Commutation de paquets logicielle sur routeurs PC multi-coeurs

Norbert Egi, Adam Greenhalgh, Marc Handley, Mickael Hoerdt, Felipe Huici, Laurent Mathy, Panagiotis Papadimitriou

Commutation de paquets logicielle sur routeurs PC multi-coeurs

Norbert Egi* — **Adam Greenhalgh**** — **Mark Handley**** — **Mickaël Hoerdt*** — **Felipe Huici***** — **Laurent Mathy*** — **Panagiotis Papadimitriou***

* *Computing Dept., Lancaster University, UK*
{n.egi,m.hoerdt,l.mathy,p.papadimitriou}@lancaster.ac.uk
** *Dept. of Computer Science, University College London, UK*
a.greenhalgh@cs.ucl.ac.uk
*** *NEC Europe, Heidelberg, Germany*
felipe.huici@nw.neclab.eu

RÉSUMÉ. *Les processeurs multi-coeurs sur PC, en même temps que les progrès récents sur la performance de la mémoire et des bus, suggèrent un candidat idéal pour la construction d'architectures de routeurs purement logicielles qui soient flexibles et en même temps performantes. Dans cet article, nous présentons une architecture de commutation de paquets pour les routeurs logiciels ou virtuels dont l'objectif principal est de prévenir des accès mémoire coûteux tout en utilisant les ressources CPU disponibles au mieux pour traiter et commuter les paquets. Nous présentons premièrement le principe d'arbre de commutation dont la fonction est de garder les paquets commutés dans la même hiérarchie de cache du processeur. Nous identifions ensuite la source de contention principale des performances de commutation comme l'accès concurrent aux interfaces de sorties puis analysons les gains potentiels de performance en utilisant un verrou sur les interfaces de sortie ainsi qu'un mécanisme de retour sur l'état des queues du routeur. Finalement, nous proposons et évaluons une modification des arbres de commutation qui permet de contourner la contention due au partage des interfaces de sortie par les arbres.*

ABSTRACT. *Multi-core CPUs, along with recent advances in memory and buses, render commodity hardware a strong candidate for building flexible and high-performance software routers. We present a packet-forwarding architecture for software or virtual routers, whose primary objective is to prevent costly memory accesses while utilizing the available CPU resources for packet processing and forwarding. Departing from the forwarding-tree principle which keeps packets on the same cache hierarchy, we identify contention at the output interfaces as the primary performance limitation factor. Subsequently, we investigate potential performance gains by using a locking mechanism and feedback from queues. Finally, we propose and evaluate a modification to the forwarding trees which alleviates contention as output interfaces are not directly shared by trees.*

MOTS-CLÉS : *Routeurs, Multi-coeurs,PC, Performances*

KEY WORDS: *Routers, Multi-core,PC, Performances*

1. Introduction

Les recherches récentes ont montré que les PC modernes sont une plateforme viable pour l'implémentation de routeurs haute performance [EGI 08b, ARG 08]. Ces architectures disposent maintenant de multiples processeurs de calcul, multi-coeurs et interconnectés par des bus et mémoires à haute vitesse. Un serveur de classe moyenne peu coûteux peut soutenir une vitesse agrégée de paquets par secondes de l'ordre de 10 Mp/s (millions de paquets par seconde) pour une taille de paquet minimale (64 octets) [EGI 10]. Pour les paquets plus longs (1500 octets) cette même architecture est capable de saturer 12 interfaces gigabits full-duplex à la vitesse du lien (12 Gb/s en aggrégé). Bien que ces mesures de performances soient idéalisées, les routeurs purement logiciels peuvent être très utiles dans beaucoup d'environnements car ils offrent un haut niveau de programmabilité et de flexibilité.

La réalisation d'une plateforme d'exécution de routeurs purement logicielle qui soit performante et économique permet d'envisager la création d'architectures et de services réseaux polymorphiques [ZHU 08, HE 08, WAN 08]. A la condition que sa conception prenne en compte les avancées récentes des systèmes comme la forte baisse du coût des switches programmables, la virtualisation et les processeurs PC multi-coeurs [GRE 09] une même infrastructure réseau physique peut devenir programmable à souhait en fonction de l'usage tout en offrant des critères d'isolation, de performances et d'équité qui étaient impossible à garantir à l'époque des réseaux actifs.

Nous avons montré dans des travaux précédents [EGI 08b] que le principal goulôt d'étranglement de ces architectures est l'accès mémoire, à travers une combinaison de latence mémoire élevée à une surcharge de la mémoire et du Front Side Bus (FSB). Ce goulôt d'étranglement sera certainement réduit par l'introduction des architectures NUMA (Non-Uniform Memory Access) sur PC. Ces architectures utilisent une interconnexion en grappe (à la place du FSB) entre les processeurs en même temps que l'utilisation d'un contrôleur mémoire pour chaque processeur. Malgré ces progrès au niveau du matériel, les problèmes liés à la latence mémoire ont de grande chance de persister pour quelque temps.

