Subutai: Implantation de primitives de synchronisation au sein d'interfaces NoCs sans modification du code source

Rodrigo Cataldo, Kevin Martin, Jean-Philippe Diguet

Subutai: Implantation de primitives de synchronisation au sein d'interfaces NoCs sans modification du code source

Rodrigo Cataldo[1,2], Kevin J.M. Martin[1], Jean-Philippe Diguet[1]
[1]Univ. Bretagne-Sud, CNRS UMR 6285, Lab-STICC, Lorient - [2]PUCRS, Porto Alegre, Brésil.

I. INTRODUCTION

Les applications parallèles profitent de la généralisation des architectures multiprocesseurs (MPSoCs) et le mouvement amplifie la demande de puissance de calcul conduisant à un nombre croissant de processeurs. Ces architectures requièrent des réseaux de communications (NoC) dont le rôle devient clef. Si les NoCs offrent une bande passante importante, ils induisent également une latence qui en augmentant pénalisent les opérations séquentielles. Parmi ces opérations nous trouvons les primitives de synchronisation qui impactent très significativement les performances des applications parallèles qui ne peuvent en faire l'économie si elles font appel à des structures déterministes telles que les queues et les exclusions mutuelles [1]. La contrainte peut être relâchée mais au prix de réécritures du code. La plupart des travaux récents requièrent de telles modifications du code qui sont identifiées comme extrêmement coûteuses [2][3] et donc peu exploitables.

Nous avons donc développé Subutai, une solution logicielle et matérielle qui traite spécifiquement la question de la pénalité introduite par le traitement des synchronisations au sein d'architectures multiprocesseurs à base de NoC. La solution proposée repose sur l'externalisation des services de synchronisation au sein des interfaces du NoC (NI) dont l'architecture matérielle est augmentée de nouvelles capacités de traitement à l'aide de ressources matérielles dédiées. Notre approche étend donc les services la NI en implantant de manière distribuée les primitives *mutex*, *barrier* et *condition*. De plus nous proposons des nouveaux types de paquets pour transporter les requêtes de synchronisation. Enfin nous avons développé des API (*Application Programming Interface*) qui remplacent les API existantes avec une interface inchangée afin de ne pas contraindre les concepteurs à changer le code originel des applications. Le code et l'OS, qui dispose d'un nouveau driver, restant inchangés une simple recompilation est requise. La **Figure 1** montre comment notre solution interagit avec son environnement. Nous avons démontré l'intérêt de notre solution à l'aide du simulateur GEM5 et d'applications du benchmark PARSEC (*streamcluster* et *bodytrack*). Nous utilisons les traces produites pour alimenter notre propre simulateur SystemC conçu produire des résultats dans un temps raisonnable. Ce travail a abouti à 5 contributions (i) une solution pour accélérer les applications parallèles sans modification du code; (ii) une architecture de NI avec des ressources dédiées pour le traitement rapide des primitives de synchronisation tout en demeurant compatible avec l'usage classique de NoC et en disposant d'une mémoire locale; (iii) un jeu d'API pour calcul parallèle; (iv) un simulateur basé sur des traces pour accélérer la simulation d'applications réelles; et (v) la synthèse de l'architecture de la NI avec une technologie 28 nm FD-SOI.

II. ETAT DE L'ART

Le Tableau 1 positionne nos travaux vis à vis de l'état de l'art. À notre connaissance il s'agit de la première solution d'implantation matérielle et distribuée des primitives de synchronisation qui permette de conserver le code des applications inchangé.

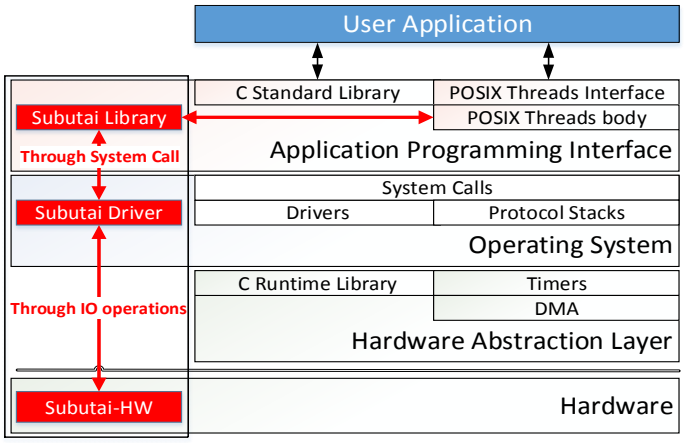

Figure 1. Subutai: library, driver, hardware

TABLE I. TABLEAU 1. COMPARAISON AVEC L'ETAT DE L'ART.

