer que le taux de HIT suit le nombre de clefs chaudes. Jusqu'à trois points de vie, le nombre de clefs chaudes augmente mais après il commence à diminuer. Alors que les points de vie sont faits pour protéger les clefs chaudes, ils protègent aussi les autres clefs, donc un nombre trop important de points de vie peut avoir un effet négatif en laissant en dehors du cache les clefs chaudes.

Après les tests nous avons remarqué deux phénomènes, le premier, c'est que la distribution des clefs dans le cache est stable dans le temps après 200M de requêtes, la raison de ce phénomène est probablement le fait que les clefs froides ou même tièdes sont trop bien protégées, il est donc impossible pour les clefs chaudes de les évincer. Le deuxième, c'est que le taux de HIT est périodique, vu que la distribution des clefs ne change pas nous supposons que la périodicité est dû au générateur aléatoire de clefs.

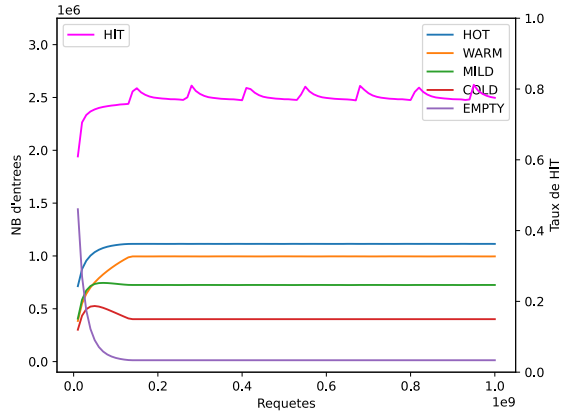
2. Berk ATIKOGLU et al. « Workload Analysis of a Large-Scale Key-Value Store ». In : *SIGMETRICS Perform. Eval. Rev.* 40.1 (juin 2012), 53–64. ISSN : 0163-5999. DOI : 10.1145/2318857.2254766. URL : https://doi.org/10.1145/2318857.2254766.



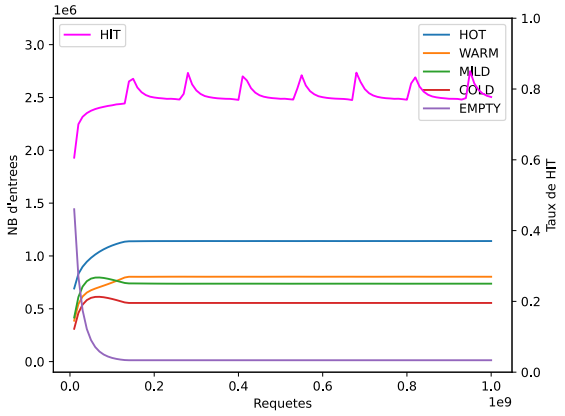
(a) BMC actuel



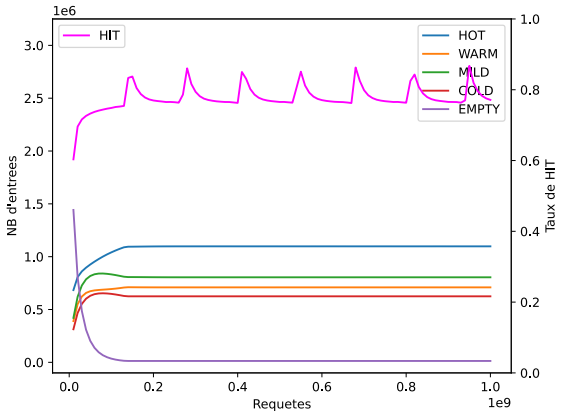
(b) BMC challenger : HP = 1



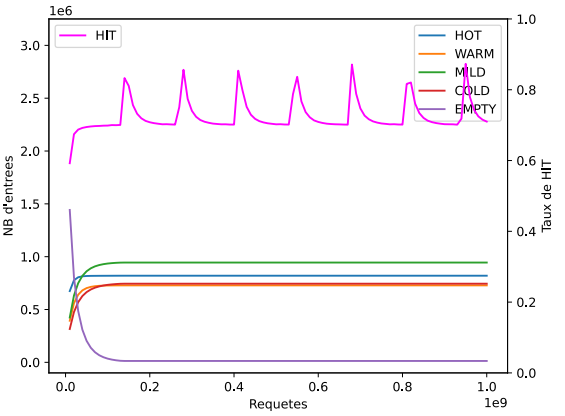
(c) BMC challenger : HP = 2



(d) BMC challenger : HP = 3



(e) BMC challenger : HP = 4



(f) BMC challenger : HP = 100

2.3.2 Test de convergence

Dans le test précédent nous avons vu que la distribution des clefs ne change pas après 200M clef et que c'est probablement du au fait que les clefs froides sont trop protégées donc une fois dans le cache c'est impossible de les évincer. Nous avons donc fait un test identique au précédent, mais cette fois ci nous avons rempli le cache de clefs froides avant de commencer. Si notre hypothèse est vraie les clefs froides vont obstruer l'entrée des clefs chaudes et comme conséquence nous aurons moins de clefs chaudes et un pire taux de HIT.

