Support matériel à l'exécution multi-threads pour le processeur Mips R3000 : Micro-architecture et caractérisation

Boris BOUTILLIER   Laurent MORTIEZ   Alain GREINER

Département ASIM, laboratoire LIP6, Université Pierre et Marie CURIE
4, place Jussieu, 75252 Paris Cedex 05
Boris.Boutillier@lip6.fr, Laurent.Mortiez@lip6.fr,
Alain.Greiner@lip6.fr

RÉSUMÉ : Ce papier présente l'architecture d'un microprocesseur MIPS R3000 multi-contextes. Cette architecture fournit un support matériel aux applications parallèles de type *multi-threads*, en permettant des changements de contexte extrêmement rapides entre tâches (un ou deux cycles), de façon à masquer la latence des accès mémoire, et à éviter le gel du processeur en cas de MISS. On présente également des premiers résultats expérimentaux obtenus pour une application multi-thread réalisant le décodage d'un flux d'images JPEG.

I – Introduction

Dans le domaine des systèmes sur puce, l'évolution des technologies de fabrication permet de concevoir des architectures complexes intégrant plusieurs coeurs de processeurs et coprocesseurs. Cette évolution vers les architectures multi-processeurs fait peser de lourdes contraintes sur les mécanismes de communication, car l'accès à la mémoire partagée devient un goulot d'étranglement du système. Différentes techniques permettent d'augmenter la bande passante du traditionnel "bus système", telles que les techniques de bus pipelinés, l'utilisation de hiérarchies de bus, ou même le recours à des techniques de micro-réseaux commutés. Toutes ces solutions ont cependant pour effet d'augmenter la latence des accès à la mémoire (mesurée en nombre de cycles du processeur).

En cas de MISS sur son cache, le processeur passe de nombreux cycles inactifs, en attente du retour de la requête. Cette tendance à l'augmentation du coût des MISS est une tendance lourde qui ne peut que s'aggraver avec l'augmentation du nombre de processeurs.

Pour résoudre ce problème plusieurs solutions existent. L'une d'entre elle consiste à implémenter plusieurs contextes matériels d'exécution à l'intérieur d'un même processeur : Lorsqu'un contexte d'exécution se trouve bloqué, suite à un MISS, le processeur bascule automatiquement sur un autre contexte d'exécution. Il faut évidemment que le mécanisme de changement de contexte soit extrêmement rapide. Par ailleurs ce mécanisme suppose que l'application possède effectivement plusieurs files d'exécution parallèles. Ce type d'architecture est donc bien adapté aux applications *multi-threads*.

Dans ce papier nous présentons les principes d'une architecture multi-contextes pour le processeur MIPS R3000, en analysant tout particulièrement le rôle du contrôleur de contexte, ainsi que l'impact sur le fonctionnement des caches.

II – PRINCIPE DU MULTI-CONTEXTES

Il faut ici bien faire la distinction entre ce mécanisme, entièrement *matériel* et un mécanisme d'ordonnancement des threads réalisé par le système d'exploitation. Du point de vue du système d'exploitation, un processeur multi-threads possédant N contextes se comporte comme un système multi-processeurs symétrique possédant N processeurs distincts.

L'objectif est de masquer les temps d'accès à la mémoire en faisant basculer le processeur d'une tâche vers une autre en cas de MISS, au niveau matériel, sans intervention du logiciel. Il est donc nécessaire de sauvegarder un état complet pour chacune des tâches en cours d'exécution à l'intérieur même du processeur. La solution retenue consiste donc à répliquer tous les registres internes au processeur autant de fois qu'il y a de contextes.
Chacun des contextes peut être dans l'un des trois états suivant :

- **Actif** : C'est celui qui utilise actuellement les ressources du processeur, il ne peut y en avoir qu'un seul dans cet état par processeur.
- **Éligible** : C'est un contexte prêt à être activé.
- **Bloqué** : Ce contexte est en attente du retour d'une requête, il ne redeviendra éligible que lorsque la requête le concernant sera revenue.

