Monitoring du noyau Linux sur une architecture NUMA

Kévin Gallardo, Eric Lombardet, Pierre-Yves Péneau

Université Pierre et Marie Curie - Jussieu - Paris VI

Table des matières

Résumé

L'essor de l'informatique en nuage a permis aux administrations et entreprises de stocker d'énormes jeux de données. Aujourd'hui, l'un des goulots d'étranglement majeurs pour les performances de traitement de ces données est le système d'exploitation de chaque machine. Les systèmes actuels ne peuvent pas gérer efficacement les applications intensives en données car ils ne disposent pas d'une vue unifiée des ressources utilisées, ce qui les empêche de déterminer des stratégies efficaces pour le placement des tâches/données sur les ressources matérielles. Une meilleure gestion des ressources permettrait une forte réduction du nombre de machines nécessaires aux traitements des données.

L'implémentation de sondes dans le noyau Linux permettrait d'identifier les ressources physiques et logicielles les plus sollicitées par les processus. Les informations remontées par ces sondes peuvent ensuite permettre de commencer à définir des stratégies de placement des tâches et des données prenant en compte à la fois la topologie de la machine et l'utilisation effectives des ressources par les tâches.

1 Introduction

Les systèmes multicoeurs modernes sont maintenant basés sur l'architecture NUMA (Non Uniforme Memory Access). Avec un système NUMA, les coeurs des processeurs sont regroupés en noeuds. Chaque noeud possède un contrôleur mémoire et est interconnecté avec les autres noeuds de la machine.

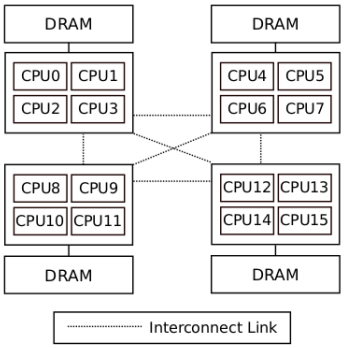

Figure 1.1 – *Un système NUMA avec 4 noeuds et 4 coeurs par noeud*

Du fait des temps d'accès mémoire non uniforme, tout le défi des systèmes tournant sur cette architecture est la répartition des données et des traitements. En effet, la principale cause de latence n'est pas due au **temps de traitement des données**, mais au **temps d'accès aux données**. Ces accès coûtent entre 10% et 40% de temps supplémentaire par rapport aux accès locaux.[**?**] Dans une configuration idéale, chaque coeur irait chercher ce dont il a besoin dans la mémoire contrôlée par le noeud dans lequel il se situe. Ainsi, les demandes d'accès distant seraient réduites à néant, et il n'y aurait aucune latence due aux échanges entre les noeuds. Nous allons voir que cet idéal est très difficile, voire impossible à obtenir. Néanmoins, il est possible de s'en approcher, en mettant au point des algorithmes de répartitions de plus en plus efficaces. La création de ces algorithmes nécessite une connaissance approfondie du noyau : comment gère-t-il la création des threads, où sont-ils placés, quelles sont les pages mémoires accédées le plus souvent, par quels noeuds sont-elles contrôlées...C'est en collectant un maximum de renseignements sur ces différents points (et de nombreux autres) que l'on pourra être en mesure d'affiner les solutions de répartition de charge. Cette étape de monitoring sera le sujet principal de ce projet de master. Nous allons devoir lire et comprendre le fonctionnement à très bas niveau du noyau, puis le modifier en utilisant divers outils de gestions d'évènements avec des bibliothèques comme IBS [1] afin de préparer l'étape de réflexion pour la création d'algorithmes.

1. Instruction Base Sampling, une technologie développée par AMD uniquement sur les processeurs Opteron

2 Étude du contexte

Le but de ce projet sera dans un premier temps de mettre en place une infrastructure de compilation, de test et d'exécution d'un noyau Linux. La seconde partie du projet sera de comprendre comment fonctionne le noyau Linux au niveau de la mémoire, notamment pour la gestion des pages (emplacement, taille), et au niveau des processeurs pour le placement des threads et le parallélisme. Ensuite, il faudra se plonger dans la lecture du code et sa modification aux endroits adéquats en utilisant des technologies comme IBS où les hardware counters pour obtenir des informations précises sur la gestion des points évoqués ci-dessus. Enfin, pour tester ce noyau avec nos modifications, nous utiliserons la machine virtuelle et gdb pour le débugage.

