ComPAS'2013 : RenPar'21/ SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Vers un cache réparti adapté au cloud computing

Maxime Lorrillere, Julien Sopena, Sébastien Monnet et Pierre Sens

Université Pierre et Marie Curie,
Laboratoire d'Informatique de Paris 6 (LIP6),
4 Place Jussieu,
75005 Paris - France,
prénom.nom@lip6.fr

Résumé
Composer plusieurs infrastructures fournies en tant que service (*IaaS*) pour former une plate-forme virtualisée (*PaaS*) nécessite l'introduction d'un mécanisme de cache permettant d'offrir certaines garanties en termes de performances et de cohérence à l'utilisateur. Les caches répartis permettent d'offrir ce type de service et sont particulièrement bien adaptés aux environnements distribués, surtout lorsqu'ils offrent une faible latence réseau. Bien souvent, ces caches répartis sont associés à des systèmes de fichiers répartis ou sont implantés au niveau applicatif, nécessitant d'adapter les applications existantes, ce qui n'est pas compatible avec la philosophie du cloud. Cet article propose une approche générique pour développer un cache réparti en l'intégrant directement dans le noyau Linux, ce qui permet d'apporter le niveau de généricité dont les clouds ont besoin. Nous avons implémenté un prototype de cache réparti utilisant cette approche. Notre évalation des performances montre que nous pouvons multiplier par 6 le débit lors de lectures aléatoires.

Mots-clés : cache réparti, cloud, iaas, noyau

1. Introduction

Former une plate-forme virtualisée dans le cloud (*PaaS*) nécessite de composer différentes ressources distantes et hétérogènes fournies par des infrastructures en tant que service (*IaaS*). Cet assemblage doit permettre de conserver une certaine souplesse et rester indépendant des *IaaS* utilisés. L'utilisation de différents fournisseurs d'*IaaS* nécessite donc l'introduction d'une interface permettant d'offrir certaines garanties à l'utilisateur, comme les performances. La principale source de ralentissements est l'accès aux données stockées dans le cloud et distantes.
Une des approches système classiques pour lever cette contrainte repose sur l'utilisation de caches. Un cache est un espace de stockage utilisé pour conserver des données à forte localité spatiale ou temporelle. La mémoire utilisée pour le cache est plus rapide que l'espace de stockage principal, ce qui permet au processeur, rapide, de travailler avec des copies des données dans le cache plutôt que dans l'espace de stockage principal, plus lent.
Les caches répartis sont des caches particulièrement bien adaptés aux environnement répartis et ont été largement étudiés par le passé [8, 20, 21, 3, 4, 13]. Si des solutions de caches répartis ont été proposées, elles sont soit intégrées au sein même d'un système de fichiers réparti [1, 2, 10, 12, 17, 24, 28], soit d'un plus haut niveau [15] et nécessitent que les applications soient adaptées pour les utiliser.
Notre approche consiste à intégrer un système de cache réparti directement dans le noyau Linux, de façon suffisamment générique pour qu'il puisse être utilisé par les systèmes de fichiers existants avec un minimum de modifications. Notre objectif pour valider cette approche est de réaliser un prototype de système de cache réparti simple et permettant d'améliorer les performances lors de lectures aléatoires.
Ainsi, nous avons réalisé un cache réparti intégré au noyau Linux tout en limitant au minimum l'intrusion nécessaire dans celui-ci. Ce cache offre de bonnes performances :

– il permet de multiplier par 6 le débit obtenu lors de lectures aléatoires de fragments de petite taille (< 1Ko) ;
– il commence à dégrader les performances lorsque la taille des fragments devient grande (> 65Ko).

Le reste de cet article s'organise comme suit. La section 2 est une présentation des différents types de caches répartis et de leurs caractéristiques. La section 3 présente l'architecture du cache réparti que nous avons développée et intégrée dans le noyau Linux. Une évaluation de ce cache réparti est exposée dans la section 4. La section 5 montre comment notre contribution se positionne par rapport aux architectures de caches répartis existantes. Enfin, la section 6 conclut avec une synthèse de notre travail et ouvre sur nos futurs travaux.

2. Contexte des caches répartis

Les caches répartis permettent d'améliorer le temps de réponse des accès aux données en profitant d'une latence réseau plus faible que la latence des accès aux périphériques de stockage. Il peuvent s'implanter à plusieurs niveaux dans la *pile système* : les solutions comme *memcached* [15] se situent au niveau applicatif, dans l'espace utilisateur, et nécessitent une adaptation des applications clientes pour qu'elles utilisent le cache, mais peuvent être plus performantes parce que la gestion du cache peut être optimisée spécifiquement pour l'application [16].

Les solutions plus bas niveau ont l'avantage d'être plus modulaires et peuvent être utilisées avec les applications sans qu'il ne soit nécessaire de les adapter. Il est possible de proposer un cache réparti au niveau du système de fichiers, ce qui permet de profiter des fonctionnalités offertes par les couches inférieures du système (buffering, prefetching, ...). Cette solution ne nécessite pas la mise à jour des applications, mais oblige les utilisateurs à utiliser un système de fichiers en particulier, ce qui n'est pas vraiment adapté à la philosophie du cloud. Une autre solution consiste à se placer à un niveau plus bas, directement dans le *page cache* du noyau, ce qui permet de rendre compatible le cache réparti avec n'importe quel système de fichiers, et donc avec n'importe quelle application.

On distingue essentiellement deux types de caches répartis : les *caches distants* dans lesquels les nœuds profitent *exclusivement* d'un espace de mémoire (rapide) distant comme extension de leur *cache local*, et les *caches coopératifs* dans lesquels les nœuds mettent en commun de la mémoire pour former un *cache global* partagé et collaborent pour le maintenir.

Introduits par Comer et Griffionen [8], les caches distants sont des caches répartis dans lesquels on distingue les serveurs de cache, qui fournissent la mémoire utilisée comme support de stockage pour le cache, et les clients, qui utilisent ce cache et qui ne communiquent pas entre eux.