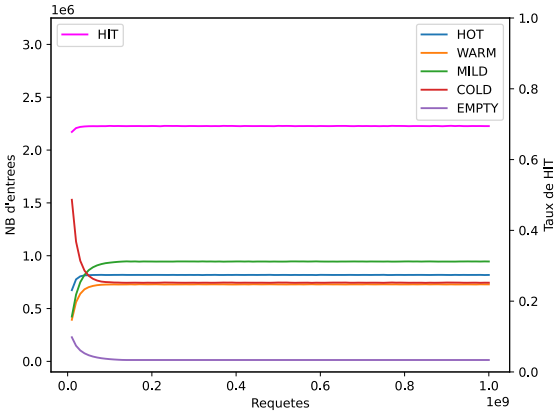
Analyse des résultats

Les résultats (figures 2a, 2b, 2c, 2d, 2e) sont similaires au test précédents mais le nombre de clefs WARM est plus petit, après pour le test avec 100 points de vie nous voyons une chute considérable du nombre de clefs chaudes.

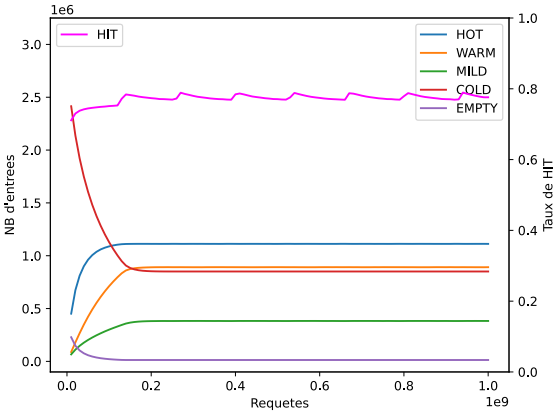
HP	HOT	WARM
1	818K	729K
2	1112K	891K
3	1131K	524K
4	1067K	287K
100	183K	126K

Table 2 – Resultats BMC challenger, cold start

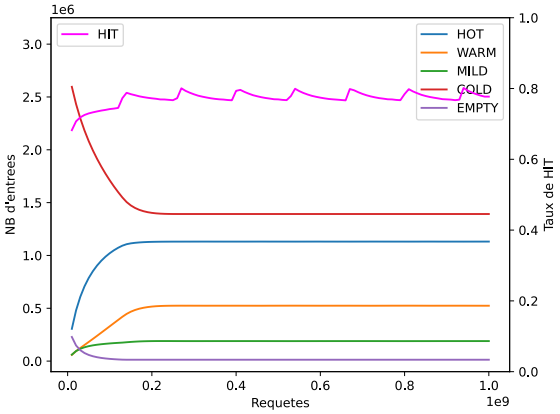
Nous considérons toujours que la perte importante de clefs chaudes dans le test avec 100 points de vie, comme pour le test précédent, est dû au fait que les clefs froides sont trop protégées, elles deviennent donc impossible à évincer.



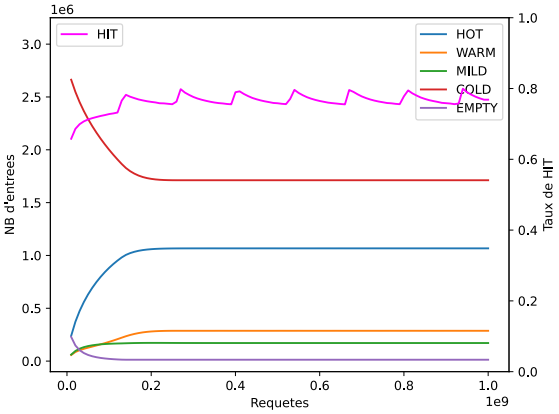
(a) BMC challenger, cold start : HP = 1



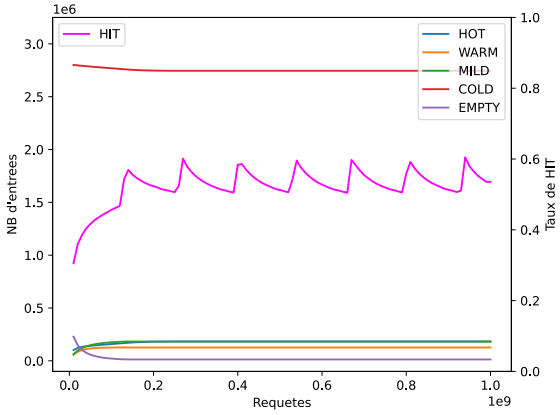
(b) BMC challenger, cold start : HP = 2



(c) BMC challenger, cold start : HP = 3



(d) BMC challenger, cold start : HP = 4



(e) BMC challenger, cold start : HP = 100

3 Cache associatif

Comme nous venons de le voir dans la partie précédente, les clefs froides peuvent grandement gêner les clefs chaudes dans le cache rendant une partie du cache inutile. Il faudrait améliorer la distribution des clefs chaudes dans celui-ci pour qu'à tout moment il ne contienne que des clefs chaudes.