2.1. Infrastructure

2.1.1. Machine virtuelle

Dans cette partie, nous allons détailler l'infrastructure dont nous disposons et celle que nous avons mise en place pour ce projet. Notre avons à notre disposition une machine AMD Opteron 6172 composée de quatre processeurs à douze coeurs chacun cadencés à 2,1GHz et répartis en 8 noeuds avec 32G de mémoire vive. (cf. Figure 2.1)

Sur cette machine, nous avons utilisé l'Hyperviseur qemu, avec son extension kvm pour afin d'optimiser l'émulation. Afin d'améliorer encore plus cette dernière, nous avons utilisé le logiciel virt-manager qui détecte automatiquement la configuration matérielle de machine hôte et configure la machine virtuelle en conséquence. Cette configuration est à notre avis très réaliste puisqu'elle est capable d'activer/désactiver des options CPU comme la gestion d'IBS où l'hypervision. Un des autres avantage de virt-manager est que l'on peut sauvegarder la configuration dans un fichier, et pouvoir ainsi l'exporter facilement.

Nous avons installé une machine virtuelle classique (Debian GNU/Linux), qui nous permettra par la suite de fournir à notre noyau compilé une architecture de base pour se lancer. En effet, la compilation du noyau se fera directement sur l'Opteron, mais le lancement et les tests se feront via l'hyperviseur. Afin que le noyau puisse se lancer et avoir une base, nous avons installé une première VM, donc nous récupérerons la configuration du noyau pour la compilation.

2.1.2. GDB mode remote

Les modifications apportées au noyau doivent être controlées en cas d'erreur dans le code pendant la periode de développement. Pour ce faire nous allons utiliser gdb, le debugger sous licence libre distribué par le groupe du projet GNU. L'utilisation que nous allons faire sur gdb n'est pas

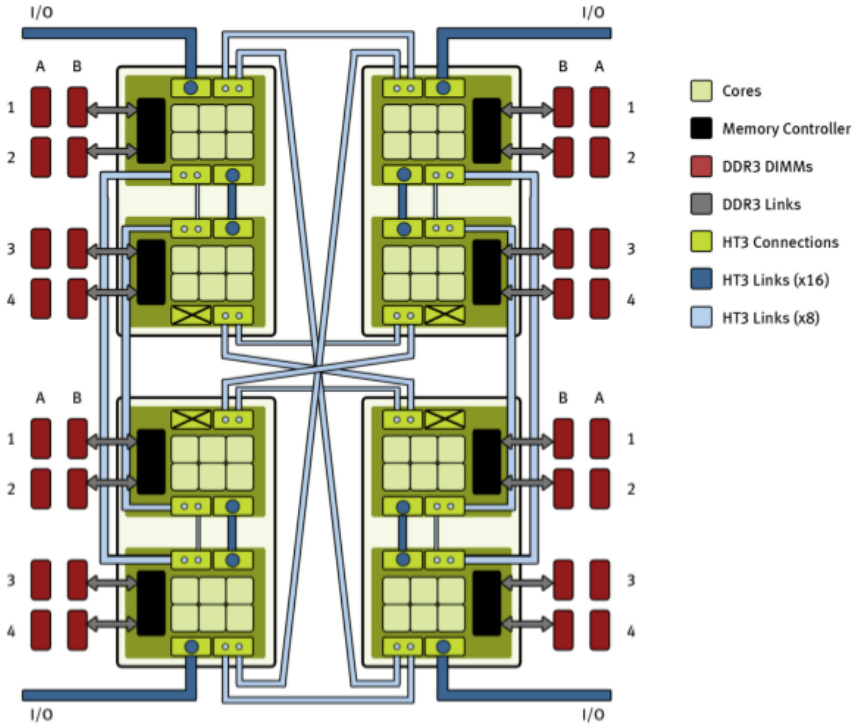

Figure 2.1 – *Topologie de la machine 6172*

une utilisation commune comme cela se fait sur un programme classique car dans notre cas, étant donné que c'est le noyau lui même qui est en train de se charger, il n'est pas possible d'executer un programme tel que gdb sur la machine. Pour pouvoir l'utiliser et débugger le noyau modifié, nous allons utiliser une option de gdb qui va nous permettre une execution controlée du noyau, plus précisément le control de breakpoints.