Les caches coopératifs sont une évolution des caches distants dans laquelle les clients peuvent intéragir entre eux pour accéder aux données du cache qui sont réparties dans les mémoires des clients. Leff et al. [20, 21] font la différence entre les architectures de caches répartis asymétriques et symétriques : les premières sont l'équivalent des caches distants, client/serveur, et dans les secondes tous les nœuds peuvent potentiellement répondre à un *miss* d'un autre nœud.

Parallèlement à cette approche, Blaze et Alonso [3] proposent d'utiliser le disque dur local des machines clientes pour étendre les capacités de leur cache et ainsi diminuer la charge du serveur. Les auteurs proposent ensuite que les clients s'échangent directement entre eux les fichiers depuis leurs caches [4].

Dahlin et al. [13] ont étendu les travaux de Blaze et Alonso en se concentrant sur le temps de réponse, et ont proposé plusieurs algorithmes de cache coopératif dans lesquels les données du cache sont réparties dans les mémoires des clients.

Pour étudier la faisabilité de notre approche intégrée au noyau Linux, nous avons choisi de développer un cache distant, ceux-ci sont moins complexes que les caches coopératifs mais les mécanismes d'intégration avec le noyau sont similaires.

3. Mise en œuvre d'un cache distant au sein du noyau Linux

La réalisation d'un cache réparti nécessite d'introduire dans le noyau des mécanismes complexes mais peu coûteux. L'une des propriétés des caches est la performance : un cache réparti dont la latence d'accès est plus importante que la latence d'accès au périphérique de stockage serait contre-productif. Il est donc nécessaire de minimiser le temps de calcul des composants que nous développons.

En outre, une implémentation classique d'un composant noyau nécessite d'allouer de la mémoire. Or, un cache réparti a justement pour rôle de réduire la pression sur la mémoire en permettant de déplacer les données du cache local vers le cache réparti.
Nous présentons l'architecture de notre cache distant et son intégration au noyau Linux dans la section 3.1, puis nous analysons les principales difficultées et leurs solutions dans la section 3.2.

3.1. Architecture du cache distant

La figure 1 présente l'architecture globale de notre cache distant. Il est constitué par 2 modules noyau : un module « client » et un module « serveur », et de modifications apportées au noyau Linux nécessaires pour faire fonctionner le client. Une couche réseau basée sur TCP est partagée entre ces deux modules pour gérer l'envoi et la réception de messages entre le client et le serveur. Notre cache est étroitement intégré au page cache du noyau Linux et à l'algorithme de récupération de mémoire (*PFRA*), l'unité de stockage est donc la page.

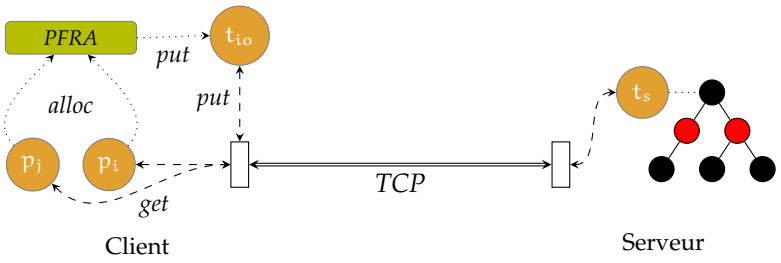


FIGURE 1 – Architecture globale du cache distant

3.1.1. Protocole de communication

Le client et le serveur communiquent entre eux par envoi de messages via une socket *TCP*. Il n'a pas été nécessaire de développer un protocole de communication complexe, en particulier parce que nous réalisons un cache distant qui ne nécessite pas d'intégrer des mécanismes complexes comme la recherche ou de placement de pages. Ainsi, ce protocole repose principalement sur 2 actions simples, *put* et *get* :

L'action *put* est effectée par le client pour placer une page dans le cache distant. Lorsque le client veut envoyer une page au serveur, il envoie un message *put* au serveur qui contient la page et des informations permettant de l'identifier de façon unique, puis attend un acquittement du serveur pour l'informer que la page a été correctement reçue.

L'action *get* est effectuée par le client pour demander une page au serveur dans le cache distant. Lorsque le client veut récupérer une page du cache distant, il envoie un message *get* au serveur avec l'identifiant unique de la page. Le serveur répond par un message *get_response*, si la page a été trouvée ce message contient la page, sinon un drapeau indique que la page n'est pas dans le cache distant.

Ce protocole repose sur l'existence d'un identifiant unique. Lors des échanges de pages entre le serveur et le client, chaque page est identifiée par un triplet ($\texttt{pool}_{\texttt{id}}$, `inode`, `index`) :
– le $\texttt{pool}_{\texttt{id}}$ identifie le support de stockage sur lequel est stockée la page, cet entier est généré par le client pour chaque support de stockage qui utilise le cache distant ;
– l'*inode* identifie l'inode qui contient la page ;
– l'*index* identifie le numéro de la page dans l'espace d'adressage du fichier, en commençant à partir de 0.

3.1.2. Architecture du serveur

Le serveur de cache est architecturé autour de 2 composants :
– un *thread* t_s en charge de la réception et du traitement des messages du client ;
– une zone de stockage pour y placer les pages envoyées par le client.

Le noyau Linux fourni une API d'arbre rouge-noir sur laquelle le serveur repose pour stocker les pages. Les arbres rouge-noir nous permettent d'effectuer les opérations d'insertion, de suppression et de recherche en $O(\log(n))$, nous fournissant des garanties sur les latences supplémentaires introduites par la structure de données.
Les pages de l'arbre rouge-noir sont également liées entre elles dans une liste *LRU* : lorsqu'un nombre de pages N est atteint, les pages les plus anciennement utilisées sont supprimées lors de l'ajout d'une nouvelle page. Lorsqu'un client demande une page au serveur, elle est déplacée en tête de liste.

3.1.3. Architecture du client

L'architecture du client est beaucoup plus complexe que celle du serveur. Le client ne doit pas introduire une latence trop importante pour que le cache distant conserve un intérêt. De plus, il doit minimiser son empreinte mémoire et traiter les problématiques qui sont impliquées par l'algorithme de récupération de mémoire (*PFRA*). Enfin, il doit gérer l'interaction avec les processus de l'espace utilisateur qui agissent de façon concurrente.