Dans cette optique, nous avons analysé la distribution des clefs ainsi que le taux de hits de ce dernier s'il était un cache associatif. Un cache associatif (N-way set associative cache) est un cache dont les emplacements sont répartis dans divers ensembles. L'objectif de ce cache serait de permettre à des clefs chaudes de "cohabiter" dans des ensembles sans qu'elles ne s'évincent entre elles et qu'une clef froid ne reste pas dans un emplacement. Nous verrons dans un premier temps la distribution des clefs dans les ensembles puis nous nous intéresserons à deux implémentations d'algorithmes possibles pour ce type de cache ainsi que leurs performances.

3.1 Distribution des clefs dans les ensembles

Pour l'analyse suivante, nous prenons en compte un total de 70 millions de clefs, avec *5%* (3250000) de clefs chaudes. Des mesures ont été réalisées avec un cache de 3250000 emplacements et 1400000 entrées. Nous avons pris les mêmes valeurs utilisées dans le document *BMC state and future work*.

Degré associativité	Total ensembles	clefs chaudes / ensemble
2	700000	4.64
4	350000	9.28
6	233334	13.92
8	175000	18.57
10	140000	23.21

TABLE 3 – Détail des différents degrés d'associativité pour un cache de 1400000 emplacements et 70000000 de clefs.

Comme nous pouvons le voir dans la table 3, avec un cache ayant 1400000 emplacements et 70000000 clefs, même dans le cas où les clefs chaudes sont distribuées équitablement à travers les différents ensembles, il y aura toujours plus de clefs chaudes pour un ensemble donné que d'emplacements dans celui-ci (il y a un facteur 2.32, pour un emplacement il y a 2.32 clefs chaudes qui pourraient y être stockées).

Degré associativité	Total ensembles	clefs chaudes / ensemble
2	1400000	2.32
4	700000	4.64
6	466667	6.96
8	350000	9.28
10	280000	11.60

TABLE 4 – Détail des différents degrés d'associativité pour un cache de 2800000 emplacements et 70000000 de clefs.

En doublant la taille du cache cela a logiquement pour effet de doubler le nombre total d'ensembles et réduit donc la probabilité que deux clefs entrent en collision. De ce fait, chaque ensemble subit moins de "pression" de la part des clefs chaudes. Étant donné que le nombre d'ensembles est doublé, le nombre de clefs chaudes (toujours en prenant en compte une distribution équitable) est divisé par deux.

Degré associativité	Total ensembles	clefs chaudes par ensemble
2	3500000	0.92
4	1750000	1.85
6	1166667	2.78
8	875000	3.71
10	700000	4.64

TABLE 5 – Détail des différents degrés d'associativité pour un cache de 7000000 emplacements et 70000000 de clefs.

C'est à partir de cette taille de cache qu'un cache *nway* pourrait devenir intéressant. En effet, les clefs chaudes représenteraient moins de la moitié des clefs présentes dans l'ensemble. De ce fait, cela permettrait à des clefs

froides voire tièdes d'entrer en collision avec des clefs chaudes sans les remplacer. Elles pourraient prendre des emplacements faisant partie du même ensemble que celle-ci. Ces données seraient idéales mais restent très peu probables dans les faits, en effet nous verrons un peu plus loin dans cette partie la distribution réelle des clefs chaudes dans les différents ensembles.

Avant de modifier BMC pour en faire un cache associatif, nous avons étudié la distribution des clefs dans les ensembles en faisant varier le taux d'associativité ainsi que la taille du cache.

La figure 3 représente la distribution des clefs dans les différents ensembles en fonctions du degré d'associativité du cache. Comme nous pouvons le voir, plus ce dernier est élevé, plus les ensembles se partagent un nombre élevé de clefs. Cela est expliqué par le fait que pour la même taille d'un cache, plus le degré d'associativité sera élevé, moins il y aura d'ensembles au total, ils contiendront seulement plus d'entrées.

Nous avons ensuite analysé la distribution des clefs chaudes pour différentes tailles de cache en faisant également varier le degré d'associativé. Pour un total de 70 millions de clefs, il y a 5% de clefs chaudes, donc 3.5 millions. L'idéal serait d'avoir un cache avec 3.5 millions d'ensembles où chacun aurait une seule clef chaude. Mais ce scénario étant peu probable, des ensembles se retrouvent avec plus de clefs chaudes que d'emplacements et d'autres sans.