Ainsi, pour que les routeurs logiciels conservent un maximum de performances, il est crucial que le nombre d'accès mémoire soit réduit au minimum. Sur un PC multi-coeur ceci peut être accompli en s'assurant que les paquets, aussi bien que les structures de données nécessaires à leur traitement, restent dans la mémoire cache du processeur lorsqu'ils sont transférés d'une interface à l'autre. Ceci introduit la notion de *hierarchie de cache*, c'est à dire l'ensemble des caches multi-niveau présents dans un processeur representant une ressource matérielle importante à prendre en considération lors de la conception de routeurs logiciels.

La figure 1 montre l'exemple de quatre hiérarchies de cache à 2 niveaux pour les deux processeurs à quatre coeurs utilisés lors de nos expérimentations : Chaque coeur contient un cache de niveau L1 non partagé et dont la taille est petite. Chaque paire de coeurs du processeur partage un cache de niveau 2 dont la taille est plus élevée. De manière générale, plus un cache est situé près du coeur, plus son accès est rapide, plus sa taille est petite et moins le nombre de coeurs qui le partagent est élevé.

Cette organisation hiérarchique reflète une réalité économique qui équilibre performances et coût d'un système par l'utilisation de mémoires très rapides (mais très chères) proches des coeurs de cal-

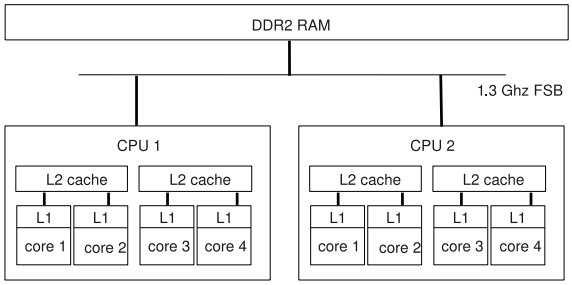


Figure 1. *Hiérarchie de cache processeur à trois niveaux*

culs afin d'obtenir un gain de performances important par la mise en cache des données lues initialement dans une mémoire principale de taille importante et peu chère, mais très lentes [HEN 03]. Aujourd'hui la différence de vitesse d'accès à une donnée située dans une mémoire de type cache de niveau 1 et la mémoire principle de type DDR2 peut atteindre un facteur 100 [WIC 08]. En conséquence, la performance des programmes éxécutés sur des PC modernes dépend fortement de l'utilisation judicieuse de la hiérarchie de cache présente sur la machine.

La conception d'un routeur logiciel peut être vue comme une suite d'unités de traitement éxécutés d'une interface à l'autre par la machine. La difficulté réside dans l'association judicieuse de ces unités de traitement à un processeur de calcul général à architecture multi-coeur tout en réduisant le nombre d'accès mémoire pour pouvoir obtenir une vitesse de commutation de paquets élevée. Comme nous allons le voir dans cet article, les solutions simplistes à ce problème ne sont pas très efficaces et la conception d'un plan de commutation efficace sur une telle plateform n'est pas triviale.

Nous avons montré dans un travail précédent qu'il est possible de fournir des critères d'isolation et d'équité entre un ensemble de routeurs virtuels éxécuté sur une même plateforme et dans le même contexte processeur à la condition importante d'ordonnancer les différentes unités de traitement des différents routeurs en fonction du nombre d'accès à la mémoire principale qu'ils provoquent[EGI 08a]. Dans une même instance de routeur, chaque unité de traitement peut avoir un nombre d'accès mémoire différent, ce qui complique l'ordonnancement global des routeurs virtuels. Dans cet article, nous définissons des groupes d'unités de traitement afin de pouvoir les associer plus facilement aux hiérarchies de cache de la machine. Ceci permet de simplifier l'ordonnancement des différentes instances de routeurs virtuels entre elles.

Nous contribuons par la définition de la notion d'*arbre de commutation* (i.e l'ensemble des unités de traitement nécessaire pour transférer un paquet d'une seule interface d'entrée vers toutes les interfaces de sortie possibles) comme l'unité d'ordonnancement de base pour l'association des composants d'un routeur à une hierarchie de cache (telle que celle présente dans le cache d'un processeur PC). Un *arbre de commutation* peut être vu comme une unité d'ordonnancement entre les routeurs virtuels, afin d'obtenir équité et isolation entre les différentes instances de routeurs en éxécution tout en conservant des performances intéressantes, puisque minimisant leur nombre d'accès à la mémoire principale.

La suite de cet article est organisée comme suit : En section 2 Nous décrivons l'idée des arbres de commutation plus en détail et évaluons les différentes stratégies pour l'assignement des arbres aux coeurs de calculs tout en évitant un changement de contexte coûteux entre les différents coeurs du processeur. La section 3 explore l'impact du coût des verrous utilisés pour synchroniser l'accès des arbres aux interfaces de sortie. La section 4 propose et évalue une modification des arbres de commutation pour diminuer la contention présente sur les interfaces de sortie. Nous concluons en section 5.