Intégration au sein du noyau Linux

Pour minimiser l'intrusion de notre cache distant dans le noyau Linux, le client interagit avec lui par l'intermédiaire de l'API *cleancache* [22]. L'objectif de cette API est de proposer au noyau Linux un second niveau de mémoire, orienté « objet », de taille inconnue et qui n'est pas directement adressable. *Cleancache* fourni une API permettant d'enregistrer des fonctions qui seront exécutées lorsqu'une page du cache local est retirée (*put*) et manquante (*get*). Le module *zcache* [9] est un exemple d'utilisation de cette API : l'opération *put* compresse les pages mémoire pour libérer de la place, et l'opération *get* les décompresse.
Cependant, il a été nécessaire d'apporter des modifications au noyau : *cleancache* impose que l'opération *put* soit non-bloquante, or dans le cas d'un cache réparti celle-ci doit effectuer des opérations d'entrée/sortie sur le réseau qui sont bloquantes.

Récupération des pages du cache distant (*get*)

Lors d'un *miss* sur le cache local, le client doit vérifier auprès du serveur de cache s'il ne possède pas la page demandée : il appelle alors la fonction *get* de l'API *cleancache* avant d'initier une entrée/sortie sur le périphérique de stockage
Cette opération est généralement bloquante, il est souvent nécessaire d'attendre la réponse du serveur avant de retourner la page demandée. Bien que moins coûteuse qu'une lecture sur un disque dur classique, elle reste très pénalisante en cas de *miss* sur le cache distant. De plus, les temps d'accès aux périphériques bloc sont généralement amortis lors de lectures séquentielles, or l'utilisation de l'API *cleancache* ne nous permet pas de savoir qu'un processus utilisateur a demandé à lire plusieurs pages consécutives : il est donc nécessaire d'introduire des mécanismes permettant de limiter le nombre d'accès au cache distant. Ces mécanismes sont présentés dans la section 3.2.

Envoi de pages dans le cache distant

Le *PFRA* a pour objectif de libérer des pages de la mémoire lorsqu'une allocation échoue. C'est cet algorithme qui gère l'éviction des pages du cache local, et donc c'est par son intermédiaire que les opérations *put* sont effectuées. L'interaction de notre cache distant avec le *PFRA* pose un problème majeur : la fonction *put* de l'API *cleancache* est appelée dans une section critique protégée par un *spinlock*, or il n'est pas possible d'effectuer des opérations bloquantes dans ce type de section critique.
Pour contourner ce problème, nous sommes obligés de traiter l'envoi des messages *put* de façon asynchrone, c'est pourquoi notre client crée un *thread* t_{io} dédié à l'envoi de ces messages. Notre implémentation de la fonction *put* de l'API *cleancache* ne fait que deux opérations :

1. Placer la page dans une file d'attente q
2. Réveiller le thread t_{io}

La fille d'attente est ensuite vidée lorsque le thread t_{io} est activé par l'ordonnanceur. De cette façon, le *PFRA* n'est pas bloqué par le *put*, mais il s'attend tout de même à ce que la page soit libérée de la mémoire, il est donc nécessaire de synchroniser le *PFRA* et notre *thread* d'envoi de messages *put*

lorsqu'un certain nombre de pages a été dépassé, de sorte que le *PFRA* libère les pages de la mémoire lorsque les acquittements sont reçus.

3.2. Détails d'implémentation

La réalisation du cache distant a soulevé plusieurs problématiques importantes pour les performances. Cette section propose de détailler ces difficultés et les solutions que nous y avons apportées.

3.2.1. Gestion des données stockées dans le cache

Le premier mécanisme consiste à éviter d'envoyer un message *get* au serveur lorsqu'on sait que le serveur ne possède pas la page dans son cache. Ainsi, nous avons introduit un *bitmap* similaire aux filtres de Bloom [5], avec une seule fonction de hachage. Cette structure permet de tester si un élément appartient à un ensemble tout en garantissant qu'il n'y a pas de faux positif. Une fonction de hachage permet d'associer un identifiant de page à un bit du bitmap. Initialement, tous les bits du bitmap sont à 0. La figure 2 illustre le fonctionnement d'un filtre de Bloom n'utilisant qu'une seule fonction de hachage. Lors d'une opération *put* sur une page, le bit correspondant est positionné à 1 pour indiquer que la page a été placée dans le cache distant. Lors d'un *get*, le bit correspondant à l'identifiant de la page est testé, s'il est à 0 c'est que la page n'est pas dans le cache distant, et on évite l'envoi d'un message au serveur.

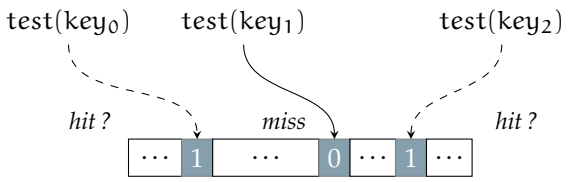


FIGURE 2 – Filtre de Bloom : le bit correspondant au hash de key_1 n'est pas positionné : il est garanti que la page key_1 n'est pas dans le cache. En revanche, les bits des clés key_0 et key_2 sont positionnés, il est possible qu'il sagisse d'un faux positif.

Lorsque le bit correspondant est à 1 il n'est pas garanti que le serveur possède la page : il possible d'avoir des faux positifs. En supposant que la sortie fonction de hachage utilisée est uniforme, à 2^{20} pages lues, équivalent par exemple à un fichier de 4Go, le taux de faux positif est d'environ 10% lorsque le bitmap contient 2^{23} bits [14].
En pratique, l'identifiant des pages n'est pas parfaitement distribué puisqu'il dépend essentiellement de 2 caractéristiques : le nombre de fichiers différents (*inode*) et la quantité de données différentes lues dans chaque fichier (*index*). Les performances gagnées grâce au filtre de Bloom sont donc fortement dépendantes du type de charge.
L'efficacité des filtres de Bloom pour limiter les *miss* distants a déjà été montrée par le passé [25, 14]. Dans cet article, nous nous intéressons plus à la faisabilité d'une approche noyau pour adapter les caches répartis au cloud computing. Ainsi, notre évaluation présentée dans la section 4 ne tient pas compte du coût de la maintenance du filtre de Bloom, ni du choix de la fonction de hachage, fortement dépendante du type de charge : nous considérons que le filtre ne donne jamais de faux positifs.

