Migration de processus pour un multi-noyau large échelle

Pierre-Yves Péneau, Mohamed Lamine Karaoui, Franck Wajsbürt

Compas'2016 : Parallélisme/ Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Migration de processus pour un multi-noyau large échelle

Pierre-Yves Péneau, Mohamed L. Karaoui et Franck Wajsbürt

Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier (LIRMM)
161 rue Ada - 34095 Montpellier - France
pierre-yves.peneau@lirmm.fr

Laboratoire d'Informatique de Paris 6 (LIP6)
4 place Jussieu - 75005 Paris - France
mohamed.karaoui@lip6.fr, franck.wajsburt@lip6.fr

Résumé
Pour continuer d'augmenter les performances, les architectes intègrent de plus en plus de cœurs dans les processeurs. Les architectures dites *manycore* avec des milliers de cœurs sont à prévoir prochainement. Pour répondre au défi énergétique, il existe des architectures manycore basées sur des cœurs 32 bits. La gestion d'une mémoire dont la taille est supérieure à 4Go est alors un problème. Une solution consiste à distribuer la gestion de la mémoire par plage de 4Go en remplacant le noyau monolithique par un multi-noyau. Cet article propose la mise en place et l'évaluation d'un service de migration de processus dans ce contexte multi-noyau. Ce travail est basé sur le multi-noyau ALMOS-MK et sur l'architecture manycore TSAR. Nos résultats montrent qu'il est possible de déplacer des processus sur une telle plateforme sans affecter les performances, permettant d'exploiter pleinement les ressources offertes et de conserver le budget énergétique des architectures 32 bits.

---

1. Introduction

Depuis 15 ans, les fabricants de processeurs ont atteint une limite pour l'augmentation de la fréquence des processeurs [1]. Pour continuer d'augmenter la puissance de calcul, les architectes intègrent de plus en plus de cœurs. Parallèlement, on observe une augmentation de la quantité de mémoire disponible dans les machines. Pour pouvoir gérer cette mémoire et franchir le mur des 4Go imposé par les adresses 32 bits, les architectes ont augmenté leur taille à 64 bits. Cela permet d'utiliser plusieurs exaoctets de mémoire. Les contreparties sont une augmentation du budget énergétique dû à l'agrandissement des composants matériels pour supporter les adresses 64 bits et une augmentation de l'empreinte mémoire des applications par l'utilisation des pointeurs de cette taille.
Pour autant, l'utilisation des cœurs 64 bits n'est pas une fatalité. L'architecture TSAR [7] conçue au LIP6 en est un bon exemple. Cette architecture NUMA massivement multicœurs à mémoire partagée cohérente est composée de plus de 1000 cœurs 32 bits répartis en clusters, et offre 1To de mémoire. Chaque cluster est composé de 4 cœurs, d'un cache L2 partagé et gère un banc mémoire de 4Go. TSAR a permis de montrer que des processeurs 32 bits sont capables de gérer efficacement une telle quantité de mémoire physique.

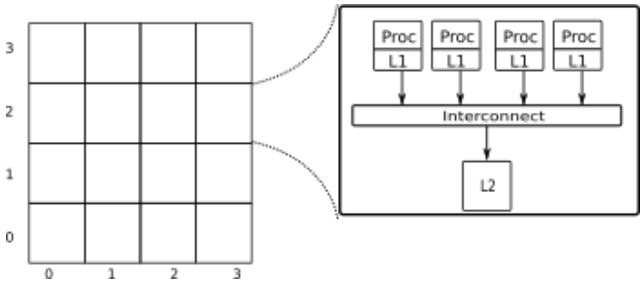


FIGURE 1 – Schéma de l'architecture TSAR composée de 4x4 clusters. Chaque cluster contient 4 cœurs, un cache L2 partagé et un accès à un banc mémoire de 4Go