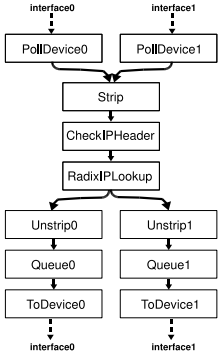
2. Arbres de commutation

Pour pouvoir maximiser les performances d'un routeur logiciel moderne sur PC, l'ensemble des coeurs de calcul du PC doivent être exploités, tout particulièrement pour ce qui concerne leurs hiérarchies de cache (Nous définissons une hiérarchie de cache comme l'ensemble des niveaux de cache présents à l'intérieur d'un processeur.) Un PC peut potentiellement contenir plusieurs processeurs et coeurs arrangés différemment selon leur l'architecture et donc contenir plusieurs hiérarchies de cache.

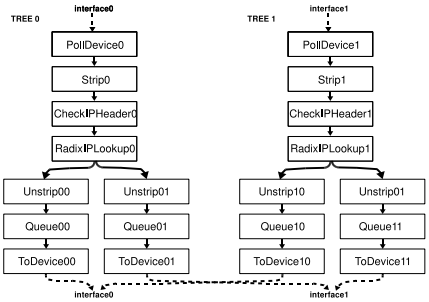
L'objectif à atteindre lors de l'implémentation d'un routeur logiciel sur un PC multi-coeur moderne est donc de garder un paquet le plus profondément possible à l'intérieur d'une hiérarchie de cache (i.e, le plus près des coeurs) tout en distribuant le traitement des paquets sur le plus grand nombre possible de coeurs situés dans la même hiérarchie de cache. Ceci permet de s'assurer que la vitesse de traitement des paquets ne soit pas limitée par la rapidité de calcul d'un seul coeur tout en réduisant les accès coûteux à la mémoire principale au minimum.

L'organisation interne du plan de commutation d'un routeur peut être vue comme un graphe interconnectant la séquence des élements de traitement ayant lieu sur chaque paquet qui traverse le routeur, aussi appellée configuration. Par exemple, la figure 2(a) illustre une configuration de commutation IP bi-directionnelle pour un routeur avec deux interfaces implémenté avec le logiciel Click modular router [KOH 00]. Atteindre les meilleurs performances que peuvent offrir le matériel est un problème non trivial : Bien que l'allocation de l'ensemble des éléments de traitement sur un seul coeur (i.e une seule hiérarchie de cache) soit possible, cette option n'est pas pour autant la plus optimale, puisque confiner l'ensemble des élements sur un sous-ensemble des coeurs de la machine peut résulter sur une utilisation non efficace de l'ensemble des ressources disponibles.

La clef pour résoudre ce problème est la décomposition du graphe du routeur en une série d'*arbres de commutation*. Chacun de ces arbres est associé avec une interface d'entrée du routeur et représente tout les chemins de traitement possibles que suivent les paquets lorsqu'ils entrent par cette interface. Comme un arbre de commutation contient l'ensemble des chemins possibles qu'un paquet peut prendre de l'entrée à la sortie du routeur, l'associer à une seule hierarchie de cache augmente la probabilité que le traitement des paquets se fassent en cache, et augmente donc la vitesse de commutation du routeur. La figure 2(b) illustre cette idée en montrant une configuration identique à celle de la figure 2(a) mais en utilisant cette fois des arbres de commutation. Comme on peut le voir, cette configuration consiste en deux arbres de commutation enracinés aux interfaces d'entrée, chacune avec son propre jeu indépendant d'élements de traitement.



(a) Configuration logicielle standard pour un routeur IP



(b) Configuration basée sur les arbres de commutations pour un routeur IP

Figure 2. *Configurations de routeurs logicielles*

L'avantage de l'utilisation des arbres de commutation est que ses élements peuvent être alloués à la même hiérarchie de cache, ce qui confine le traitement d'un paquet à sa hiérarchie et son coeur de traitement associé. Ceci réduit par conséquent le nombre d'accès à la mémoire principale. De plus, les arbres de commutations constituent une unité d'ordonnancement plus petite qu'un routeur, ce qui simplifie l'allocation l'allocation des ressources pour une plateforme de routage virtuelle. Concrêtement chaque arbre de commutation est éxécuté sur le coeur de calcul associé à la hiérarchie de cache désirée par une tâche dans le noyau Linux.

Bien que les arbres de commutations puissent être alloués aux différentes hiérarchies de cache de manière indépendante, ils ne sont pas complètement indépendants les uns des autres. En effet, les arbres de commutations qui appartiennent au même routeur (virtuel) doivent envoyer leurs paquets vers la même interface de sortie matérielle. L'accès à cette interface est exclusif et nécessite l'utilisation d'un mutex logiciel pour s'assurer que les différentes instances des arbres de commutation n'y accédent pas en même temps. Un verrou est associé à chaque interface de sortie et n'importe quel élément voulant écrire un paquet sur cette interface (l'élement "ToDevice" sur la figure) doit acquérir le verrou correspondant avant de pouvoir envoyer un paquet.