3.2.2. Gestion des accès au cache

Dans le cas de lectures séquentielles, l'API *cleancache* nous empêche de savoir que les algorithmes de plus haut niveau, présents dans les systèmes de fichiers et dans le *VFS*, demandent à accéder à des pages consécutives au sein d'un même fichier. L'algorithme de préchargement du noyau linux peut donc se transformer en goulot d'étranglement puisque chaque page réclamée par cet algorithme va impliquer un RTT entre le client et le serveur, alors qu'elles sont consécutives. Nous devons donc implémenter un algorithme de préchargement similaire pour l'opération *get*. La figure 3 illustre le fonctionnement de cet algorithme. Un ensemble de variables représentent l'état du préchargement :
– $ahead_{page}$ contient les pages qui ont été lues en avance

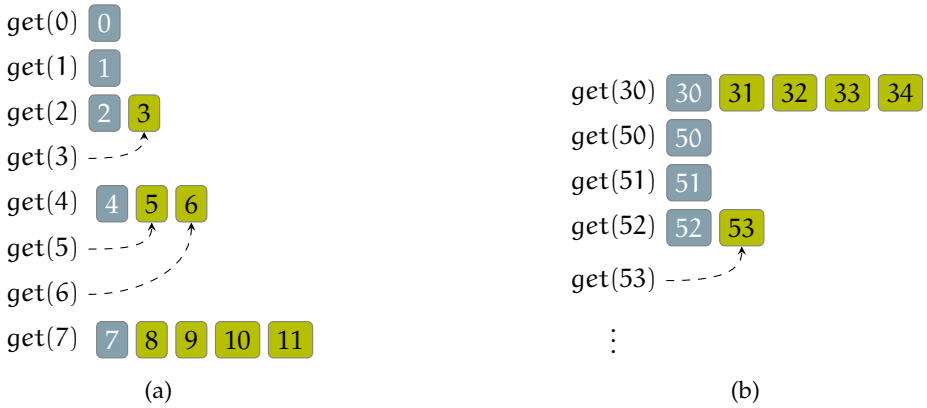


FIGURE 3 – Préchargement de pages depuis le cache distant : lorsqu'une séquence est détectée (a), l'algorithme double le nombre de pages à préchargement à chaque *miss*. Lors d'un saut (b), il le détecte et conserve la même fenêtre de préchargement pour tenter d'optimiser les accès aléatoires à des suites de pages. Lors d'un second saut, la taille de la fenêtre est réinitialisée.

- $ahead_{size}$ contient le nombre de pages que peut contenir la fenêtre de préchargement : cette valeur correspond au nombre de pages présentes dans $ahead_{page}$ ou au nombre de pages à précharger lorsque $Card(ahead_{page}) = 0$
- $last$ contient l'identifiant de la dernière page lue et permet de détecter une lecture séquentielle
- $consecutive$ contient le nombre de pages consécutives lues

Lorsque l'algorithme détecte une séquence, il recherche dans $ahead_{page}$ si la page n'a pas été préchargée. Si c'est le cas, elle peut être retournée sans qu'il ne soit nécessaire de contacter le serveur. Sinon, les variables $last$ et $consecutive$ sont mises à jour et si la séquence détectée a dépassé 2 pages, on met à jour $ahead_{size}$ pour qu'il soit préchargé au moins $(consecutive - 2)$ pages, au plus 64 pages en dehors de la page correspondant à id.

Lorsque la page demandée n'est pas dans une séquence, l'état du préchargement est réinitialisé. S'il y a eu une séquence détectée précédemment, $ahead_{size}$ est réinitialisée pour précharger au moins $consecutive$ pages et au plus 64 pages. L'idée est que le mode d'accès au cache peut correspondre à des lectures de séquences aléatoires de même taille. Si ce n'est pas le cas, l'état du préchargement sera réinitialisé une seconde fois et positionnera $consecutive = 0$, ce qui aura pour effet de désactiver le préchargement.

3.2.3. Gestion des communications

Les communications entre le client et le serveur sont le goulet d'étranglement du cache distant : d'une part parce qu'elles nécessitent de la mémoire alors même qu'elles sont utilisées pour en libérer, d'autre part parce que la majorité du temps des communications sont les latences réseau.

Empreinte mémoire des buffers réseau

La pile réseau utilise de la mémoire pour fonctionner : des buffers sont alloués pour contenir les données à envoyer. Lorsque le *PFRA* s'active, les pages de données doivent être envoyées par le réseau au serveur de cache et donc de la mémoire est réclamée par la pile réseau : il peut donc y avoir un interblocage.

Pour limiter cet effet de bord, nous transférons les messages à la pile réseau en *zero-copy*. Cependant, des descripteurs doivent quand même être alloués pour décrire les segments qui seront envoyés. Nous avons donc forcé la pile réseau à effectuer des allocations de mémoire atomiques : lorsque la mémoire subit une forte pression, les allocations de la pile réseau échouent et renvoient une erreur que nous pouvons traiter.

Impact de la latence réseau

Chaque page échangée avec le cache distant nécessite un échange sur le réseau : même en cas de faible latence le coût lors de lectures séquentielles n'est plus négligeable. Pour amortir au maximum cette latence, les échanges entre le client et le serveur se font via un *pipeline*.
Côté serveur, le *thread* chargé de la réception et du traitement des messages réceptionne un maximum de messages en mode « non-bloquant » avant de les traiter les uns à la suite des autres, ce qui permet de maximiser l'utilisation des caches du processeur et des buffers réseau, ainsi que de réduire la temporisation introduite par TCP. En particulier, les acquittements aux messages *put* sont de petite taille et cette optimisation permet à TCP de réduire le nombre de segments qui seront nécessaires.
Côté client, le *thread* en charge des *put* envoie un maximum de messages *put* avant d'attendre tous les acquittements du serveur. Cette optimisation augmente d'autant plus les performances qu'elle permet de limiter le nombre de fois que le *PFRA* et ce thread devront se synchroniser, et donc de réduire le nombre de changements de contexte.