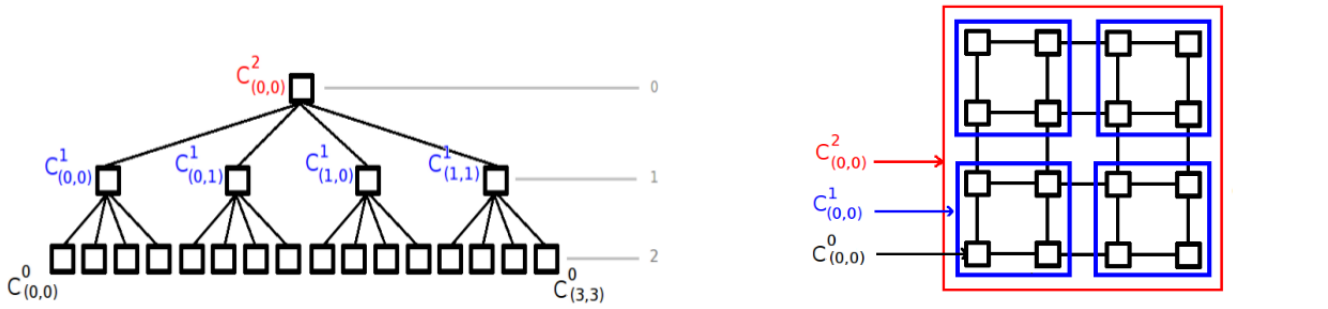
Afin de profiter de ce type d'architecture et d'avoir de bonnes performances, les systèmes d'exploitation doivent s'adapter. C'est le but du noyau monolithique ALMOS (*Advanced Locality Management Operating System*) [2] développé lors de la thèse de G. Almaless, puis transformé en multi-noyau [4] : ALMOS-MK [12] (*ALMOS-MultiKernel*). L'architecture TSAR étant clusterisée, il a été décidé de placer un noyau complet dans chaque cluster. Les noyaux sont indépendants et ne partagent aucune ressource. Toutefois, l'union des noyaux permet de gèrer toutes les ressources. Enfin, les communications entre les noyaux sont assurées par un système de passage de messages matériel basé sur le principe des Remote Procedure Call (RPC). Ces RPC sont stockés dans des boîtes aux lettres qui sont lues périodiquement, ou peuvent être exécutés immédiatement via une interruption en fonction de leur priorité.
Dans un contexte où la gestion des ressources, et notamment la mémoire, est distribuée, il faut assurer un service de migration de tâches entre les différents clusters ; soit pour des raisons de parallélisme (applications haute performance), soit pour des raisons d'équilibrage de charge. Dans cet article, nous décrivons un mécanisme de placement de processus à la création, et non lors de l'exécution. L'équilibrage de charge à la volée est la prochaine étape de ce travail.
Le reste de cet article est organisé comme suit : la section 2 présente les problèmes liés au déplacement des processus et les solutions mises en place pour résoudre ces problèmes ; la section 3 présente les évaluations expérimentales ; la section 4 présente un état de l'art sur la migration des processus, et nous concluons avec la section 5.

2. Problèmes liés au déplacement

On peut définir un processus en cours d'exécution comme un ensemble d'éléments contenant : un identifiant, un programme à exécuter, des entrées/sorties et un service de communication avec d'autres processus. Chacun de ces éléments doit être pris en compte pour la migration.

2.1. Identification

Les identifiants des processus (*PID*) sont fixes et sont attribués par le noyau à leur création. Dans un système multi-noyau indépendants, deux processus créés sur deux noyaux différents peuvent avoir le même identifiant. Une conséquence possible est d'avoir, après déplacement, deux processus portant le même identifiant et s'exécutant sur le même noyau.
Pour pallier ce problème, un système d'identification global a été mis en place. Nous avons découpé de manière statique l'ensemble des identifiants que peuvent allouer les noyaux en attribuant une plage de 1024 identifiants à chaque noyau. Ce découpage repose sur le numéro de cluster physique sur lequel s'exécute les noyaux. Il est donc possible, à partir d'un PID, de retrouver le noyau qui a délivré chaque identifiant. Avec cette solution, nous assurons une identification unique de chaque processus quelque soit son emplacement.



(a) L'arbre DQDT

(b) Découpage d'une architecture à 16 clusters physiques en clusters logiques.