Effectivement, le mode remote de gdb va nous permettre de mettre en place une telle configuration. Et il se trouve que qemu va fournir également une option d'interface entre une VM et une machine distante. Ainsi, dans notre cas, il va falloir lancer premièrement qemu avec l'option « -s -S »qui va se charger de fournir le « gdb stub »nécéssaire a la connexion distante de gdb en mode remote, qemu se lance alors, sans lancer la machine virtuelle, en attente d'une connexion sur le port 1234 (par defaut) de la machine qui execute la machine virtuelle. Ensuite il nous faut donc lancer gdb en lui fournissant l'image du noyau Linux compilé afin que celui-ci puisse charger la table des symboles pour pouvoir placer les breakpoints :

```
gdb ./vmlinux
```

Ensuite nous lançons la connexion sur le gdb stub de qemu (ici nous lançons les deux programme sur la même machine, la connexion se fait donc en local) :

```
(gdb) target remote localhost:1234
```

Maintenant nous avons connecté gdb à qemu et nous pouvons commencer le debuggage du noyau (à distance) en placant des breakpoints là où il y en a besoin.

2.2. La gestion de la mémoire

Sur une architecture NUMA, les performances des applications dépendent fortement du placement mémoire. Ce placement est contrôlé par le système d'exploitation via le mécanisme de pagination offert par le matériel. Le mécanisme de pagination permet d'offrir une vue virtualisée de la mémoire. Lorsque la pagination est activée, les instructions d'accès à la mémoire manipulent des adresses mémoire dites virtuelles. La MMU (Memory Management Unit) d'un coeur convertit alors l'adresse mémoire virtuelle en adresse mémoire physique avant d'effectuer l'accès mémoire. Pour effectuer cette conversion, la MMU utilise une table des pages dont l'adresse est stockée dans un des registre du coeur. Cette table associe des plages d'adresses virtuelles de taille fixe à des plages d'adresses physiques de même taille. Dans le contexte de la pagination, cette plage d'adresse fixe est appelée une page.

Le système d'exploitation maintient une table des pages par processus, donc partagée par les threads du processus. Le système s'occupe de la remplir et de la modifier en fonction des besoins du processus. Techniquement, lorsqu'un processus alloue un espace mémoire à bas niveau (fonction mmap), le système lui donne une autorisation d'accès à une plage d'adresse virtuelle appelé XX [1] mais n'y associe pas encore de page physique. Elles sont associées aux pages virtuelles de façon paresseuse : à chaque fois que le processus accède à une page virtuelle qui n'est pas encore présente dans la table des pages, le processeur déclenche une faute des pages. Cette faute est rattrapée par une fonction du système qui s'occupe alors de trouver une page physique libre pour l'associer à la page virtuelle ayant déclanché la faute.

Sur une architecture NUMA, l'espace d'adressage physique est lui-même partitionné entre les noeuds NUMA. Les premières adresses physique sont associées au premier noeud, les suivantes au second etc...Lorsque le système d'exploitation associe une page virtuelle à une page physique dans une table des pages, il choisit donc sur quel noeud les accès à la page virtuelle seront effectués en fonction de l'adresse de la page physique. Le système peut aussi choisir de migrer une page virtuelle d'un noeud vers un autre en copiant la page physique vers un autre noeud et en mettant à jour la table des pages.

Pour effectuer un placement mémoire optimisant les performances des processus, le système d'exploitation doit donc être capable de sélectionner sur quel noeud placer les pages virtuelles. Ce choix dépend directement du coeur sur lequel s'exécute le thread qui accède à la page, le but étant d'essayer de placer le thread et les pages auxquelles il accède sur le même noeud. La question devient donc, quels sont les threads qui demandent le plus d'accès mémoire et sur quels noeuds sont-ils placés ? Si les accès sont distants, les noeuds sont-ils très éloignés ? Engendrent-ils une congestion dans les interconnexions ? Si oui, quel est le coût d'un déplacement de page mémoire ? Toutes ces questions et tous ces paramètres sont à prendre en compte lors des (dé)placements en mémoire des données.