4. Évaluation

Nous avons mesuré les performances de notre cache distant à l'aide d'un micro-benchmark de lectures aléatoires que nous avons réalisé. La section 4.1 présente la plateforme expérimentale que nous avons mis au point pour ces expériences. Notre première expérience, présentée dans la section 4.2 consiste à évaluer le degré d'intrusion du cache dans le noyau Linux. La section 4.3 présente les résultats d'une expérience consistant à mesurer les performances maximales que nous pouvons obtenir de notre cache. La section 4.4 présente les performances maximales lorsque le *PFRA* a besoin de faire des *put*, puis la section 4.5 tente d'expliquer en quoi les *put* ralentissent les performances de notre cache.

4.1. Plateforme d'expérimentation

Notre plateforme d'expérimentations est composée d'un ensemble de machines virtuelles et d'un micro-benchmark permettant de mesurer le débit de lecture suivant différentes configurations.

Plateforme expérimentale

Les expérimentations ont été menées à l'aide de 3 machines virtuelles s'éxécutant sur une machine hôte équipée d'un Core i7-2600 (4 cœurs avec hyperthreading) et de 8Go de mémoire. Nous disposons également d'un disque dur de 1To ayant 32Mo de cache et une vitesse de rotation de 7200 tours/minutes, soit une latence moyenne de 4.16ms. Les 3 machines virtuelles sont organisées comme suit :

- Le serveur *iSCSI* permet de simuler une infrastructure dans laquelle les périphériques blocs sont placés dans un *SAN*, et nous permet de mesurer l'impact de notre cache lorsque le périphérique bloc est accédé par le réseau. Nous avons attribué à cette machine virtuelle 2 processeurs virtuels et 512Mo de mémoire.
- Le serveur de cache contient le cache distant. Nous lui avons attribué 2 processeurs virtuels et 4Go de mémoire, ce qui nous permet d'éviter les effets de bords liés à l'éviction de pages de la liste LRU du serveur pour nos mesures de performances.
- Le client exécute un micro-benchmark de lecture de fichiers aléatoires sur son disque local virtuel et/ou sur le périphérique exporté par le serveur *iSCSI*. Nous avons implémenté dans le client du cache distant une interface au *sysfs* du noyau Linux qui nous permet de récupérer des statistiques sur l'utilisation du cache distant, comme le nombre de *miss* ou la durée moyenne d'un *put*. Le client dispose de 2 CPU virtuels et de 512Mo de mémoire.

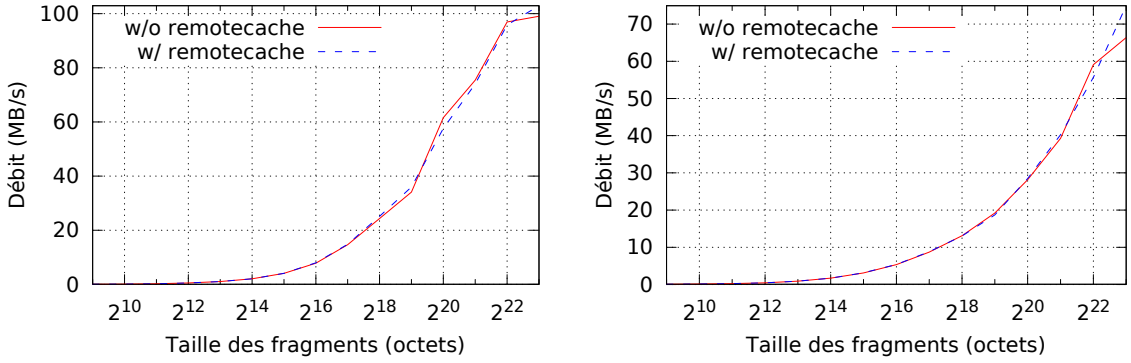
Les 3 machines virtuelles sont reliées entre elles par une connexion ethernet 1Gbit avec une latence (RTT) d'environ 600µs. Le système d'exploitation utilisé est Archlinux 64-bit avec un noyau Linux 3.4.0 contenant notre cache distant. L'hyperviseur utilisé est qemu-kvm. Pour éviter les effets de bord liés au page cache de la machine hôte, les disques durs virtuels utilisés pour les mesures de performances sont manipulés par les machines virtuelles en mode synchrone.

Métriques

Les applications réelles accèdent aux périphériques de stockage de façon plus ou moins aléatoire et/ou plus ou moins séquentielle. Les lectures aléatoires ne permettent pas d'atteindre des débits importants à cause de la latence d'accès au périphérique qui est trop élevée. À l'inverse, les lectures séquentielles sont rapides puisque le périphérique est accédé moins souvent.
Le micro-benchmark que nous avons développé permet de mesurer le temps nécessaire pour lire N octets dans un fichier. La lecture est *fragmentée* en fragments de taille F, chacun étant lu à une position choisie aléatoirement dans le fichier. En faisant varier F d'une expérimentation à une autre nous pouvons représenter différents types de *workload* : très aléatoires à séquentiels. Le temps d'exécution moyen mesuré est ensuite ramené en débit moyen obtenu (Mo/s). Les expérimentations ont été menées sur des fichiers de 2Go stockés sur le disque local virtuel du client et sur le périphérique bloc exporté par le serveur *iSCSI*. Pour nos expériences, nous avons choisi N = 32Mo et F variant exponentiellement de 512 octets à 8Mo.

4.2. Surcoût de l'exécution du code

Pour mesurer le surcoût de l'exécution de notre code, nous avons évalué les performances de notre cache distant lorsqu'il est activé mais vide. La figure 4 présente les résultats de cette expérience : le surcout de l'exécution du code client est négligeable puisque le filtre de Bloom nous permet d'éviter des accès au cache alors que la donnée n'y est pas.