FIGURE 2 – Représentation et utilisation de la DQDT

2.2. Localisation

2.2.1. Placement

Le noyau ALMOS-MK dispose d'une entité ayant connaissance de l'occupation de la plateforme (mémoire et cœurs). Cette entité s'appelle la DQDT, pour *Distributed Quaternary Decision Tree*. Cette structure est un arbre d'arité quatre. Chaque niveau de l'arbre regroupe les informations relatives à un cluster ou à un groupe de clusters. Chaque feuille de l'arbre correspond à un cluster physique. En remontant dans l'arbre, chaque nœud correspond à un groupe de 4, 16, 64 ou 256 clusters. Pour améliorer les temps d'accès, les nœuds de l'arbre sont distribués sur les bancs de mémoire, et la mise à jour se fait sans verrou. La figure 2 donne une vue d'ensemble de la structure et de son utilisation.

Nous nous basons sur ce composant pour déterminer l'emplacement où seront migrées les tâches. Lors de la création d'une tâche par un noyau (appel système `fork()`), la DQDT est interrogée pour connaître l'emplacement futur de la tâche. Si cet emplacement est un noyau distant, il faut interroger ce dernier pour obtenir un PID. Cette demande se fait via le mécanisme de passage de messages disponible dans ALMOS-MK. Le noyau qui a délivré l'identifiant devient le *noyau ancre* de la tâche et enregistre l'identité du demandeur (le numéro de cluster physique émetteur de la requête). Une fois l'identifiant obtenu, la création se termine et le processus reste sur le *noyau créateur*. Il sera déplacé une seule fois, lors son lancement, et sera toujours envoyé vers son noyau ancre.

2.2.2. Déplacement

La création d'un processus est coûteuse pour un noyau. Elle nécessite plusieurs étapes et initialise de nombreuses données. Or, une partie de celles-ci sera effacée et remplacée lors du lancement de la tâche (appel système `exec()`). Afin d'éviter de migrer des données qui seront effacées plus tard, nous avons choisi de déplacer les processus à leur lancement et non à leur création. Dans le cas où le processus ne ferait jamais appel à `exec()`, ce dernier restera sur son noyau créateur jusqu'à sa terminaison.

Le déplacement se fait en utilisant le mécanisme de passage de messages. De la même manière qu'un `exec()` classique, on envoie au noyau cible le PID de la tâche à lancer ainsi que le chemin vers le binaire à exécuter. Sur la base de ces informations, le noyau cible créé un processus pour exécuter le programme, tandis que le noyau créateur supprime l'ancien processus de sa mémoire.

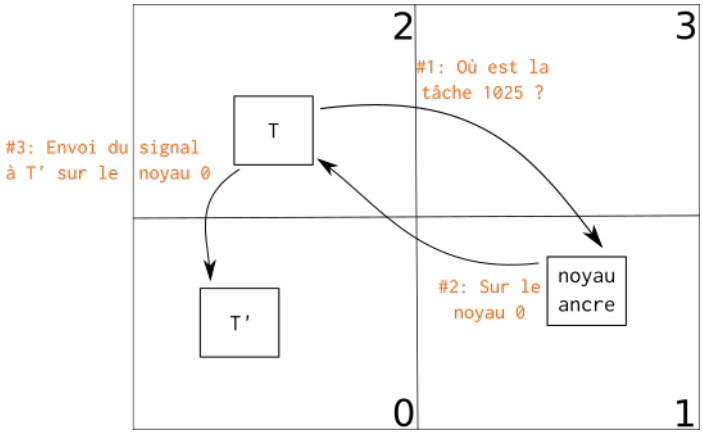


FIGURE 3 – Protocole de résolution de lieu. La tâche T (client) veut envoyer un signal à la tache T′. Celle-ci se trouve sur le cluster 0, mais son noyau ancre (serveur) est situé sur le cluster 1.

2.3. Communication

Les processus doivent pouvoir communiquer entre eux pour assurer le bon fonctionnement du système. Les signaux POSIX [9] comme le `SIGCHLD`, utilisé lors de la terminaison d'un processus, sont un bon exemple. Le respect de cette norme a été choisi pour ALMOS et doit être maintenu dans ALMOS-MK.

Le fait d'avoir des processus en mouvement sur la plateforme nécessite de connaitre leur localisation avant l'envoi d'un message. Or, nous avons vu en section 2.1 qu'un processus n'est pas toujours sur son noyau ancre ; il peut être sur son noyau créateur. Néanmoins, le noyau ancre connaît le noyau créateur puisqu'il a enregistré son identité lors de la requête de PID.