Les tableaux 6 et 7 contiennent différentes valeurs statistiques qui montre la répartition des clefs chaudes dans les ensembles. Par exemple, dans le tableau 6, on peut voir que pour un cache d'une taille de 1400000 entrées et un degré d'associativité de 2, en moyenne il y a 5 clefs chaudes par ensemble. La colonne $\emptyset$ correspond au nombre d'ensemble sans aucune clef chaude. Dans le cas présent, il y a un total de 700000 ensemble (se référer au tableau 3), il y a donc 0.63% des ensembles qui n'ont pas de clefs chaudes.

Pour plus de détails concernant la distribution des clefs dans le cache, nous pouvons regarder les graphiques de la figure 4. On peut voir sur le graphe 4a que la distribution des clefs est proche d'une gaussienne, ce qui peut indiquer que les clefs chaudes sont bien distribuées dans les ensembles.

En augmentant la taille du cache, on voit que la part d'ensemble sans clefs chaudes augmente (0.6% pour un 1.4M, 8.08% pour 2.8M et 36.51% pour 7M), en augmentant la taille du cache d'un facteur de 5 (1.4M à 7M), le taux d'emplacement sans clef chaude a été multiplié par 60. Comme nous pouvons le voir sur le graphe 4c, la distribution n'est plus du tout proche d'une gaussienne.

	1400000							2800000						
	Moyenne	σ	Q1	Médiane	Q3	D9	$\emptyset$	Moyenne	σ	Q1	Médiane	Q3	D9	$\emptyset$
2	5	2.22	3	5	6	8	4433	2.5	1.57	1	2	3	5	113157
4	10	3.13	8	10	12	14	14	5	2.22	3	5	6	8	4433
6	15	3.87	12	15	18	20	0	7.5	2.73	6	7	9	11	264
8	20	4.44	17	20	23	26	0	10	3.13	8	10	12	14	14
10	25	5.06	21	25	28	32	0	12.5	3.58	10	12	15	17	1

TABLE 6 – Répartition des clefs chaudes dans les ensembles de caches contenant 1400000 et 2800000 entrées en fonction du degré d'associativité.

	7000000						
	Moyenne	σ	Q1	Médiane	Q3	D9	$\emptyset$
2	1	0.99	0	1	1	2	1278008
4	2	1.4	1	2	3	4	233336
6	3	1.73	2	3	4	5	58131
8	4	1.98	3	4	5	7	15549
10	10	2.21	3	5	6	8	4433

TABLE 7 – Répartition des clefs chaudes dans les ensembles de cache contenant 7000000 entrées en fonction du degré d'associativité.

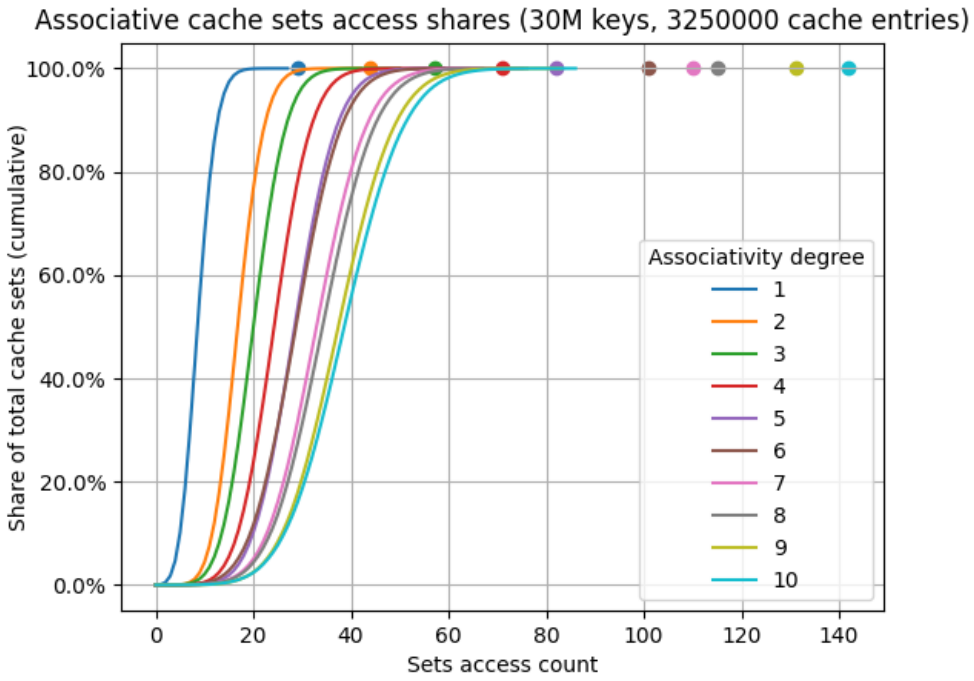
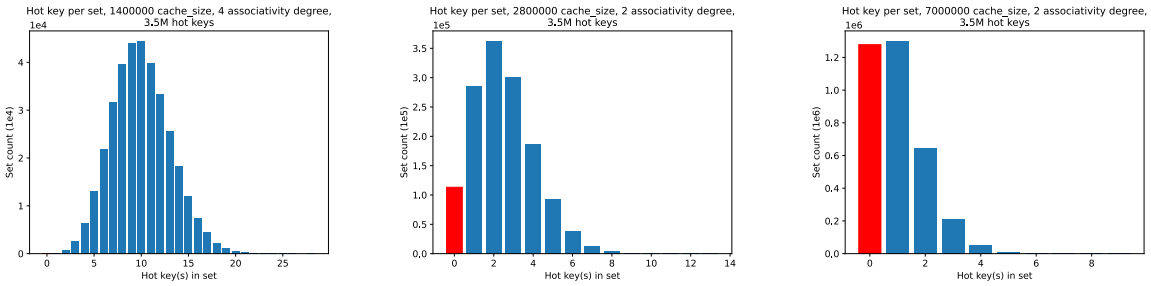