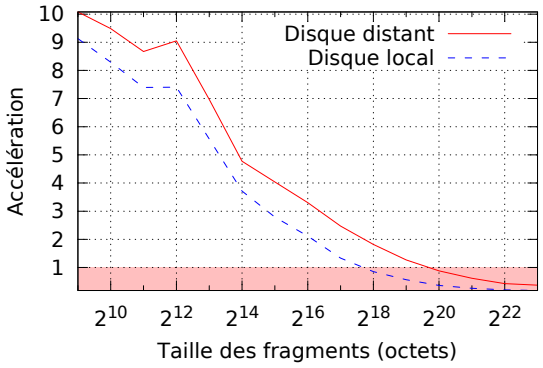
(a) Disque local

(b) Disque distant

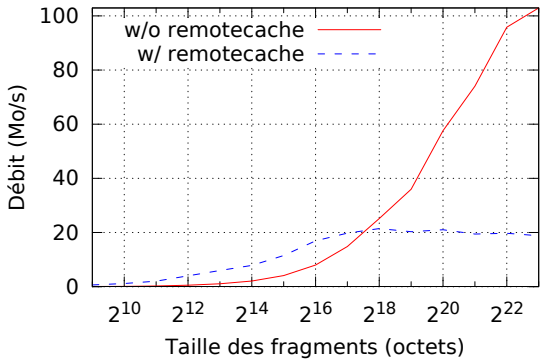
FIGURE 4 – Surcoût de l'exécution du code : lorsque le cache est activé, il n'y a pas de *put* et les *get* finissent toujours par un *miss*, mais les accès réseau sont évités gâce au filtre de Bloom.

4.3. Performances maximales

Les performances maximales du cache distant correspondent aux performances obtenues lorsque les données demandées sont systématiquement dans le cache et que le cache local du client ne contient aucune donnée qui peut être *put* sur le serveur. Les résultats de cette expérience sont présentés dans la figure 5. La figure 5a présente l'accélération obtenue avec notre cache sur le débit. Notre cache permet de multiplier par 10 le débit lorsque la taille des fragments est faible (< 1Ko), mais les performances diminuent lorsque les lectures deviennent plus séquentielles. L'efficacité de notre algorithme de préchargement est démontrée par le palier entre 2^{11} et 2^{12} : c'est à ce moment là que notre algorithme de préchargement commence à s'activer, et permet donc de limiter la diminution des performances. La zone rouge indique le moment à partir duquel notre cache distant dégrade les performances (accélération < 1).
La figure 5b présente le débit obtenu lors de cette expérimentation sur un disque local : le moment où notre cache dégrade les performances correspond au moment où le débit obtenu atteint les limites des performances de notre couche réseau (environ 20Mo/s).



(a) Accélération (disque local et distant)

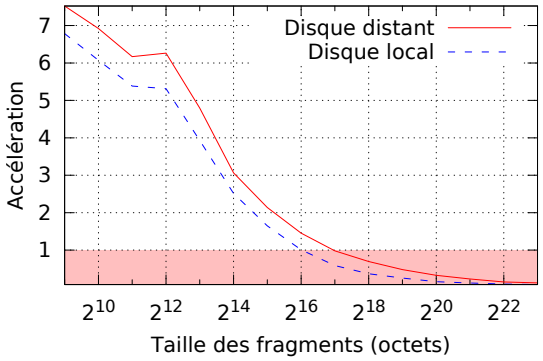


(b) Débit avec et sans cache distant (disque local)

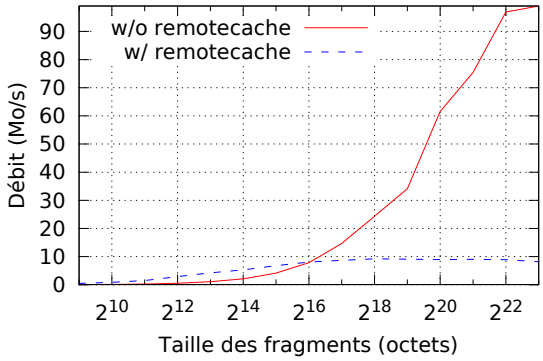
FIGURE 5 – Peformances maximales : il n'y a jamais de *put* et les pages sont systématiquement présentes dans le cache distant.

4.4. Performances maximales avec des *put*

L'opération *put* est nécessaire au fonctionnement du cache puisque c'est elle qui place les pages dans le cache distant. Nous avons mesuré les performances obtenues lorsque chaque opération *get* nécessite de libérer de la mémoire, et donc des opérations *put*. Les résultats de cette expérience sont présentés dans la figure 6. Les *put* ont un coût important puisque l'accélération maximale obtenue (figure 6a) est



(a) Accélération (disque local et distant)



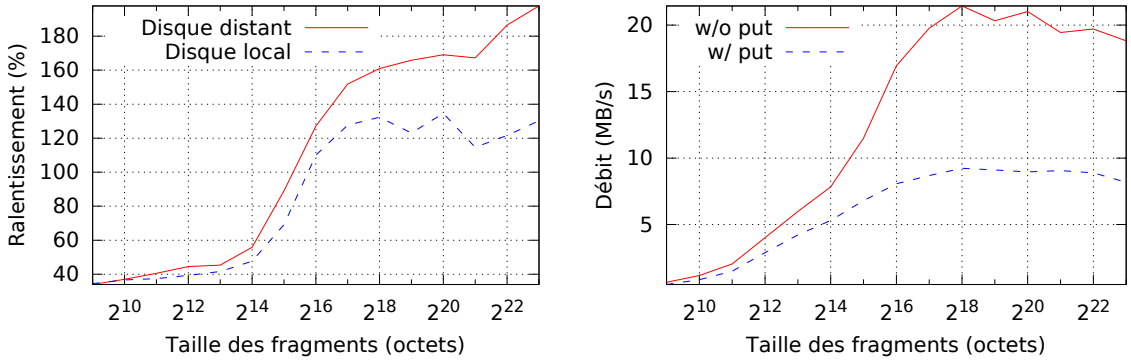
(b) Débit avec et sans cache distant (disque local)

FIGURE 6 – Peformances maximales avec des *put* : la mémoire locale est saturée en pages « distantes », pour chaque *get* il est nécessaire de faire un *put* pour libérer de la mémoire.

beaucoup plus faible. De plus, le moment où le cache distant dégrade les performances arrive plus tôt : au delà de 2^{16}, contre 2^{18} lorsqu'il n'y a pas de *put*. La figure 6b présente le débit obtenu lors d'une lecture sur un disque local : les allocations de mémoire nécessaires aux opérations *get* impliquent des opérations *put*, ce qui divise par 2 les performances par rapport au débit présenté dans la figure 5b.