En utilisant cette propriété, nous avons construit un protocole de résolution de lieu représenté par la figure 3. Le noyau ancre agit comme le serveur. Le client est une tâche T voulant envoyer un signal à une tâche T′. Comme le client connaît le PID de T′, il peut retrouver l'identifiant du noyau ancre jouant le rôle du serveur. T contacte alors le noyau ancre en lui demandant où se trouve la tâche T′. Deux réponses sont alors possibles : soit *i)* la tâche est sur le noyau ancre (lui-même) soit *ii)* elle est sur le noyau créateur dont le serveur connaît l'identifiant. La réponse est donnée à T, qui l'utilise pour envoyer son message.

En nous basant sur ce protocole, nous avons ré-implémenté le gestionnaire de signaux d'ALMOS-MK. Néanmoins, ce protocole peut être utilisé pour mettre en place des opérations de communication autres que les signaux.

2.4. Gestion des entrées/sorties

ALMOS-MK est un système de type UNIX. Dans ces systèmes, les périphériques, le réseau, les disques durs, etc, sont abstraits par la notion de fichier. Toutes les entrées/sorties utilisent ce mécanisme. Lorsqu'un processus ouvre un fichier, celui-ci est placé dans la mémoire du noyau où s'exécute le processus. Dans un contexte multi-noyau avec des espaces mémoire disincts, on perd l'accès au fichier lors de la migration.

Le système de gestion de fichiers d'ALMOS-MK a alors été modifié pour tenir compte de la localité des fichiers. Un fichier est vu comme étant composé de deux parties, une locale et une distante. La partie locale est propre à chaque processus. On y trouve des informations comme l'heure d'ouverture du fichier ou le mode (lecture-seule, lecture-écriture). Ces informations peuvent être facilement déplacées en même temps que le processus. La partie distante est partagée par tous les processus ayant ouvert un même fichier. Celle-ci contient notamment le système de synchronisation des accès qui empêche d'avoir des incohérences lors d'une lecture

et d'une écriture simultanées par deux processus différents. Cette partie distante est toujours dans le même banc mémoire et ne peut être déplacée.
De plus, dans les systèmes UNIX, il existe des fichiers partagés par toutes les tâches, comme l'entrée standard, la sortie standard et la sortie d'erreur. Ces fichiers sont tous ouverts à l'initialisation de la machine par le noyau maître, le noyau 0. Si un de ces fichiers est utilisé simultanément par toutes les tâches, il devient un goulot d'étranglement. Pour résoudre ce second problème, la gestion des fichiers a été répartie sur la plateforme. Lors de l'initialisation de la machine, on fixe pour chaque fichier du disque dur le banc mémoire où il sera placé lors de son ouverture. Les fichiers sont ainsi répartis sur toute la mémoire et sont gérés par différents noyaux. De fait, l'accès aux fichiers partagés n'engendre plus de problèmes de contention puisqu'ils ne sont plus tous situés sur le noyau maître.

3. Evaluation expérimentale

Nous avons considéré quatre expériences pour évaluer notre solution. La première consiste à mesurer le temps de transport d'un message entre deux noyaux. Les résultats seront utilisés pour analyser les solutions mises en place qui reposent toutes sur ce mécanisme. Les trois autres expériences consistent à mesurer *i)* le temps d'exécution de l'appel système `kill()` utilisé pour l'envoi de signaux, *ii)* le temps d'exécution de l'appel système `fork()` lors la création d'un processus local et distant et *iii)* le temps d'exécution de l'appel système `exec()` lors d'un lancement local et distant.
Nous avons considéré le pire cas d'exécution lors de nos expériences. Lors de la création d'un processus distant, le noyau qui attribue le PID du processus en cours de création sera toujours le noyau le plus éloigné physiquement du noyau créateur. Les expériences ont été réalisées avec un modèle SystemC [10] de l'architecture TSAR. Ce modèle est exécuté par le simulateur SystemCASS [5] respectant le modèle CABA (*Cycle-Accurate/Bit-Accurate*). Nous avons introduit des macros dans le code du noyau au début et à la fin des fonctions à mesurer. Ces macros sont interprétées par le simulateur lors de l'exécution et nous permettre de compter le nombre de cycles nécessaire à l'exécution de ces fonctions.