FIGURE 3 – Coubres cumulatives ... trouvez un titre



(a) 1400000 emplacements, 700000 ensembles

(b) 2800000 emplacements, 1400000 ensembles

(c) 7000000 emplacements, 3500000 ensembles

FIGURE 4 – Répartition des clefs chaudes en fonction de la taille du cache avec un degré d'associativité de 2.

3.2 Implémentation LRU

Les programmes eBPF étant restreints par différents paramètres, ils peuvent être compliqués à modifier. De ce fait, cet algorithme a pour le moment était seulement implémenté en C pour ne pas perdre de temps avec les contraintes d'eBPF. Les requêtes ont ensuite été simulées en suivant une distribution zipf. L'algorithme est disponible en annexe 3. [3]

Sachant qu'avec un cache associatif chaque ensemble possède plusieurs emplacements, il faut un moyen pour donner une priorité à chacun. Pour ce faire, nous avons simulé un cache avec un algorithme *Least Recently Used (LRU)*. Cette implémentation ne prend pas en compte un challenger, dès qu'il y a un conflit, l'entrée à la fin de la queue est remplacée par la nouvelle et celle-ci est mise au devant de la queue.

Des tests ont été réalisés pour différents degré d'associativité (2, 4, 6, 8, 10) ainsi que différentes tailles de cache (1.4M, 2.8M, 7M). Tous les résultats obtenus l'ont été en réalisant 1 milliard de requêtes *GET* avec un cache initialement vide.

Le tableau 8 contient les résultats de ces tests. La première ligne (*BMC basique*) correspond aux taux de

3. Les algorithmes ont été implémentés en essayant de se rapprocher de ce qui pourrait être écrit en code eBPF, c'est à dire en évitant les boucles imbriquées, en essayant de garder le code concis pour éviter le surcoût des *tail-call* et en itérant les tableaux à chaque fois pour simuler les *maps*.

hits récupérés dans le document *BMC state and future work* à la page 8. Comme les conditions de test n'étaient pas les mêmes que pour nos tests (Ces taux de hits ont été obtenus avec approximativement 10 milliards de requêtes *GET*), nous avons simulé un cache basique sans associativité pour l'utiliser comme repère. Les taux de hit lui correspondant se trouvent à la ligne *Cache bisque*.

Comme nous pouvons le voir dans le tableau 8, peu importe la taille du cache, un degré d'associativité de 2 a toujours le plus faible taux de hit (en moyenne 6.89% de moins que pour un degré de 4). Assez logiquement, le taux de hit augmente lorsque la taille du cache augmente étant donné que le nombre d'ensembles augmente proportionnellement à celle-ci. Lorsque la taille double (1.4M à 2.8M) le taux de hit augmente de 4.23% pour 2, 4.37% pour 4, 4.179% pour 8 et de 4.144% pour 10. La plus grande augmentation est pour un degré de 4, il en est de même lorsqu'on passe de 2.8M à 7M (5.845%). Il semblerait que 4 soit le degré d'associativité idéal.

Comme nous pouvons le voir sur les graphes de la figure 6 en annexe, les clefs chaudes sont mieux distribuées qu'avec un degré d'associativité de 2. Pour un cache d'une taille de 7M, il y a 13.33% d'ensembles sans clefs chaudes, c'est 23.18% de moins que pour un degré d'associativité de 2. Pour les deux autres tailles de cache, le taux d'ensembles non occupés est inférieur à 0.63%.

Les autres degrés d'associativités ne semblent pas intéressant sachant qu'ils augmentent le taux de hits de 0.456% (degré 8, taille de cache 1.4M) mais induisent une complexité qui pourrait poser problème avec le vérifieur d'eBPF. Sachant qu'il est possible lors d'une requête de scanner les emplacements plusieurs fois, si le degré d'associativité sera élevé cela peut résulter en explosion combinatoire pour le vérifieur. Si des boucles doivent être déroulées, cela va également bloquer le programme sachant qu'un programme eBPF est limité en instructions (4096 instructions).