Prenons l'exemple d'une application générant énormément d'accès mémoire. Le nombre de cache miss peut être très élevé si les données sont réparties équitablement sur différents noeuds, ou au contraire placées sur un seul noeud distant, avec un bus d'échange plus ou moins rapide avec le noeud demandant les données. Et là encore, tout dépend de l'application utilisant ces données. Certaines applications fonctionneront mieux si les données sont réparties équitablement sur les

1. TODO

noeuds, et au contraire certaine verront leur performances largement dégradées.

Une expérience illustrant ce phénomène a été réalisée l'année dernière[**?**]. L'équipe de recherche a « stressé »une machine NUMA avec différentes applications plus où moins gourmande en ressource. La mémoire est soit gérée en « First-Touch (F) », c'est à dire que les données sont placées dans le premier noeud entrainant un cache miss, soit en « interleaving (I) », et dans ce cas les données sont réparties équitablement entre les noeuds. Voici les résultats qu'ils ont obtenus :

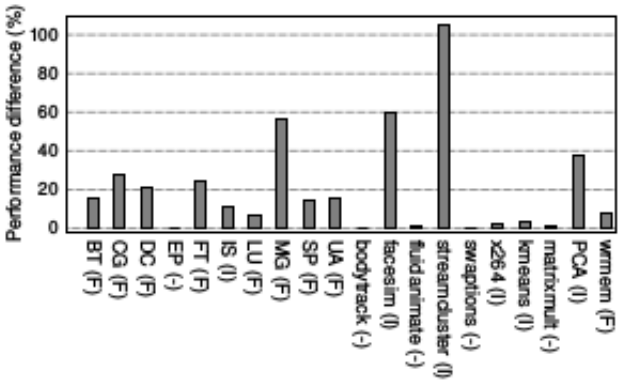

Figure 2.2 – *Différences de performances en fonction de la politique de placement en mémoire sur une architecture NUMA (First-Touch ou Interleaving)*

On voit que les résultats sont largement différents selon les applications et la politique de gestion. Ainsi, une de nos approches sera d'observer le fonctionnement de la mémoire selon ces différentes applications, étudier les défaut de pages, comprendre pourquoi certaines fonctionnent mieux que d'autres, et essayer d'identifier des « familles d'applications »ayant comme trait commun leur comportement vis-à-vis de la mémoire.

2.3. Communication entre threads

Un autre challenge quant à l'optimisation de performance sur architecture NUMA sont les communications et les echanges de données qui peuvent s'effectuer entre plusieurs threads de façon simultanée. En effet nous allons devoir fournir un moyen de mesurer ces communications afin que notre noyau puisse utiliser une politique de placement des thread qui leur permette de se retrouver les plus proches les uns des autres, de cette façon, leurs échanges ne se feront (idéalement) que sur un seul et même noeud. Nous effectuerons donc une grande partie de ce monitoring sur les échanges menés par des tubes.

2.4. But à terme

Ce projet PSAR a pour but d'être ensuite intégré au sein d'un projet de plus grande envergure, en effet, le but ici étant de fournir des informations et des outils de monitoring sur différentes activités du noyau. Cela pour permettre ensuite l'étude de différents algorithmes des placement mémoires au sein du noyau Linux qui permettra d'optimiser les performances des architectures NUMA (algorithmes de « ressorts »).

Bibliographie

[1] Baptiste Lepers, Phd. Thesis (2014). *Improving performances on NUMA systems architecture.*

[2] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. *Traffic Managment : A holistic Approach to Memory Placement on NUMA Systems*. Architectural Support for Programming Languages and Operating Systems (ASPLOS), Houston, USA, March 2013.

[3] 2005–2013 Advanced Micro Devices, Inc. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors* Rev 3.62 - January 11, 2013

[4] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, Bill Hughes, Advanced Micro Devices, Inc. CACHE HIERARCHY AND MEMORY SUBSYSTEM OF THE AMD OPTERON PROCESSOR

[5] Paul J. Drongowski *Basic Performance Measurements for AMD AthlonTM 64, AMD OpteronTM and AMD PhenomTM Processors* Advanced Micro Devices, Inc. Boston Design Center, 25 September 2008