3.1. Passage de messages

Les résultats de la figure 4a montrent que les messages coûtent entre 3 000 et 12 000 cycles, avec un coût moyen de 7 000 cycles. Cela représente uniquement le temps de transport d'un noyau à un autre. Les coûts addionnels liés au traitement ne sont pas pris en compte. On note une augmentation à partir de 12 clusters. On retrouve cette tendance dans les expériences suivantes. Cette observation est actuellement non-expliquée et fait l'objet d'une étude plus approfondie.

3.2. Envoi d'un signal

Nous avons créé deux procesus sur le même noyau et l'un envoie un signal de terminaison à l'autre (`SIGKILL`). On obtient un coût moyen de 2 000 cycles. Dans le cas d'un processus distant, la tâche qui reçoit le signal est sur le noyau le plus éloigné. En moyenne, cette opération prend 17 000 cycles en comptant le passage de message. Nous on avons vu en section 3.1 qu'un passage de messages coûte en moyenne 7 000 cycles, le coût réel des signaux distants est donc en moyenne de 10 000 cycles.

3.3. Création d'un processus distant

Pour cette seconde expérience, nous avons d'abord mesuré le nombre de cycles nécessaire à la création d'un processus local. Nous avons désactivé la DQDT et tous les processus sont

créés par le noyau maître. Nous avons ensuite rejoué l'expérience en nous placant dans le pire cas d'exécution défini précédemment. Le coût total de l'opération est en moyenne de 160 000 cycles, contre 144 000 dans le cas d'une tâche locale (figure 4c). On paye un surcoût moyen de 20 000 cycles pour créer un processus distant. Parmis ces 20 000 cycles, 7 000 en moyenne sont nécessaires pour obtenir l'identifiant de la tâche créée. Le coût effectif est en moyenne de 13 000 cycles.

3.4. Déplacement d'un processus

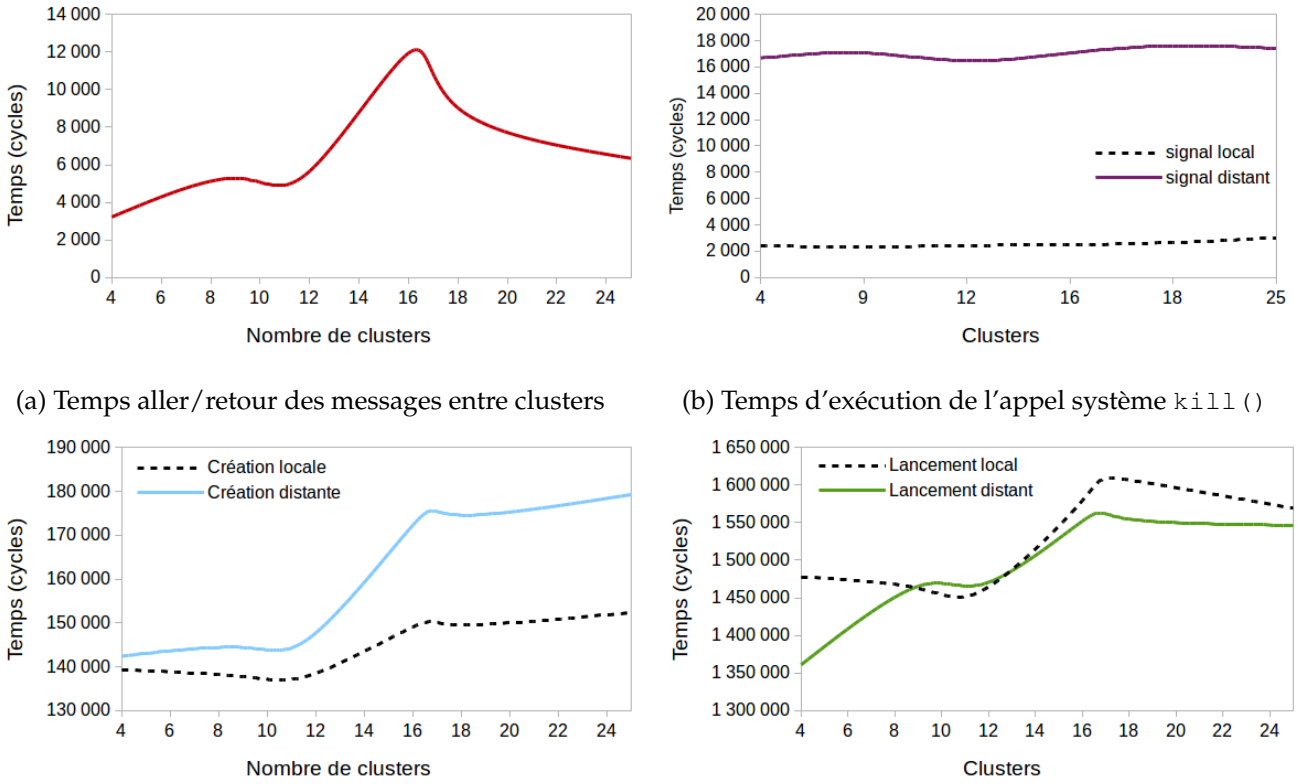
Pour cette troisième expérience, nous avons mesuré le nombre de cycles nécessaire au lancement d'un processus sur son noyau créateur. Nous avons ensuite fait la même chose lorsque celui-ci est lancé sur un autre noyau, distant. Les résultats de la figure 4d montrent qu'à partir de 12 clusters (48 cœurs), la migration d'un processus est plus rapide qu'un lancement sur son noyau créateur.
En effet, lors d'un lancement local, le noyau créateur doit d'abord effacer une partie des informations initialisées précédemment pour ensuite effectuer le lancement. Or, dans le cas du déplacement, le noyau créateur envoie le message au noyau destinaire qui va lancer la tâche pendant que le créateur supprime les informations initiales. Les opérations mémoires étant onéreuses, on voit ici l'avantage de les effectuer en parallèle, et donc de déplacer la tâche lors de son lancement.
En moyenne, l'opération prend 26 000 cycles de moins, auxquels nous pouvons retirer les 7000 cycles dûs aux communications (voir 3.3), soit un gain de 19 000 cycles. En tenant compte du surcoût payé lors de la création, on économise en moyenne 6 000 cycles lorsque l'on déplace une tâche d'un noyau à un autre.