Comme les résultats avec un degré d'associativité de 4 semblent les plus prometteurs et réalisables, nous avons réalisés des tests avec 10 milliards de requêtes. Les taux de hits obtenus sont à la ligne "4 (10 milliards req.)". Les taux de hits s'améliorent peu (sauf pour le cache de 2.8M d'entrées, il augmente de 2%). Etant donné que nous avons essayé de nous rapprocher au plus près des conditions de test de "BMC classique", ces résultats ne sont pas satisfaisant sachant qu'un cache associatif va rajouter un surcoût lors de l'exécution en plus de ne pas avoir des performances égales à l'algorithme courant.

Maintenant que nous avons implémenté et analysé l'algorithme LRU, nous allons essayer d'améliorer son taux de hits en y introduisant un challenger.

	1400000	2800000	7000000
BMC basique	70%	75%	83%
Cache basique	66.68%	70.88%	76.46%
2	62.45%	66.68%	72.10%
4	69.12%	73.49%	79.335%
4 (10 milliards req.)	69.14%	75.58%	79.55%
6	69.49%	73.75%	79.339%
8	69.571%	73.76%	79.20%
10	69.576%	73.72%	79.04%

TABLE 8 – Taux de hit en fonction du degré d'associativité et de la taille du cache.

3.3 Implémentation LRU-challenger

Dans un premier temps, nous avons implémenté un algorithme avec un *challenger*, reprenant le même principe que dans la section 2. Dans cette première version, le challenger est accepté si est seulement s'il est déjà challenger. C'est à dire qu'il doit faire deux requêtes consécutives pour pouvoir remplacer une donnée dans l'ensemble. Le détail de l'algorithme est disponible en annexe 4.

Le tableau 9 contient les résultats des tests que nous avons réalisé. Nous les avons réalisé dans les mêmes conditions que pour l'algorithme *LRU* basique c'est à dire avec 1 milliard de requêtes et un cache initialement vide. Les requêtes ont utilisé la même distribution zipf pour les clefs. Nous avons également intégré les données concernant le cache basique pour qu'elles servent de références.

Comme nous pouvons le voir dans le tableau 4, le degré d'associativité qui présente le meilleur taux de hits peu importe la taille du cache est 4. Les performances du cache avec un degré d'associativité de 2 sont expliqué

par le fait qu'il y a beaucoup trop de clefs chaudes par ensembles, il y a de ce fait beaucoup de conflits et les clefs candidates ont une probabilité plus faible de faire deux requêtes consécutives pour être insérées dans l'ensemble.

Etant donné que le cache avec un degré d'associativité de 4 présente les meilleures performances, nous allons nous concentrer sur celui-ci de nouveau. Les autres configurations ont des performances un tout petit peu meilleures voire moins bonnes qu'avec l'algorithme LRU initial.

Cet algorithme a un meilleur taux de hits avec les deux tailles de cache les plus petites (1.4M et 2.8M) mais pour le cache avec 7M d'emplacements son taux de hits baisse de 0.835%. Cela peut être expliqué par le fait qu'avec un cache de cette taille, il y en moyenne 2 clefs chaudes par ensembles. Le neuvième décile est de 4, ce qui signifie qu'au moins 90% des ensembles ont au maximum 4 clefs chaudes qui peuvent y être stockées. En lisant le graphique 6c de l'annexe, on peut voir que presque 95% de ensembles ont au maximum 4 clefs chaudes. Il y a donc peu de conflits entre les clefs chaudes et le concept de challenger perd de son utilité, il ne fait que retarder l'entrée des clefs dans les ensembles et les expose au risque de se faire prendre la place de challenger par une clé froide ou tiède. Nous n'avons pas eu le temps d'analyser les données pour trouver l'origine de cette baisse de performance, cela est purement théorique.

Nous avons essayé d'implémenter un système de point de vie pour cet algorithme également. Les performances ont grandement été affectées. En effet, pour un cache de degré 4M et 1.4M d'entrées le taux de hits était de 52.81% avec 1, 2 et 3 points de vie. Nous avons testé avec les autres tailles de cache, 2.8M et 7M et nous avons obtenu des taux de hits de 57.73% et 63.55% pour 1, 2 et 3 points de vie également. Par manque de temps nous n'avons pas réalisé d'autres tests concernant les points de vie mais cette piste nous semblait ne pas pouvoir offrir de meilleurs résultats.

	1400000	2800000	7000000
Cache basique	66.68%	70.88%	76.46%
2	67.67%	71.19%	75.54%
4	72.19%	75.08%	78.50%
6	71.34%	74.15%	77.36%
8	70.60%	73.38%	76.67%
10	70.25%	72.97%	76.20%

TABLE 9 – Taux de hit de l'algorithme en fonction du degré d'associativité et de la taille du cache.