4.5. Impact des *put* sur les performances

La figure 7 présente une comparaison des résultats présentés dans les sections 4.3 et 4.4. La figure 7a présente le ralentissement observé lors des mesures de débit D_1 sans *put* par rapport au débit D_2 mesuré avec des *put* : $(D_2 - D_1)/D_1$. Lorsque la taille des fragments est faible ($< 2^{14}$), les lectures sont nombreuses et l'essentiel du coût est la latence : le temps nécessaire à la récupération de la mémoire peut



(a) Ralentissement causé par les *put* sur les accès au cache

(b) Débit avec et sans *put* (disque local)

FIGURE 7 – Impact des *put* sur les performances : les échanges client/serveur et les synchronisations avec le *PFRA* limitent la bande passante utilisable.

donc être masqué par celui-ci. Lorsque la taille des fragments augmente, les lectures sont beaucoup plus rapides et le *PFRA* subit une forte pression, les *put* ne peuvent plus être masqués par la latence réseau et finissent par ralentir le cache.
La figure 7b présente le débit obtenu lors de la lecture d'un fichier local depuis le cache distant, avec et sans *put*. Le débit est divisé par 2 en présence de *put*, ce qui est lié à l'ordre dans lequel sont effectuées les opérations : le système de fichier alloue d'abord de la mémoire (*put*) avant d'effectuer un *get* ou une lecture depuis le périphérique bloc. Comme ces opérations ne peuvent pas s'effectuer en parallèle, les latences de celles-ci s'accumulent.

5. État de l'art

Les caches répartis ont été largement étudiés ces 2 dernières décennies. Des architectures de caches répartis répondant à des problèmes spécialisés ont été proposées mais ne sont pas appropriées pour les environnements hétérogènes tels que ceux que nous visons.

5.1. Caches répartis applicatifs

Les caches répartis applicatifs sont implantés dans l'espace utilisateur, sous forme de bibliothèques ou de services, et nécessitent que les applications qui souhaitent les utiliser soit adaptées. Memcached [15] est une solution de cache distant largement utilisée par de très grands systèmes comme Youtube ou Facebook. Même si memcached est utilisé pour des *PaaS* comme les Google App Engine ou Amazon Web Services, c'est une solution applicative et nécessite donc que les applications utilisatrices soient modifiées en conséquence, ce qui n'est pas adapté à la philosophie d'un cloud dans lequel les *PaaS* sont composés d'*IaaS* hétérogènes.

5.2. Caches répartis associés à un système de fichiers

Beaucoup de cache répartis ont été développés au sein d'un système de fichiers pour répondre un à besoin particulier. Ces caches répartis ne peuvent pas être utilisés dans un environnement aussi hétérogène que celui que nous visons puisqu'il serait nécessaire d'utiliser le même système de fichiers partout, ce qui réduit le niveau de flexibilité offert par le cloud.
xFS [11] est l'un des premiers systèmes de fichiers à proposer une solution de cache réparti. Il permet les écritures différées dans le cache réparti : les données sont d'abord placées dans le cache local avant d'être transférées dans le cache réparti. Les accès concurrents en écriture sont gérés par un serveur qui attribue des jetons permettant aux clients de s'assurer qu'ils ont l'exclusivité des accès aux données.
Les auteurs de PAFS [10] ont proposé une solution originale pour contourner les problèmes de concurrence en écriture : les données passent directement du cache réparti vers l'application, sans passer par le

cache local, ce qui permet de garantir le niveau de cohérence exigé par le standard *POSIX* sans nécessiter de mécanisme de synchronisation complexe.
Des propositions de caches coopératifs pour NFS ont également été faites. NFS-cc [28] est une version de NFS intégrant un cache coopératif. Dans NFS-cc, seules les lectures sont effectuées depuis le cache coopératif, les écritures sont directement envoyées au serveur. NFS-CD [2] est une autre solution de cache coopératif pour NFS qui permet cette fois les écritures dans le cache réparti.

5.3. Caches répartis « bas niveau »

Les caches répartis de plus bas niveau sont généralement implantés au sein d'une couche plus basse que le *Virtual File System*, ce qui permet d'atteindre le niveau de flexibilité que l'ont peut exiger du cloud.
Xhive [19] est une solution de cache coopératif *copy-on-write* spécialisée dans la coopération entre les machines invitées d'un même hôte. Dans Xhive, l'hyperviseur joue le rôle de coordinateur pour permettre aux invités de localiser les pages mémoire ou de participer au cache. La solution que nous visons n'est pas spécifique aux hyperviseurs et doit fonctionner entre plusieurs machines physiques.
CaaS [16] est une architecture de cache élastique développée pour proposer du cache à la demande à des serveur de calcul dans le cadre du cloud. Dans cette architecture, des serveurs de mémoire mettent à disposition des serveurs de calcul des fragments de mémoire qui sont accédés via le protocole *RDMA*. Les auteurs utilisent *dm-cache* [26], une cible pour former un périphérique bloc virtuel intégrant le périphérique bloc représentant le cache (chunks de mémoire accédés en *RDMA*) et le périphérique bloc local. Leur solution se rapproche de ce que nous souhaitons proposer. Cependant, les systèmes de fichiers répartis comme NFS ou Ceph [27] ne sont pas toujours représentés par un périphérique bloc, condition nécessaire à l'utilisation du framework de virtualisation de périphériques blocs du noyau Linux.

