Analysis of sharing patterns in multicore applications and new instructions to accelerate such patterns

Français (english below)

Mots-clés : *processeurs multicoeurs, cohérence de caches, RISC-V*

Contexte : Afin d'améliorer la performance, un processeur généraliste associe à chacun de ses coeurs une mémoire cache privée (utilisable uniquement par ce coeur), qui va garder un sous-ensemble des données utilisées proche des unités d'exécution afin d'en accélérer l'accès. La puce possède de plus un cache de plus grande taille, partagé par tous les coeurs. Cette architecture est illustrée dans la Figure 1.

Afin de faciliter le développement d'applications parallèle, les différentes mémoires caches situées sur la puce assurent la cohérence des données, c'est à dire qu'un mot mémoire peut résider dans plusieurs caches privés uniquement si les processeurs ne font que lire la donnée. Toute écriture entraîne l'invalidation de la donnée dans les caches privés la possédant (excepté l'écrivain), et un cache ayant perdu sa copie suite à une écriture devra recharger la nouvelle version de la donnée. Dans les processeurs généralistes, la gestion de la cohérence est faite par le matériel, et est complètement transparente au programmeur. Dans le cas contraire, le logiciel devrait explicitement gérer la cohérence des données ne serait-ce que pour obtenir un résultat correct, ce qui ralentirai significativement le développement d'applications multithreadées.

Cependant, le matériel gérant la cohérence est obligé de faire certains choix sous-optimaux car il n'a pas une vision globale des motifs d'accès et de partage des données. Par exemple, lors d'un chargement depuis la mémoire, un processeur peut demander à charger une donnée depuis la mémoire et l'insérer dans son cache privé avec uniquement la permission de lire la donnée, ou avec la permission de la lire et de l'écrire. Suivant le motif d'accès et de partage, la bonne décision (du point de vue de la performance et du traffic sur la puce) n'est pas toujours la même. Si la donnée n'est pas partagée, le processeur a tout intérêt à obtenir la permission d'écrire même si il ne fait que lire la donnée, afin d'éviter d'envoyer une seconde requête dans le futur demandant la permission d'écrire sur la ligne déjà présente en lecture seule. Si la donnée est partagée et que le processeur ne va pas l'écrire, alors il vaut mieux au contraire ne pas demander la permission d'écrire, car cela impliquerait d'invalider les autres copies présentes dans le système (i.e., dans les caches privés des autres processeurs). De plus, de manière générale, suivant l'intervalle de temps entre la lecture d'une donnée et son écriture par un processeur, il peut-être soit plus intéressant d'obtenir la permission d'écrire au plus tôt (à la première lecture) afin de ne pas ralentir l'écriture si elle est proche, ou au plus tard (lors de l'écriture) afin de permettre aux autres lecteurs de garder leur copie aussi longtemps que possible.

Ces motifs d'accès et de partage sont cependant connu par la personne ayant conçu le programme. Il serait donc intéressant de pouvoir exprimer ces motifs afin que le matériel soit en capacité de prendre les bonnes décisions lors de l'exécution, plutôt que de tenter de deviner quelle est la bonne décision. Le jeu d'instruction RISC-V étant ouvert et facilement extensible, il nous donne une opportunité d'étudier cette possibilité, via l'ajout d'instructions indiquant avec quel motif une donnée est accédée ou partagée.

Objectifs : Le stage s'articule en trois temps. Premièrement, il conviendra de faire une revue de l'état de l'art portant sur les différents motifs de partages observés dans les programmes multithreads et les heuristique matériels permettant de les identifier (notamment leur précision). Cela permettra à l'étudiant ou l'étudiante de formuler les motifs que l'on souhaiterai exprimer via des instructions dédiées. Dans un second temps, il conviendra de quantifier la fréquence de tels

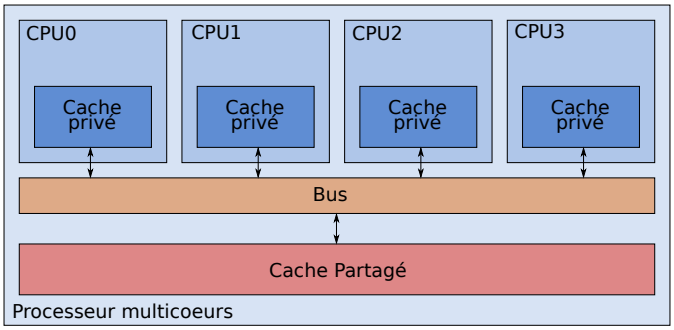

FIGURE 1 – Processeur multicoeur avec 4 coeurs, chacun possédant sa mémoire cache privée, ainsi qu'une mémoire cache partagée.