Un module spécialisé, le contrôleur de contexte tient à jour l'information concernant l'état de chacun des contextes, il se sert de signaux provenant des caches et du processeur, c'est lui qui implémente l'algorithme d'arbitrage permettant de sélectionner le contexte actif. La figure 1 donne les chronogrammes de fonctionnement d'un processeur multicontexte en comparaison d'un processeur classique. On y a représenté l'état de chacun des contextes et le temps d'inactivité du processeur.

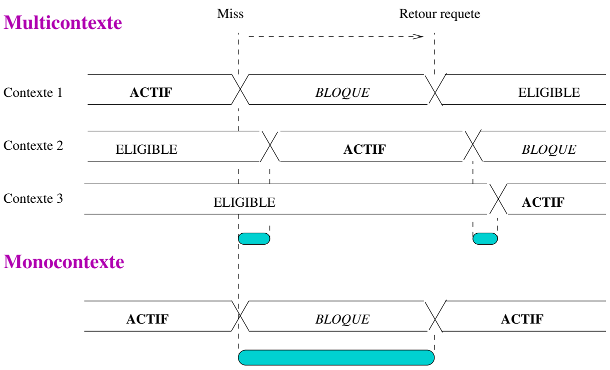

FIG. 1 – Comparaison des périodes d'inactivités

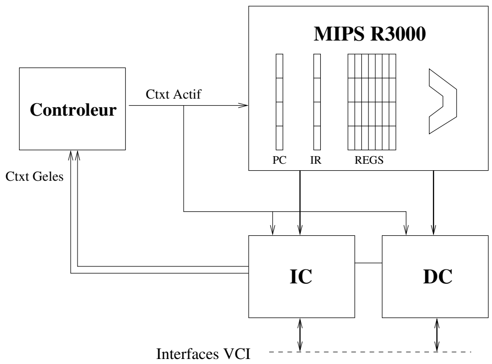

FIG. 2 – Architecture du système

III – MICRO-ARCHITECTURE

Plusieurs solutions sont possibles. Nous avons choisi une solution simple permettant un coût faible pour le changement de contexte. Il s'agit de représenter le processeur comme un ensemble de N machines à état dont une seule est active à la fois. Pour cela on remplace chaque registre du processeur par un banc de N registres. Cette approche possède l'avantage de permettre des changements de contextes en un ou deux cycles. Par contre elle est un peu coûteuse en surface de silicium, et ne peut convenir pour plus de 16 contextes.

L'architecture que nous avons modélisée se compose d'un processeur Mips R3000 multi-contextes, un cache instruction, un cache de donnée (tous deux avec une interface VCI), et un contrôleur de contexte. La figure 2 résume l'architecture de l'ensemble.

L'interface VCI nous permet d'être indépendant du réseau d'interconnexion et de pouvoir tester notre architecture dans différents environnements, bus PI, bus AMBA ou un réseau d'interconnexion à commutation de paquets comme le micro-réseau SPIN.

Le Mips multi-contextes est construit à partir d'un Mips classique pour lequel l'ensemble des registres sont remplacés par des bancs de registre adressés en lecture et en écriture par le contrôleur de contexte qui désigne le contexte courant.

Les deux caches sont construits autour de la même architecture, ce sont des caches à correspondance directe, qui utilisent une stratégie d'écriture systématique en mémoire uniquement pour le cache de données. Ces caches doivent avoir un comportement non-bloquant, c'est à dire qu'ils doivent pouvoir accepter un nouvelle requête du processeur, même après qu'une première requête de lecture ait fait MISS. En effet, suite à un MISS, le processeur n'est pas gelé, mais change de contexte d'exécution.