4 Conclusion

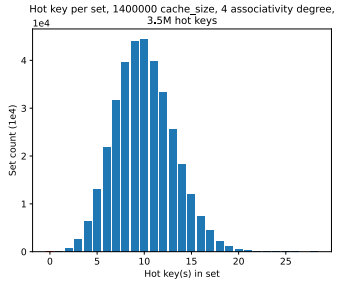
Durant le stage nous pu avons comparer l'algorithme classique avec l'algorithme dit challenger, les résultats étaient positifs avec une amélioration du taux de HIT notable (6% pour 3 points de vie). Nous nous sommes également rendus compte qu'il était nécessaire de tester les cas extrêmes avec beaucoup de points de vie car ils réduisent drastiquement le taux de HIT.

Comme nous pouvons le voir dans les graphes de la section 2, le taux de HIT prend une forme de vague qui s'accentue en fonction des points de vie. Cela pourrait s'expliquer par le générateur de clés qui se répéterai au bout d'un certain nombre de générations. Il serait intéressant d'analyser la génération aléatoire des clés pour comprendre cette forme.

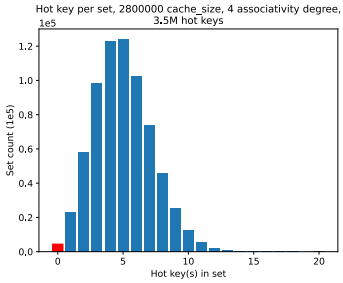
Concernant les résultats obtenus pour le cache associatif, avec les deux implémentations (parties 3.2 et 3.3) nous n'avons pas égalé le taux de HIT de BMC classique. Malgré l'analyse de la distribution des clés chaudes dans le cache, nous n'avons pas réussi à implémenter une version qui améliorerait le taux de HIT. Sachant qu'un cache associatif (peu importe son degré d'associativité) sera beaucoup plus complexe à mettre en oeuvre et sera très probablement contraint par le vérifier d'eBPF, il ne semble pas intéressant de modifier BMC pour en faire un cache associatif.

Ce stage a été très enrichissant pour nous étant donné qu'il nous a permis de rencontrer une utilisation réelle d'un programme eBPF suite à notre projet au cours de l'UE "Projet SAR" qui portait sur eBPF. Nous avons pu également mettre en pratique des connaissances acquises durant l'année de Master 1 concernant les caches associatifs.

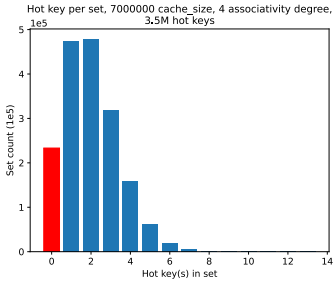
5 Annexes



(a) 1400000 emplacements, 350000 ensembles

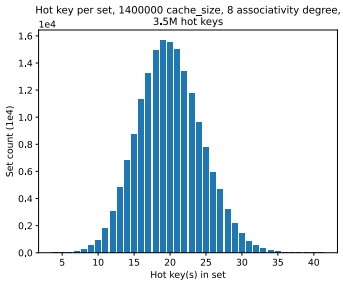


(b) 2800000 emplacements, 700000 ensembles

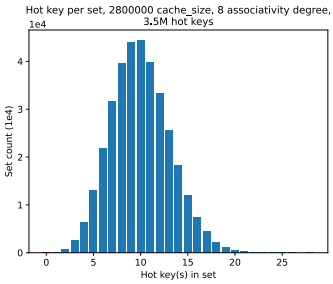


(c) 7000000 emplacements, 1750000 ensembles

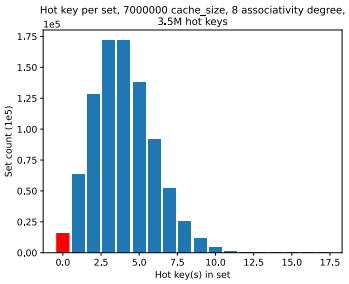
FIGURE 5 – Répartition des clés chaudes en fonction de la taille du cache avec un degré d'associativité de 4.



(a) 1400000 emplacements, 175000 ensembles



(b) 2800000 emplacements, 350000 ensembles



(c) 7000000 emplacements, 875000 ensembles

FIGURE 6 – Répartition des clés chaudes en fonction de la taille du cache avec un degré d'associativité de 8.