Le mécanisme de verrou Linux est essentiellement implémenté en utilisant une intruction "test and set" du processeur, qui permet de tester la valeur d'une variable et de la changer de manière atomique si c'est nécessaire. Une tâche qui désire acquérir le verrou teste sa valeur en permanence par une boucle active jusqu'à ce qu'elle l'acquière (nous utilisons l'implémentation spinlock sous Linux). Cependant dans le contexte d'un routeur logiciel PC performant, continuellement boucler sur la valeur d'un verrou est coûteux, puisque les autres élements de l'arbre de commutation pourraient re-utiliser les ressources dépensées pour boucler pour pouvoir envoyer ou traiter d'autres paquets. Dans l'implémentation Click le verrou est utilisé à chaque fois que l'élement ToDevice nécessite un accès exclusif à la file d'écriture matérielle sur la carte réseau. Les files de lecture et d'écriture matérielles sont séparées sur les cartes que nous utilisons. Le processeur est libéré à chaque fois que l'acquisition du verrou échoue, ce qui permet à des éléments ayant besoin du processeur d'être éxé-

cutés (nous utilisons l'implémentation spin_trylock sous Linux). Cette solution permet d'augmenter les performances d'un routeur dans des scénarios simples tels qu'un routeur transportant du trafic qui n'induit pas de contention sur l'accés aux interfaces de sortie. Cependant cette optimisation ne s'applique pas dans le cas général d'un routeur commutant du trafic vers une interface partagée par plusieurs arbres de commutation comme nous allons le voir dans la section suivante[1].

3. Impact des verrous

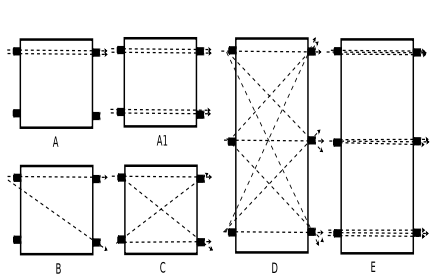
Bien que le but admis des arbres de commutations soit de réduire le nombre d'accès mémoire par le confinement du traitement d'un paquet sur une même hiérarchie de cache, certains accès mémoire ont tout de même lieu. Par exemple nous utilisons des verrous pour synchroniser l'accès aux interfaces de sorties de chaque arbre de commutation. Comme un verrou est une variable "globale" qui est accédée potentiellement depuis des hiérarchies de cache différentes, chaque changement de l'état du verrou en mémoire a pour conséquence l'invalidation de sa valeur dans le cache du coeur. Chaque valeur du verrou est alors marquée comme invalide dans toutes les hiérarchies de cache à part dans celle où le verrou a été acquis puis libéré. En conséquence et de manière plus détaillée chaque élement ToDevice qui réside sur une hierarchie essayant d'accéder la valeur d'un verrou avec celle-ci ayant été invalidée déclenche un accès mémoire et utilise un nombre précieux de cycles processeur.

3.1. *Impact sur les performances*

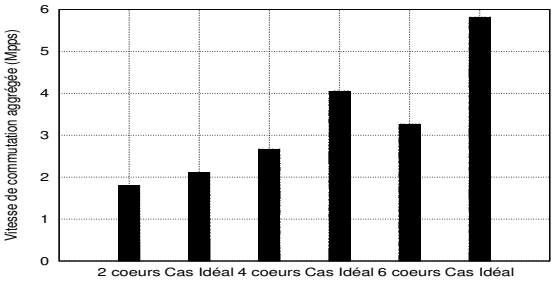
Pour évaluer l'impact de l'accès aux verrous sur les performances globales de nos routeurs, nous effectuons l'expérience suivante avec des routeurs configurés de manière identique, mais traitant deux flux de trafic différents. Dans le "Cas Idéal", le trafic est injecté entre des paires exclusives d'interfaces d'entrée et de sortie comme on peut le voir par exemple sur la figure 3(a) scénario A1 pour un routeur 2x2 et scénario E pour un routeur 3x3. Pour chaque arbre de commutation, une seule branche traite des paquets et les arbres ne sont jamais en concurrence les uns avec les autres pour la même interface de sortie (Le verrou correspondant est toujours libre lorsque l'élement ToDevice essaie de l'acquérir). Ce cas idéal est ensuite comparé à un scénario avec une matrice de trafic complète entre chaque interface d'entrée vers toutes les interfaces de sortie. Dans ce scénario la même quantité de trafic est présente sur chaque interface d'entrée et chaque interface de sortie. Les arbres sont constamment en concurrence pour accéder aux interfaces de sortie comme on peut le voir par exemple sur la figure 3(a) scénario C pour un routeur 2x2 et scénario D pour un routeur 3x3. Dans l'ensemble de nos expérimentations, le trafic injecté est de type CBR et utilise une taille de paquet ethernet minimale (64 octets). Nous avons répété les expériences avec une taille de paquet de 1500 octets et observé la saturation des liens dans chaque scénario.