Les caches se composent de trois automates principaux, le premier s'occupe de l'interface avec le processeur, il récupère les adresses, et renvoie les données si elles sont présentes dans le cache. S'il est nécessaire d'accéder à la mémoire (en cas d'écriture, ou en cas de MISS en lecture), il place une requête dans une FIFO des requêtes avec le numéro du contexte auquel elle est associée. Cette FIFO joue également le rôle d'un tampon d'écritures postées. Si cette requête est bloquante (MISS en lecture), l'automate gèle le contexte en question, et cette information de gel est renvoyée vers le contrôleur de contexte. Un deuxième automate s'occupe de l'émission des requêtes sur l'interface VCI, en consommant les requêtes présentes dans la FIFO d'émission. On utilise le signal *thread_id* de l'interface VCI pour différencier les requêtes correspondant à différents contextes. Lors du retour de la réponse, un troisième automate s'occupe de la réception et remplit le buffer correspondant au *thread_id* du paquet VCI reçu. A la fin de la réception de la requête cet automate dégèle le contexte correspondant pour lui permettre de relancer l'accès à la donnée manquante. C'est lorsque ce contexte refait la demande de la donnée que le cache est mis à jour. Cela permet d'être sûr qu'au moins une donnée du paquet reçu sera consommée, et permet d'éviter des deadlocks. On a donc un buffer de réception par contexte.

Le contrôleur est le module d'arbitrage entre les contextes. Il permet de sélectionner parmi les contextes elligibles celui qui sera actif lorsque le contexte courant est gelé suite à un MISS. En principe, les changements de contexte sont donc consécutifs à un MISS. Cependant, pour éviter qu'un contexte qui ne ferait jamais MISS monopolise le processeur, il existe un compteur interne qui est réinitialisé à chaque changement de contexte, et qui permet de forcer un changement de contexte lorsqu'un nombre de cycle prédéfini s'est écoulé sans MISS, et qu'il y a au moins un autre contexte éligible.

IV – ENVIRONNEMENT DE SIMULATION

Le premier objectif de cette étude est de déterminer avec le plus de précision possible le potentiel d'une architecture multi-contextes en comparaison d'une architecture classique comportant un seul contexte, et cela dans différents environnements (bus classique de type PIbus, réseau à commutation de paquets de type SPIN), caractérisés par des latences et des bandes passantes très différentes. Il faut donc réaliser de nombreuses simulations sur différentes applications multithreads. Ces simulations nécessitent au minimum une centaine de millions de cycles d'horloge. De manière à ne pas avoir des temps de simulation exorbitants nous avons modélisé notre architecture sous forme de modèles C, qui peuvent être simulés indifféremment dans l'environnement SystèmeC, ou sous le simulateur Cass développé au LIP6. Ce simulateur a été optimisé pour simuler des modèles de niveau RTL, qui sont à la fois *cycle-true* et *bit-true*, en utilisant une méthode de simulation de type cycle par cycle.

V – PREMIERS RÉSULTATS

Pour effectuer ces simulations nous nous sommes servi d'un OS multi-processeurs/multi-threads pour Mips R3000 (MUTEK) développé au LIP6 par F.Pétrot, et nous avons exécuté une application multithreads réalisant la décompression JPEG. Cette application consiste à décoder une suite d'images au format JPEG et à les afficher sur un écran virtuel. L'application est décomposée en 7 threads réalisant chacun une partie différente du décodage, et fonctionnant comme un macro-pipeline. L'architecture simulée comporte un seul processeur MIPS multi-contextes, dont on a fait varier le nombre de contextes entre 1 et 4. Le parallélisme intrinsèque de cette application est tel que l'occupation effective correspond en moyenne à 2.5 processeurs. Cette application n'est donc pas adaptée à l'étude de processeur à plus de 4 contextes. Il s'agit donc de résultats préliminaires, qui devront être complétés par l'étude d'autres applications.

La figure 3 donne une première série de résultats qui nous permettent déjà de tirer quelques conclusions.

Tout d'abord on peut remarquer que dans le cas où on utilise un processeur simple contexte, le processeur met de 100 à 270 millions de cycles pour terminer une application qui pourrait être exécutée en 30 millions de cycles si le système mémoire était parfait (c'est à dire sans MISS). Ce résultat est amplifié par le fait que dans ces premières simulations tous les accès données sont non-cachés, pour éviter les problèmes de cohérence mémoire. On voit aussi qu'un réseau de type SPIN, à commutation de paquets ayant une latence plus grande qu'un bus, le temps d'inactivité s'en trouve fortement allongé.