6. Conclusion et perspectives

Nous avons implémenté un cache distant générique dans le noyau Linux. Notre approche est peu intrusive et opérationnelle. L'évaluation de ce cache a montré qu'il permet d'atteindre de bonnes performances, en particulier lors de lectures aléatoires : le débit obtenu est multiplié par 6 lors de lectures aléatoires de fragments de 1Ko. Cependant, les performances se dégradent lors de lectures séquentielles, notamment lorsque la taille des fragments dépasse 65Ko.
Les mesures de performances que nous avons réalisé à l'aide d'un micro-benchmark permettent d'analyser le comportement de notre cache suivant différents types de *workloads*. À court terme, nous souhaitons reproduire ces expériences sur la plateforme Grid'5000 [6], en particulier en utilisant des connexions Infiniband 20G pour analyser le comportant de notre cache sur un réseau haute performance. De plus, nous allons mener de nouvelles évaluations avec d'autres benchmarks comme Bonnie++ [7], Postmark [18] ou Filebench [23] pour mieux caractériser les performances de notre cache distant.
Notre cache dégrade les performances lors de longues lectures séquentielles. Ce problème est étroitement lié à l'algorithme de récupération de mémoire qui doit envoyer des pages sur le réseau et attendre les acquittements, et donc ralenti le système. Nous allons travailler à une meilleure intégration avec cet algorithme pour limiter cet effet de bord.
Enfin, ces travaux seront étendus aux caches coopératifs. Notre objectif est de réaliser un cache coopératif adapté au cloud computing : il devra permettre d'offrir différents niveaux de service (performance, temps réel, cohérence,...) adaptés aux applications.

Bibliographie

1. Annapureddy (S.), Freedman (M. J.) et Mazières (D.). – Shark: scaling file servers via cooperative caching. *In : Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. p. 129–142. – Berkeley, CA, USA, 2005.
2. Batsakis (A.) et Burns (R.). – NFS-CD: write-enabled cooperative caching in NFS. *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, n3, mars 2008, pp. 323 –333.
3. Blaze (M.) et Alonso (R.). – Long-term caching strategies for very large distributed file systems. *In : Proceedings of the USENIX Summer 1991 Technical Conference*, pp. 3–16.

4. Blaze (M.) et Alonso (R.). – Dynamic hierarchical caching in large-scale distributed file systems. *In : Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 521 –528.
5. Bloom (B. H.). – Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, vol. 13, n7, 1970, p. 422–426.
6. Cappello (F.), Desprez (F.), Dayde (M.), Jeannot (E.), Jegou (Y.), Lanteri (S.), Melab (N.), Namyst (R.), Primet (P. V. B.), Richard (O.) et al. – Grid5000: a nation wide experimental grid testbed. *International Journal on High Performance Computing Applications*, vol. 20, n4, 2006, p. 481–494.
7. Coker (R.). – Bonnie++. – http://www.coker.com.au/bonnie++, 2001.
8. Comer (D.) et Griffioen (J.). – A new design for distributed systems: The remote memory model. *In : Proceedings of the USENIX Summer 1990 Technical Conference*, p. 127–136.
9. Corbet (J.). – zcache: a compressed page cache. – http://lwn.net/Articles/397574, juillet 2010.
10. Cortes (T.), Girona (S.) et Labarta (J.). – Design issues of a cooperative cache with no coherence problems. *In : Proceedings of the fifth workshop on I/O in parallel and distributed systems*. p. 37–46. – New York, NY, USA, 1997.
11. Dahlin (M. D.). – *Serverless network file systems*. – Thèse de PhD, UNIVERSITY of CALIFORNIA, 1995.
12. Dahlin (M. D.), Wang (R. Y.), Anderson (T. E.), Neefe (J. M.), Patterson (D. A.) et Roselli (D. S.). – Serverless network file systems. *ACM Transactions on Computer Systems (TOCS)*, vol. 14, n1, 1996, p. 41–79.
13. Dahlin (M. D.), Wang (R. Y.), Anderson (T. E.) et Patterson (D. A.). – Cooperative caching: using remote client memory to improve file system performance. *In : Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. – Berkeley, CA, USA, 1994.
14. Fan (L.), Cao (P.), Almeida (J.) et Broder (A. Z.). – Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, vol. 8, n3, 2000, p. 281–293.
15. Fitzpatrick (B.). – Distributed caching with memcached. *Linux journal*, vol. 2004, n124, 2004, p. 5–.
16. Han (H.), Lee (Y.), Shin (W.), Jung (H.), Yeom (H.) et Zomaya (A.). – Cashing in on the cache in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, n99, 2011, p. 1.
17. Howard (J. H.). – An overview of the andrew file system. *In : Proceedings of the USENIX Winter Technical Conference*, p. 23–26.
18. Katcher (J.). – *Postmark: A new file system benchmark*. – Rapport technique TR3022, Network Appliance, 1997.
19. Kim (H.), Jo (H.) et Lee (J.). – XHive: efficient cooperative caching for virtual machines. *IEEE Transactions on Computers*, vol. 60, n1, janvier 2011, pp. 106 –119.
20. Leff (A.), Pu (C.) et Korz (F.). – *A Comparison of Cache Performance in Server-Based and Symmetric Database Architectures*. – Rapport technique CUCS-016-90, Columbia University, 1990.
21. Leff (A.), Yu (P. S.) et Wolf (J. L.). – Policies for efficient memory utilization in a remote caching architecture. *In : Proceedings of the First International Conference on Parallel and Distributed Information Systems*, p. 198–207.
22. Magenheimer (D.), Mason (C.), McCracken (D.) et Hackel (K.). – Transcendent memory and linux. *In : Proceedings of the Linux Symposium*, p. 191–200. – Montréal, Québec, Canada, 2009.
23. McDougall (R.), Crase (J.) et Debnath (S.). – *Filebench: File system microbenchmarks*. – 2006.
24. Ousterhout (J. K.), Cherenson (A. R.), Douglis (F.), Nelson (M. N.) et Welch (B. B.). – The sprite network operating system. *IEEE Computer*, vol. 21, n2, 1988, p. 23–36.
25. Rousskov (A.) et Wessels (D.). – Cache digests. *Computer Networks and ISDN Systems*, vol. 30, n22, 1998, p. 2155–2168.
26. Van Hensbergen (E.) et Zhao (M.). – *Dynamic policy disk caching for storage networking*. – Rapport technique RC24123, IBM, 2006.
27. Weil (S. A.), Brandt (S. A.), Miller (E. L.), Long (D. D. E.) et Maltzahn (C.). – Ceph: a scalable, high-performance distributed file system. *In : Proceedings of the 7th symposium on Operating systems design and implementation*. p. 307–320. – Berkeley, CA, USA, 2006.
28. Xu (Y.) et Fleisch (B. D.). – NFS-cc: tuning NFS for concurrent read sharing. *International Journal of High Performance Computing and Networking*, vol. 1, n4, janvier 2004, pp. 203–213.