motifs dans des programmes multithreadés classiques (suite de benchmarks PARSEC [1]), afin d'une part de raffiner les motifs à considérer et d'autre part d'identifier lesquels prioriser. Enfin, il conviendra d'estimer le gain en performance que pourrait apporter l'ajout de telles instructions en i) Ajoutant le support de nouvelles instructions dans gcc (via intrinsic) ii) Ajout des instructions dans les benchmarks PARSEC et iii) Simulation des benchmarks PARSEC sur un modèle de processeur généraliste multicoeur auquel l'étudiant ou l'étudiante aura ajouté le support pour les nouvelles instructions non seulement dans le processeur mais dans le bloc matériel responsable de la gestion de la cohérence. Pour cette dernière étape, on se place dans un simulateur haut niveau (gem5 [2]) et il ne sera donc **pas** question de développer du matériel en VHDL ou Verilog.

Compétences attendues/mises en œuvre :

- Maîtrise des concepts de synchronisation pour la programmation parallèle (threads, verrous...)
- Des bases en C++
- Maîtrise d' (au moins) un langage de scripting (*e.g.*, Bash, Python)
- Notion de jeu d'instruction (des notions de RISC-V sont bienvenues)
- Des notions sur les protocoles de cohérence sont un plus

English

Keywords : *Multicore, cache coherency, RISC-V*

Context : To improve performance, a general purpose processor associates a private cache memory to each of its cores, in order to keep a subset of data close to the execution units of the core and thereby accelerate data access. The chip also features a larger cache shared by all cores. This architecture is depicted in Figure 1.

To facilitate the development of parallel applications, the various cache memories implemented on chip provide data coherency. That is, a given memory address may be cached in several private caches only if the associated cores are only reading the data. Any write to the data requires invalidating all existing copies in the private caches (except for that of the writer). A cache that lost its copy following a write will have to reload the new version of the data (e.g., from memory). In general purpose processors, coherency is handled by hardware and is completely transparent to the programmer. If that were not the case, software would have to manage coherency explicitly for the program to be correct, which would significantly slow parallel application

development down.

However, the hardware handling coherency is forced to make sub-optimal choices as it does not have a global vision of access and sharing patterns accross the system. For instance, when data is loaded from memory, a core can aks to insert it into its private cache either with read permission or both read and write permissions. Depending on the access and sharing patterns, the correct decision (for performance or chip traffic) is not always the same. If the data is not shared, the core should obtain write permission in order to avoid sending a second request asking for write permission in the future. If the data is shared but the core is only going to read it, then, the core should ask for read permission only. Asking for write permission would imply invalidating the other copies and would be wasteful in this case. In addition, generally speaking, depending on the number of cycles between when a data is read by a core and when it is written by that core, it can be more interesting to obtain the write permission early (when reading) in order to not slow down the write if it is close in time, or late (when performing the write), in order to allow other cores to keep their copies as long as possible.

Those access and sharing patterns are known by the developer. It would therefore be interesting to express those patterns in the source code to help the hardware make correct decisions at runtime, rather than just trying to guess what that decision might be. The RISC-V instruction set being open source and extensible, it provides us with an opportunity to study this technique, by adding instructions conveying with what access and sharing patterns a data is being manipulated.

Objectives : The internship is built around three items. First, reviewing the litterature on sharing patterns in multicore programs as well as hardware techniques to identify them will allow the candidate to identify patterns that we would want to express via dedicated instructions. Second, quantifying the frequency at which such patterns occur at runtime in typical multicore workloads (PARSEC [1]) will be needed, in order to confirm the usefulness of such patterns and prioritize which patterns to support. Finally, the candidate will study the performance gain that the introduction of new instructions will bring by i) Adding support for those instructions in gcc (through intrinsic) ii) Adding those instructions in the PARSEC benchmarks and iii) By simulating PARSEC benchmarks on a multicore processor model to which the candidate will have added support for the new RISC-V instructions, both in the processor core and the blocks handling cache coherency. For this last step, a high level simulator will be used (gem5 [2]), and hardware will not be developped using VHDL or Verilog.

Expected Skills :

— Understanding of synchronization concepts for parallel programming (threads, locks...)
— Basics in C++
— Proficiency of (at least) one scripting language (*e.g.*, Bash, Python)
— Understanding of concept of instruction set (knowledge of RISC-V is welcome)
— Basics on coherency protocoles are a plus

References

[1] `https://parsec.cs.princeton.edu/parsec3-doc.htm`
[2] `https://www.gem5.org/`

Location and Contact :

The internship takes place at the TIMA Lab, in the center of Grenoble in the historical premises of Grenoble INP, 46, avenue Félix Viallet. **Contacts :** Julie Dumas (Julie.Dumas@univ-grenoble-alpes.fr), Arthur Perais (Arthur.Perais@univ-grenoble-alpes.fr)