3.5. Performances globales

En moyenne, l'opération de création et de déplacement prend 26 000 cycles de moins, auxquels nous pouvons retirer les 7000 cycles dûs aux communications. Nos expériences montrent un maintien des performances et souvent un gain lors du déplacement des tâches sur la plateforme. On voit également que le passage à l'échelle est assuré puisque jusqu'à 96 cœurs, l'opération de déplacement est plus rapide qu'un lancement local. Ces expériences montrent que nous avons réussi à mettre en place un service de déplacement de tâches entre noyaux ayant un coût minimal. De plus, ce service passe à l'échelle sur l'architecture TSAR.

4. Etat de l'art

Sprite [6] est un système d'exploitation distribué développé au début des années 1990. Ce dernier est réparti sur plusieurs machines, et assure un service de migration de processus totalement transparent pour l'utilisateur. Une partie de notre implémentation est directement inspirée de ces travaux. Néanmoins, cette solution n'a pas été développée dans un contexte multi-noyau puisque chaque système est en exécution sur une machine physique distincte. Ici, nous visons une seule machine physique, clusterisée, avec un seul espace d'adressage partagé par tous les cœurs. Schüpbach *et al.* [13] sont les premiers à avoir adressé cette problématique dans un contexte multi-noyau. La migration de processus est prise en compte par le système, mais le service ne permet pas le partage des descripteurs de fichiers, ce qui limite le déplacement puisque par défaut, tous les processus partagent au moins le fichier de sortie standard. PopCorn Linux [3, 11] est un noyau Linux transformé en multi-noyau. Ce dernier supporte uniquement la création distante de threads. Enfin, il n'offre pas de système de fichiers partagés entre les noyaux. Plus récemment, c'est le multi-noyau Hare [8] qui a été proposé. Il cible les



(a) Temps aller/retour des messages entre clusters

(b) Temps d'exécution de l'appel système `kill()`

(c) Temps d'exécution de l'appel système `fork()`

(d) Temps d'exécution de l'appel système `exec()`

FIGURE 4 – Évaluation des appels systèmes modifiés pour la migration de processus. Les mesures sont données en cycles en fonction du nombre de clusters. Chaque cluster comporte 4 cœurs.

architectures manycore, permet de déplacer des processus et maintient cohérent les descripteurs de fichiers partagés. De plus, il respecte partiellement la norme POSIX. Néanmoins, cette solution ne gère pas la délivrance des signaux à travers le système. De plus, l'implémentation n'est pas basée sur un noyau monolithique, et nous souhaitons conserver ce modèle.