En figure 3(b) on peut voir les résultats comparatifs pour un routeur 2x2 (2 ports d'entrée et 2 ports de sortie), un routeur 4x4 et un routeur 6x6. Chaque arbre de commutation est assigné à un

1. Nous avons effectué l'ensemble des expérimentations sur une machine Dell PowerEdge 2950 avec deux processeurs 2.66Ghz quad-coeurs Intel X5355. Ces systèmes ont 8 GB de mémoire DDR2 667Mhz et 3 cartes Intel Gigabit Ethernet quadruple port, chacune utilisant un canal PCIe x4, pour un total de 12 ports Gigabit.



(a) Scénarios d'évaluation



(b) Performances pour les cas non optimisés des routeurs 2x2, 4x4 and 6x6

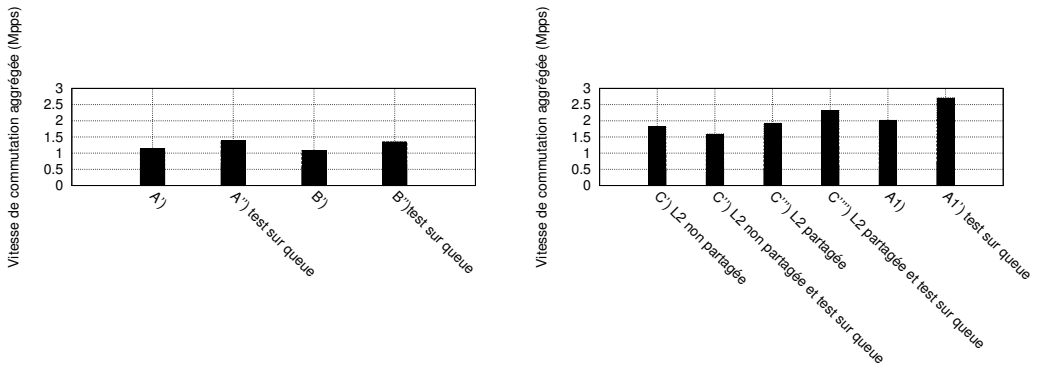
Figure 3. *Performances de base et scénarios d'évaluation*

coeur différent (Remarquez qu'il y a autant d'arbre de commutations qu'il y a d'interfaces d'entrée). Les résultats montrent clairement que la différence augmente en fonction du nombre d'interfaces présentes sur les routeurs entre le cas idéal et le cas où il y a une contention pour l'accès aux interfaces de sortie. Dans le cas le pire (avec le routeur 6x6), il y a le plus d'arbres de commutation en concurrence pour un même verrou (6 exactement) et on peut observer une perte de performances d'environ 45% rapport au cas idéal. Comme les branches individuelles de commutation de chaque arbre sont identiques et que les états maintenus par les élements intermédiaires (tels que la table de commutation) sont suffisament petits pour être contenus dans le cache de chaque coeur, nous concluons que la différence de performances est causée par une contention sur les interfaces de sortie (i.e, les opérations d'attente et de libération du verrou). Le reste de cet article s'attache à réduire cette contention en prenant avantage des hiérarchies de cache et en effectuant des optimisations logicielles sur l'accès au verrou. Pour clarifier la description des résultats qui suivent en figures 4 et 5, chaque colonne a un label unique dont la première lettre se refère aux scénarios illustrés en figure 3(a). Les lettres suivantes identifient les différentes combinaisons d'optimisation logicielles et de partage du cache processeur matériel.

3.2. *Réduction de la contention*

Dans l'expérimentation précédente, nous avons intentionnellement créé un maximum de contention sur les verrous en ayant les éléments ToDevice qui essaient d'acquérir le verrou pour l'accès aux interfaces de sortie aussitôt que l'élement est exécuté. Ceci est clairement non nécessaire et détériore sûrement les performances des autres élements ToDevice essayant d'accéder à l'interface alors qu'ils ont des paquets à envoyer. L'élément ToDevice essaye d'acquérir le verrou y compris lorsqu'il n'a pas de paquet à envoyer, ce qui implique une invalidation de cache coûteuse pour les coeurs éxécutant les autres arbres accédant la même interface de sortie. Dans l'expérience suivante, nous modifions l'élement ToDevice pour qu'il essaie d'acquérir le verrou seulement lorsqu'il y a des paquets à envoyer dans la queue en amont. Dans l'implémentation Click cette information est facilement accessible puisqu'elle permet de partager des états entres les différents élements de la configuration. Nous utilisons les 6 scénarios de la figure 3(a) pour illustrer les effets de cette réduc-

tion de contention pour l'accès au verrou. Chaque scénario exécute une configuration composée d'un graphe complet connectant 6 interfaces d'entrée et 6 interfaces de sortie. Cette configuration utilise 6 coeurs de calcul (1 par arbre de commutation). Seuls les chemins de commutation illustrés par une ligne à pointillés sont activement impliqués dans la commutation de paquets dans la configuration, tous les autres sont inactifs (i.e, ils ne traitent aucun trafic) et donc ordonnancés que très peu souvent grâce à l'optimisation Click que nous avons décrit à la fin de la section 2.