Algorithme LRU Cette algorithme utilise le champ *valid* de la structure *cache_ entry_ nway* qui représente une entrée dans le cache. Le membre *valid* de la structure est utilisé comme priorité dans l'ensemble S'il vaut 0, l'entrée est invalide, s'il est supérieur à 0 c'est sa position dans la l'ensemble, 1 étant la plus grand priorité (l'entrée la plus récemment utilisée). Lors d'un remplacement, l'entrée ayant le champ *valid* le plus élevé (égal au degré d'associativité du cache) sera remplacée. [4]

```
struct cache_entry_nway {
    char key[16];
    char data[16]; // Arbitrary length
    unsigned int len;
    unsigned int hash;
    unsigned char valid;
};
```

Algorithme LRU-Challenger Cette algorithme reprend le même principe que l'algorithme précédent (3) mais en implémentant un challenger. Pour ce faire il utilise une seconde structure, nommée *set_ meta* qui permet de stocker des méta-données des ensembles. Elle sera décrite en dessous de ce paragraphe. Une structure set allouée pour chaque ensemble.

```
struct set_meta {
    int hp;
    uint32_t challenger;
};
```

4. Dans les deux algorithmes des "." ont été utilisés pour accéder aux champs des pointeurs pour rendre le code plus lisible lors de l'affectation de valeurs. Ce qui donne $current.key \leftarrow NULL$ à la place de $current \rightarrow key' \leftarrow NULL$.

Algorithme 3 : Algorithme LRU

input : char[16] *key*, char[1024] *data*, int *associativity_degree*, int *set_count*

```
1  case GET do
2      hash ← get_hash(key);
3      cache_id ← (hash%set_count) * associativity_degree;
4      target ← NULL;
5      current ← &cache[cache_id];
       // Check if any entry is empty
6      for ( i = 0; i < associativity_degree; i++ ) do
7          current ← &cache[cache_id + i];
8          if current.valid == 0 then
9              target ← current;
10             goto replace;
11         end
12     endfor
       // We try to find our key in the current set
13     for ( i = 0; i < associativity_degree; i++ ) do
14         current ← &cache[cache_id + i];
15         if current.valid then
               // Potential cache hit
16             if current.hash == hash then
                   // hit, we place this entry at the head of the queue
17                 if current.key == key then
18                     target ← current;
19                     goto shift_lru;
20                 end
21             end
22         end
23     endfor
       // All entries are used and current key isn't stored yet so we replace last key in
          LRU's queue
24     for ( i = 0; i < associativity_degree; i++ ) do
25         current ← &cache[cache_id + i];
26         if current.valid == associativity_degree then
27             target ← current;
28             goto replace;
29         end
30     endfor
31     replace :
32     target.key ← key;
33     target.data ← data;
34     target.hash ← hash;
35     shift_lru :
36     target.valid ← 1;
       // Valid field's used to store priority of the entry, we increase all the entries
          currently in the set to put the new one as the most recently used
37     for ( i = 0; i < associativity_degree; i++ ) do
38         if cache[cache_id + i].valid! = associativity_degree then
39             cache[cache_id + i].valid++;
40         end
41     endfor
42 end
```

Algorithme 4 : Algorithme LRU-challenger

```
   input : char[16] key, char[1024] data, int associativity_degree, int set_count
 1 case GET do
 2     hash ← get_hash(key);
 3     hash2 ← get_hash2(key);
 4     cache_id ← (hash%set_count) ∗ associativity_degree;
 5     target ← NULL;
 6     current ← &cache[cache_id];
 7     current_set ← &set_meta[cache_id] // Check if any entry is empty
 8     for ( i = 0; i < associativity_degree; i++ ) do
 9         current ← &cache[cache_id + i];
10         target ← current;
11         goto replace;
12     endfor
13     for ( i = 0; i < associativity_degree; i++ ) do
14         current ← &cache[cache_id + i];
15         if current.valid then
               // Potential cache hit
16             if current.hash == hash then
                   // hit, we place this entry at the head of the queue
17                 if current.key == key then
18                     target ← current;
19                     goto shift_lru;
20                 end
21             else
22                 if current_set.challenger == hash2 then
23                     current_set.hp ← current_set − 1;
24                     if current_set.hp == 0 then
25                         current_set.hp ← start_hp;
26                         current_set.challenger ← NULL;
27                         goto replace ;
28                     else
29                         return ;// Current set still has hp(s), we can't replace any entry
                               right now
30                     end
31                 else
                       // Current challenger isn't me, set myself as challenger
32                     current_set.challenger ← hash2;
33                     current_set.hp ← start_hp;
34                     return ;
35                 end
36             end
37         end
38     endfor
       // All entries have data and current key isn't stored yet so we replace last key in
          LRU's queue
39     for ( i = 0; i < associativity_degree; i++ ) do
40         current ← &cache[cache_id + i];
41         if current.valid == associativity_degree then
42             target ← current;
43             goto replace;
44         end
45     endfor
46     replace : target.key ← key;
47     target.data ← data;
48     target.hash ← hash;
49     shift_lru : target.valid ← 1;
50     for ( i = 0; i < associativity_degree; i++ ) do
51         if cache[cache_id + i].valid! = associativity_degree then
52             cache[cache_id + i].valid++;
53         end
54     endfor
55 end
```