Work	HW/SW solution	Legacy code	Primitive	Application
[4]	HW	Not addressed	Fault-tolerant barrier	Synthetic
[5]	HW	Not addressed	Barrier	Synthetic
[6]	HW	Not addressed	Locking	Synthetic
[3]	Both	Changed	Atomic operations	Synthetic
[7]	Both	Changed	Atomic operations	STAMP suite + Synthetic
[8]	SW	Changed	Locking	Synthetic + Mandelbrot
[9]	SW	Changed	Locking	OS Kernel + Synthetic
[10]	HW	Unchanged	FIFO events	MPEG4-SP
This	**Both**	**Unchanged**	**Locking, Barrier, and Condition**	**Synthetic + PARSEC Benchmark**

III. ARCHITECTURE MPSOC ET NOC ENRICHI

La Figure 2 décrit l'architecture matérielle de Subutai. Chaque cœur comporte un niveau de cache L1 dédié (instructions, données) et un niveau L2. Il communique avec son périphérique NI qui lui donne accès au NoC et au-delà aux autres NI et donc aux cœurs. La Figure 2 montre aussi les modifications apportées à la NI classique. L'architecture matérielle implante des listes chainées pour traiter rapidement les opérations de gestions des queues des différentes primitives (ex. Mutex). Dans la version actuelle, 1 KiB de mémoire (SPM) est intégrée pour la gestion des primitives. Il est important de noter que ces mémoires locales libèrent de fait les caches qui n'ont pas prendre en charge ces données de synchronisation. Nous utilisons également un *scheduler* local qui évite l'utilisation d'un OS distribué et ses conséquences sur les pénalités induites au niveau des caches. Cette réplication de l'OS permet d'éviter notamment d'introduire dans l'ordonnancement des accès couteux aux mémoires [11], ce qui est crucial pour notre solution.

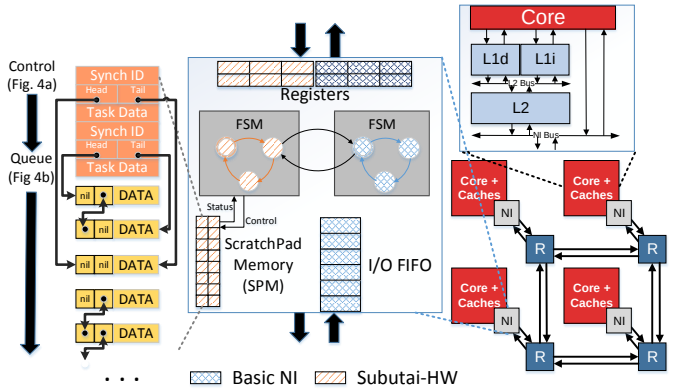

Figure 2. Architecture MPSoC basée NoC et Subutai

Les résultats de synthèse de la NI, avec une technologie 28nm SOI pour la logique et 32nm pour la mémoire (CACTI), montrent un surcoût de 36% mais qui inclut la mémoire locale laquelle représente 17% et libère la mémoire principale. Ramené à la surface d'une architecture à 64 processeurs de 400mm2 [3], le surcoût se limite à 0.03%.

IV. API ET COMMUNICATIONS HW/SW.

Comme preuve de concept nous avons choisi d'implanter la librairie POSIX Threads (PThreads) qui constitue une des interfaces les plus courantes dans le domaine des applications parallèles. La Figure 3 montre le processus de communication de notre solution.

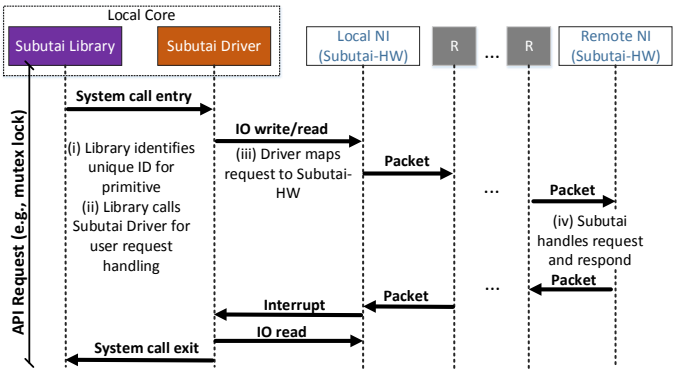

Figure 3. Protocole de communication de Subutai

V. RESULTATS ET CONCLUSION

La Figure 4 montre les 4 étapes de la méthode suivie pour obtenir les résultats de simulation en considérant un compromis entre temps de simulation et précision. La Figure 5 présente les accélérations significatives (x1.71 à x2.71) obtenues pour les benchmarks (i) Bodytrack (computer-vision) et (ii) Streamcluster (data mining).