Si l'on regarde les résultats obtenus pour le PIbus, on voit qu'un processeur à 2 contextes est à peu près 1,6 fois plus rapide que la version simple contexte. Par contre un processeur à 4 contextes est plus lent qu'un processeur à

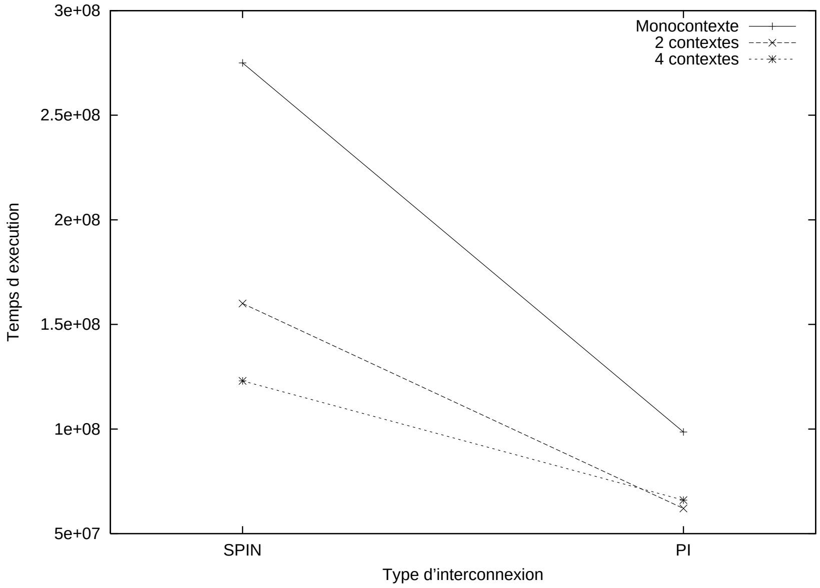

FIG. 3 – Résultats sur deux réseaux d'interconnexion différents

2 contextes. Ce résultat qui peut paraître surprenant s'explique simplement : l'application utilisée n'occupe qu'en moyenne 2.5 processeurs en même temps. Avec 4 contextes, on a donc 1.5 contextes inactifs. Mais, même en étant inactif, un contexte exécute un peu de code du système d'exploitation, dans la boucle vérifiant s'il n'y a pas de tâches en attente d'un processeur. Une partie du temps du processeur est ainsi gaspillé par ces contextes inactifs. On a donc une dégradation des performances parce que le nombre de contexte du processeur est trop élevé pour l'application.

Si on regarde les résultats obtenus pour le réseau à commutation de paquets, on voit qu'un processeur à 2 contextes est 1.75 fois plus rapide qu'un processeur simple contexte. Pour un processeur à 4 contextes, les performances sont meilleures que pour 2 contextes, car malgré l'inactivité d'une partie des contextes, les temps de latence plus long sur un réseau SPIN sont partiellement masqués par les basculements de contexte.

Globalement, on voit que l'utilisation d'un réseau à commutation de paquets n'est pas justifiée pour cette "petite" application, qui est effectivement sous-dimensionnée par rapport aux objectifs d'un réseau de type SPIN, qui a été conçu pour des systèmes comportant plusieurs dizaines de composants, alors que nous n'en utilisons que 6.

VI – CONCLUSION

Le simulateur CASS et le système d'exploitation MUTEK ont permis de construire une plate-forme d'expérimentation efficace permettant d'évaluer des applications de taille significative sur des architectures matérielles non triviales. Il se confirme que le mécanisme de commutation rapide de contexte matériel permet effectivement de compenser en partie l'augmentation du coût des MISS, particulièrement dans le cas où la nécessité d'augmenter la bande passante du bus entraîne une augmentation de la latence. Ces premiers résultats doivent cependant être analysés avec prudence, car les conditions expérimentales étaient biaisées par la non-utilisation du cache de données. De nouvelles expérimentations devront être entreprises pour des applications multi-threads plus complexes, permettant une utilisation effective des caches de données.