5. Conclusion

Les architectures 64 bits semblent être la suite logique du développement matériel. Néanmoins, l'architecture TSAR montrent que les processeurs 32 bits peuvent être capables d'offrir la même puissance tout en conservant les avantages de la basse consommation et de la faible quantité de silicium requise à leur fabrication. De plus, TSAR montre que le passage à l'échelle est possible pour de tels processeurs. Nous devons à présent adapter les systèmes pour profiter de ces évolutions matérielles. Le noyau ALMOS-MK est un des exemples de travaux allant dans cette direction. Bien que spécifique à l'architecture TSAR, il tend à montrer que nous pouvons construire des architectures 32 bits performantes et les exploiter pleinement.
Ces travaux sont toujours en cours et d'autres services comme le déplacement à la volée (pendant l'exécution du processus) vont être implémentés. Néanmoins, ce prototype est suffisant pour montrer que l'on peut gérer une grande quantité de mémoire en utilisant une architecture 32 bits et un noyau développé à cet effet en maintenant les performances et le budget énergétique.

Bibliographie

1. Agarwal (V.), Hrishikesh (M.), Keckler (S. W.) et Burger (D.). – *Clock rate versus IPC : The end of the road for conventional microarchitectures*. – ACM, 2000volume 28.
2. Almaless (G.). – *Operating System Design and Implementation for Single-Chip cc-NUMA Many-Core*. – PhD Thesis, Université Pierre et Marie Curie, 2014.
3. Barbalace (A.), Ravindran (B.) et Katz (D.). – Popcorn : a replicated-kernel OS based on Linux. – In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.
4. Baumann (A.), Barham (P.), Dagand (P.-E.), Harris (T.), Isaacs (R.), Peter (S.), Roscoe (T.), Schüpbach (A.) et Singhania (A.). – The multikernel : a new os architecture for scalable multicore systems. – In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 29–44. ACM, 2009.
5. Buchmann (R.) et Greiner (A.). – A fully static scheduling approach for fast cycle accurate systemc simulation of mpsocs. – In *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, pp. 101–104. IEEE, 2007.
6. Douglis (F.) et Ousterhout (J.). – Transparent process migration : Design alternatives and the sprite implementation. *Software : Practice and Experience*, vol. 21, n8, 1991, pp. 757–785.
7. Greiner (A.). – TSAR : a scalable, shared memory, many-cores architecture with global cache coherence. – In *9th International Forum on Embedded MPSoC and Multicore (MPSoC'09)*volume 15, 2009.
8. Gruenwald III (C.). – *Providing a Shared File System in the Hare POSIX Multikernel*. – PhD Thesis, Massachusetts Institute of Technology, 2014.
9. IEEE. – *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. – Institute of Electrical and Electronics Engineers, 2001, xliv + 448p. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992, Open Group Technical Standard Base Specifications, Issue 6.
10. Initiative (O. S.). – Functional specification for SystemC 2.0. *Outubro*, 2001.
11. Katz (D. G.). – *Popcorn Linux : Cross Kernel Process and Thread Migration in a Linux-Based Multikernel*. – PhD Thesis, Virginia Tech, 2014.
12. Lamine Karaoui (M.). – *Système de fichiers scalable pour architectures many-cores à faible empreinte énergétique*. – PhD Thesis, Université Pierre et Marie Curie, 2016. Under review.
13. Schüpbach (A.), Peter (S.), Baumann (A.), Roscoe (T.), Barham (P.), Harris (T.) et Isaacs (R.). – Embracing diversity in the Barrelfish manycore operating system. – In *Proceedings of the Workshop on Managed Many-Core Systems*, p. 27, 2008.