VI. CONCLUSIONS

Subutai est une solution logicielle/matérielle pour l'accélération des applications parallèles sans modification du code source. Elle repose sur une architecture de NI augmentée avec une gestion mémoire particulièrement optimisée. Elle inclut un driver et une implantation de la bibliothèque POSIX *Pthreads*. Les résultats montrent des accélérations significatives sur les temps de synchronisation en comparaison de l'implantation logicielle classique avec un surcoût acceptable au regard du coût d'une architecture multiprocesseurs.

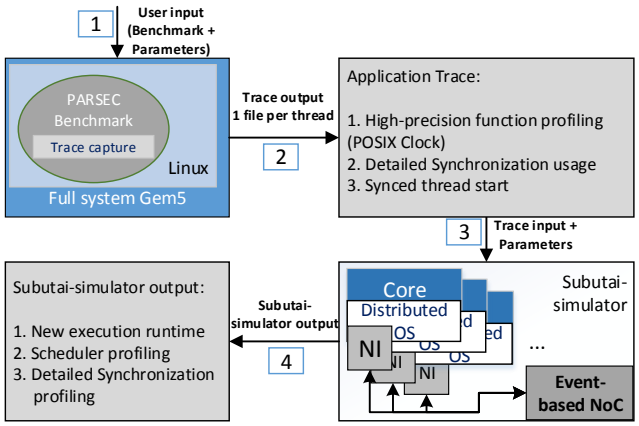

Figure 4. Méthode de Validation

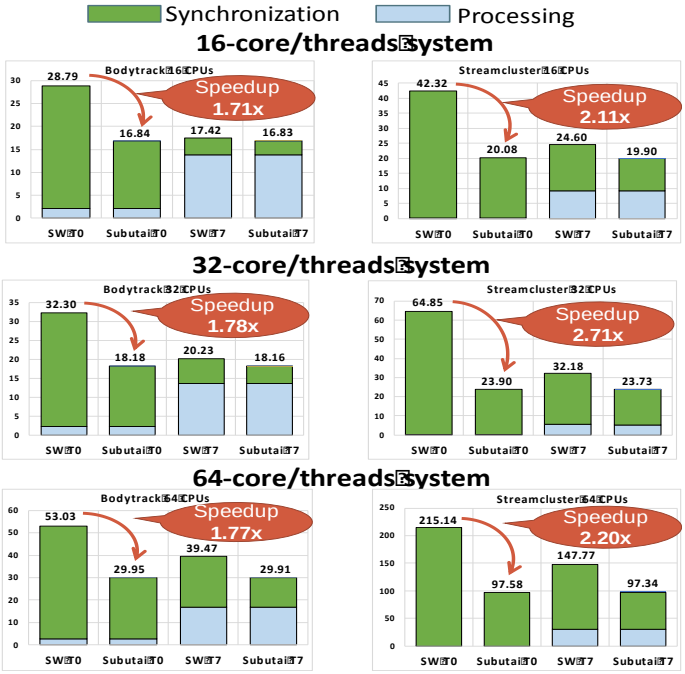

Figure 5. Accélérations des applications Bodytrack et Streamcluster

REFERENCES

[1] H. Attiya et al. "Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated". *POPL*, v. 46 n. 1, 2011.

[2] P. McKenney et al. "Introducing Technology into the Linux Kernel: A Case Study", SIGOPS Oper. Syst. Rev, v.42, n.5, pp 4-17, 2008.

[3] S. Patel et al. "A Hardware Implementation of the MCAS Synchronization Primitive". DATE, pp. 918-921, 2017.

[4] R. Sivaram; et al. "A Reliable Hardware Barrier Synchronization Scheme". *Int. Parallel Processing Symp. (IPPS)*, pp. 274-280, 1997.

[5] Abellán et al. "Design of a Collective Communication Infrastructure for Barrier Synchronization in Cluster-Based Nanoscale MPSoCs". *DATE*, pp. 491-496, 2012.

[6] C. Stoif et al. "Hardware Synchronization for Embedded Multi-Core Processors". *ISCAS*, pp. 2557-2560, 2011.

[7] N. Diegues et al. "Virtues and Limitations of Commodity Hardware Transactional Memory". *Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pp. 3-14, 2014.

[8] C. Kirsch et al. "Fast and Scalable, Lock-Free k-FIFO Queues". *Int. Conf. on Parallel Computing Technologies (PaCT)*, 2013.

[9] M. Desnoyers et al. "User-Level Implementations of Read-Copy Update". *IEEE Trans. Parallel Distrib. Syst.*, v. 23 n. 2, pp. 375-382, 2011.

[10] K. Martin et al. "Notifying Memories: a case-study on Data-Flow Applications with NoC Interfaces Implementation". *DAC*, 2016.

[11] D. Howells et al. "Linux Kernel Memory Barriers".Available at: www.kernel.org/doc/Documentation/memory-barriers.txt. 2017.