(a) Scénarios d'arbres de commutation, un coeur (b) Scénarios d'arbres de commutation, deux coeurs

Figure 4. *Impact des verrous sur les performances*

La figure 4(a) montre une comparaison de performances avec et sans la réduction de la contention due au verrou pour les scénarios A et B. Les résultats montrent clairement que l'on peut éviter la contention due à la présence d'arbres de commutation inactifs qui acquièrent le verrou lorsque celà n'est pas nécessaire. En testant d'abord si il y a des paquets à envoyer, l'acquisition du verrou peut être évitée et les performances du routeur éxécutant l'ensemble de la configuration peuvent être améliorées. Notons qu'un "arbre de commutation inactif" est simplement un arbre qui ne transporte pas de trafic : Les élements dans l'arbre sont toujours ordonnancés et éxécutés et la version non optimisée de l'élement ToDevice essaie d'acquérir un verrou à chaque exécution contrairement à la version optimisée. On peut observer une augmentation de performances aussi bien pour le scénario A" dans lequel un seul verrou (c'est à dire une seule interface de sortie) est accédé que pour le scénario B" pour lequel deux verrous sont accédés en parallèle. En vérifiant simplement si il y a des paquets à envoyer dans la queue avant d'accéder au verrou les performances observées dans le scénarios B" sont quasiment équivalentes à celles observées dans le scénario A'.

Dans l'expérience précédente, seulement une ou deux branches appartenant à un seul arbre étaient actives. L'ensemble de la commutation était effectuée par un seul coeur. En figure 4(b) nous examinons les interactions entre deux coeurs, chacun commutant les paquets sur deux branches et étant en compétition sur les verrous d'accès aux interfaces de sortie, ce qui nous donne une matrice pleine de trafic unidirectionnelle 2x2 telle que illustrée dans le scénario C. Le scénario A1 est un scénario témoin qui illustre une matrice de trafic sans contention sur les interfaces de sortie puisque

le verrou est toujours dans le cache déservant une seule branche de l'arbre de commutation associé. Lorsqu'on observe les résultats du scénario C on peut voir que lorsque les paquets partagent une hiérarchie de cache, vérifier si il y a des paquets dans la queue qui précéde l'élement ToDevice augmente les performances (résultats C"' et C""). Lorsque les coeurs ne partagent pas de cache la vérification de l'état du verrou entraine un défaut de cache. On peut observer que si les deux coeurs ne partagent pas leur cache L2, l'optimisation précédente qui consistait à vérifier l'état de la queue en amont avant d'envoyer devient alors une source de réduction de performances avec une perte absolue d'environ 290 kpps entre les résultats C' et C". Entre la pire performance (C") et la meilleure performance (C"") du scénario C on observe une différence absolue de 720 kpps, avec la meilleure performance à une différence absolue de 400 kpps du résultat idéal A1'. Ceci illustre l'importante d'une conception qui soit adaptée à la hiérarchie de cache sous-jacente.

4. Optimisation des arbres de commutation

La section précédente a montré que la contention sur les verrous d'accès aux interfaces de sortie d'un routeur logiciel contribue à la diminution globale de ses performances. De plus, même l'optimisation qui consiste à vérifier si il y a des paquets à envoyer dans la queue avant d'essayer d'acquérir le verrou consomme des cycles processeurs précieux sur un routeur potentiellement très occupé tel qu'un routeur avec un nombre important d'interfaces ou une plateforme de routeurs virtuels supportant un nombre important d'entre eux.

4.1. *Un seul ToDevice par interface*

Une façon naturelle d'éviter la contention due aux verrous est d'éliminer totalement le besoin des verrous dans notre architecture. Ceci peut être accompli en associant *un seul* élément ToDevice à chaque interface de sortie et en associant une queue en amont de cet élement à chaque branche de chaque arbre de commutation accédant à cette interface. De cette façon la branche de l'arbre de commutation correspondante peut envoyer des paquets vers cette interface en envoyant des paquets dans la queue correspondante qui termine cette branche. Les paquets sont ensuite envoyés en sortie par un coeur différent qui peut résider ou non dans la même hiérarchie de cache, selon la topologie du processeur.

La figure 5(a) montre les résultats obtenus avec une telle configuration, dans les cas où les différents arbres de commutation actifs et l'unique élément ToDevice par sortie partagent ou non la mémoire cache de deux coeurs (scénarios B et C). Pour des raisons de lisibilité et pour illustrer les gains ou pertes de performances possibles, les scénarios témoins A1 (deux coeurs) et A (un seul coeur) sont reproduis sur la figure. De manière générale on peut tout de suite constater que lorsqu'un paquet doit traverser une hiérarchie de cache les performances diminuent.

Pour le scénario B, lorsque le cache de niveau 2 n'est pas partagé (résultat B"), les performances sont significativement réduites car la queue située avant l'élément ToDevice contient des états partagés qui sont accédés aussi bien par le coeur déposant les paquets dans la queue que par le processeur les retirant de la queue. En fait on arrive à une situation contradictoire où les performances du rou-

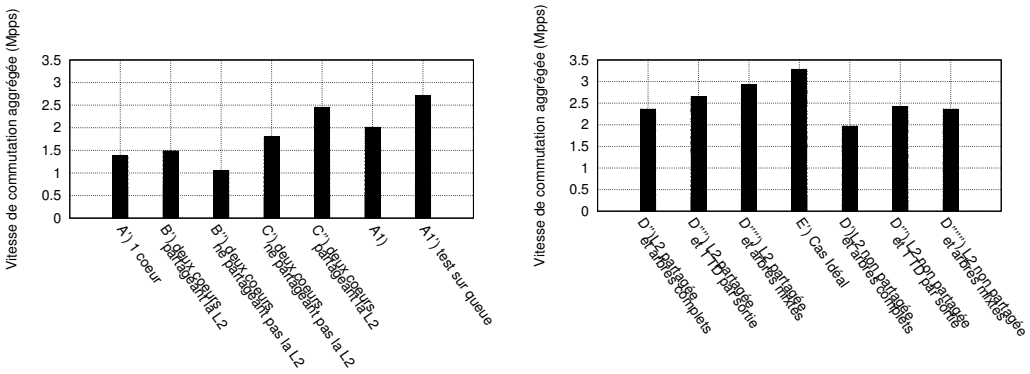
teur sont pires lorsqu'il utilise deux coeurs de calcul qui ne partagent pas de L2 (résultat B") que lorsqu'il utilise un seul coeur (scénario A).

Le nombre de défauts de cache augmente lorsqu'on augmente le nombre de coeurs comme dans le scénario C qui affiche des performances pires que ceux observés en section 3. Cependant lorsque le cache de niveau 2 est partagé (résultat C"), les performances augmentent par rapport aux cas précédents lorsque les élements ToDevice faisaient partie des arbres de commutation et on n'observe plus qu'une différence d'environ 250 kpps entre le résultat témoin A1' et le résultat C". Ceci s'explique par le fait que dans les deux cas, les paquets n'ont pas besoin de traverser une hiérarchie de cache, mais dans le cas présent la contention est réduite sur le verrou car il n'y a essentiellement qu'un seul élément ToDevice par hiérarchie de cache. En figure 5(b) on peut voir que les résultats de la configuration qui utilise un seul élément ToDevice par interface de sortie et par hiérarchie de cache offre les meilleures performances. Nous l'étudions dans la section suivante.

4.2. *Un seul ToDevice par interface par hiérarchie de cache*

Un élément ToDevice par interface a pour conséquence que certains paquets doivent traverser les hiérarchies de cache, ce qui entraine une baisse de performance, comme on ne peut pas garantir que le nombre de coeurs sur la même hierarchie de cache sera égal au nombre d'interfaces de sortie du routeur. Un élément ToDevice par coeur entraine une perte de performance en raison du problème de l'accès concurrent au verrou de l'interface. Une solution logique, hybride à la solution utilisant un seul élément ToDevice par port est la solution utilisant un seul élément ToDevice par coeur par port.

Les résultats obtenus avec ce nouveau principe sont présentés sur la figure 5(b) et labellés sous le nom "d'arbres mixtes". Les résultats labellés "arbres complets" correspondent aux performances que l'on obtient avec l'allocation initiale des arbres présentée en section 3. Nous comparons trois options pour le placement de l'élément ToDevice avec le "Cas Idéal" seulement avec des routeurs 3x3 (scénarios D et E). Afin d'éviter une limitation sur la rapidité des coeurs et pour amener le système



(a) Un élément TD par interface

(b) Arbres mixtes

Figure 5. *Comparaison arbre mixtes et un élément TD par interfaces de sortie*

à une contention sur l'accès mémoire en premier lieu, nous utilisons 4 coeurs dans le scénario avec tous les arbres de commutations, 3 coeurs avec un seul élement ToDevice par sortie et 4 coeurs dans le cas avec les arbres mixtes et 3 coeurs dans le cas idéal. En effet notre plateforme de test ne nous permet d'associer que deux coeurs par hiérarchie de cache, ce qui nécessite un nombre minimal de 4 coeurs sur un routeur 3x3 commutant du trafic vers toutes les interfaces de sortie. Nous testons ensuite les performances du système lorsque les paires de coeurs partagent ou non un cache de niveau 2 avec les différentes configurations d'arbres énoncées dans les sections précédentes. Toutes les optimisations précédemment discutées sont activées. On peut observer que les meilleures performances sont effectivement obtenues lorsque toutes les paires de coeurs partagent un cache de niveau 2 et que nous avons placé un élement ToDevice par hiérarchie de cache (résultat D"") avec une valeur absolue d'environ 3Mpps (et 3.3Mpps dans le résultat idéal E').

Lorsque le cache de niveau 2 est partagé, avoir un seul élément ToDevice par interface par hiérarchie de cache améliore les performances par rapport aux deux autres cas. Lorsque le cache de niveau 2 n'est pas partagé une perte mineure est observable en comparaison avec le cas où on utilise un seul élément ToDevice par interface de sortie (résultats D"' et D""). La configuration optimisée où les paquets restent toujours sur le même coeur est utilisée dans le scénario E. Bien sûr, ce scénario n'est pas réaliste, puisque l'on doit s'attendre à ce que le trafic change constamment d'interfaces. C'est pourquoi nous croyons que le compromis que nous offrons dans cette configuration est à la fois flexible est efficace.

5. Conclusion et travaux futurs

Dans cet article nous avons présenté une nouvelle méthodologie pour la conception d'une architecture de commutation de paquets pour les routeurs logiciels éxécutés sur du matériel moderne PC multi-coeurs. Nous avons montré qu'obtenir les meilleurs performances sur une telle plateforme nécessite un compromis entre l'utilisation de plusieurs coeurs de calcul et le maitient du nombre d'accés mémoire à un minimum. Dans ce but, nous avons proposé le concept d'arbre de commutation utilisant un élément ToDevice partagé par hiérarchie de cache et par interface de sortie. Nous avons identifié que le verrou de l'interface de sortie est une source significative de contention entre les différents coeurs voulant y accéder. Nous avons résolu partiellement ce problème par l'utilisation d'un élément ToDevice partagé par hiérarchie de cache. Ce problème de contention pourrait être supprimé totalement si les interfaces de sortie utilisaient des queues multiples dans le matériel, avec au moins une hiérarchie de cache présente dans le système, mais de préférence une par coeur de calcul. Cette tendance est déjà en train de se développer avec un nombre important de cartes réseaux supportant des queues matérielles multiples.

Dans des travaux futurs, nous avons l'intention d'explorer les problèmes liés aux élements de calculs qui contiennent des états par paquet, tels que les filtres de paquets ou les traffic shapers. Dans le modèle traditionnel de chemin de commutation, les flux passant d'une interface à l'autre passent à travers un seul élement. Cependant dans les arbres de commutation soit nous avons des copies de l'élement qui agissent de manière indépendente les unes des autres, indépendement des flux, soit nous pouvons combiner les arbres en un seul chemin "slow path", tel que c'est le cas pour de nombreuses opérations complexes dans les routeurs matériels traditionnels.

6. Bibliographie

[ARG 08] ARGYRAKI K., BASET S. A., CHUN B.-G., FALL K., IANNACCONE G., KNIES A., KOHLER E., MANESH M., NEDVESCHI S., RATNASAMY S., « Can Software Routers Scale ? », *Proceedings of PRESTO'08*, Seattle, USA, August 2008.

[EGI 08a] EGI N., GREENHALGH A., HANDLEY M., HOERDT M., HUICI F., MATHY L., « Fairness Issues in Software Virtual Routers », *Proceedings of PRESTO'08*, Seattle, USA, August 2008.

[EGI 08b] EGI N., GREENHALGH A., HANDLEY M., HOERDT M., HUICI F., MATHY L., « Towards High Performance Virtual Routers on Commodity Hardware », *Proceedings of ACM CoNEXT 2008*, Madrid, Spain, December 2008.

[EGI 10] EGI N., IANNACCONE G., KNIES A., MANESH M., GREENHALGH A., HANDLEY M., MATHY L., RATNASAMY S., « Improved Forwarding Architecture and Resource Management for Multi-core Software Routers », *Proceedings of IFIP NPC 2010*, Gold Coast, Australia, October 2010.

[GRE 09] GREENHALGH A., HUICI F., HOERDT M., PAPADIMITRIOU P., HANDLEY M., MATHY L., « Flow processing and the rise of commodity network hardware », *SIGCOMM Comput. Commun. Rev.*, vol. 39, n° 2, 2009, p. 20–26, ACM.

[HE 08] HE J., ZHANG-SHEN R., LI Y., YEN LEE C., REXFORD J., CHIANG M., « DaVinci : Dynamically Adaptive Virtual Networks for a Customized Internet », *Proceedings of ACM CoNEXT 2008*, Madrid, Spain, December 2008.

[HEN 03] HENNESSY J., PATTERSON D., *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann, 2003.

[KOH 00] KOHLER E., MORRIS R., CHEN B., JAHNOTTI J., KASSHOEK M. F., « The Click Modular Router », *ACM Transaction on Computer Systems*, vol. 18, n° 3, 2000, p. 263-297.

[WAN 08] WANG Y., KELLER E., BISKEBORN B., VAN DER MERWE J., REXFORD J., « Virtual routers on the move : Live router migration as a network-management primitive », *Proceedings of SIGCOMM'08*, Seattle, USA, August 2008.

[WIC 08] WICKIZER S. B., CHEN H., CHEN R., MAO Y., KAASHOEK F., MORRIS R., PESTEREV A., STEIN L., WU M., DAI Y., ZHANG Y., ZHANG Z., « Corey : An Operating System for Many Cores », *In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, California, USA, August 2008.

[ZHU 08] ZHU Y., ZHANG-SHEN R., RANGARAJAN S., REXFORD J., « Cabernet : Connectivity Architecture for Better Network Services », *Proceedings of ACM ReArch '08*, Madrid, Spain, December 2008.