Savaş Ali Tokmen
M2 Pro Informatique
Option Systèmes Embarqués

Fenêtres de Registres SPARC

Introduction: les registres

La plupart des processeurs actuels sont équipés de registres. Ces registres ont l'avantage d'être accessible en un coup d'horloge (en comparaison avec les quelques dizaines de coups d'horloge pour la mémoire) et de pouvoir être accédés de façon parallèle. Le nombre de registres disponibles se mesure généralement en quelques dizaines.

En outre, les registres deviennent très rapidement un goulot d'étranglement: chaque fonction ayant besoin de son propre espace de calcul, doit sauvegarder ses registres avant d'appeler une fonction et les restaurer au retour. Même si cette sauvegarde se fait sur le cache (accessible en quelques coups d'horloge), le temps perdu est énorme.

De plus, toutes les fonctions ayant été compilés pour un certain nombre de registres, changer ce nombre est difficile (les programmes doivent être recompilés pour pouvoir en profiter).

Pour palier à ces problèmes, Sun propose un système de fenêtres de registres. On expliquera dans un premier temps comment fonctionne le système de fenêtres et parlera des avantages. Puis, on détaillera les cas extrêmes et les problèmes lors de l'usage de ce système. Finalement, on comparera les performances avec le système d'allocation globale.

Fenêtres de registres

Hiérarchie des registres d'une fenêtre

Dans le système décrit par Sun, les registres de calcul (donc les registres sauf SP, PC, etc.) sont divisés en quatre groupes:

- **Les registres globaux**: ces registres sont accessibles de toute fenêtre à tout moment, ils sont donc partagés entre les fonctions. Dans l'assembleur Sun, ces registres ont des noms commençant par la lettre g.
- **Les registres locaux**: ces registres sont spécifiques à chaque fenêtre, donc à chaque fonction. Dans l'assembleur Sun, ces registres ont des noms commençant par la lettre l.
- **Les registres entrée**: ces registres sont utilisés pour les paramètres passés par la fonction appelante à la fonction appelée et pour les retours de la fonction appelée vers la fonction appelante. Dans l'assembleur Sun, ces registres ont des noms commençant par la lettre i.
- **Les registres sortie**: ces registres sont utilisés pour les paramètres à passer vers une fonction appelée et pour les retours de la fonction appelée. Dans l'assembleur Sun, ces registres ont des noms commençant par la lettre o.

Dans la norme de Sun, chaque groupe de registre a 8 membres. Une fenêtre, formée des registres entrée, locaux et sortie (les registres globaux sont partagées entre toutes les fenêtres), a donc une taille totale de 24 registres. Sun a fait des processeurs avec un nombre total de fenêtres variant entre 2 et 32.

Passage d'une fenêtre à une autre

Pour changer de fenêtre, nous avons deux appels assembleur:

- **L'appel SAVE**: cet appel est fait avant d'appeler une fonction. Dans ce cas:
  - Les registres globaux ne changent pas
  - Les registres sortie de la fenêtre courante deviennent (=sont renommés en) les registres entrée de la nouvelle fenêtre,
  - Le processeur “alloue” de nouveaux registres locaux et sortie (=prend ceux de la fenêtre suivante).
- **L'appel RESTORE**: cet appel est fait une fois que la fonction est terminée. Dans ce cas:
  - Les registres globaux ne changent pas
  - Les registres entrée de la fenêtre courante deviennent (=sont renommés en) les registres sortie de la nouvelle fenêtre,
  - Le processeur “retrouve” les registres locaux et sortie de la fonction à laquelle on a retourné.

Si on schématise ceci pour un appel de fonction:

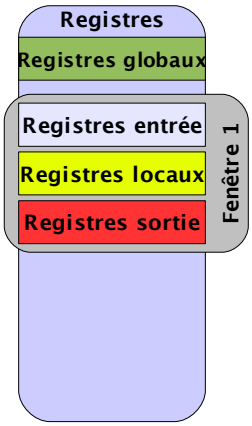

Figure 1: La fonction est dans la fenêtre numéro 1

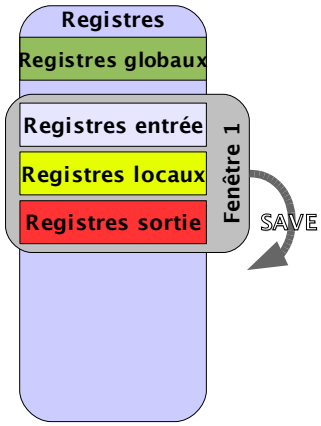

Figure 2: La fonction va appeler une autre: elle met dans les registres de sortie les paramètres de celle-ci, fait un **SAVE** et appelle.

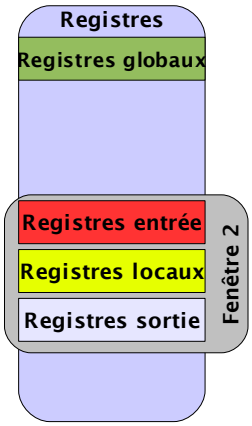

Figure 3: La nouvelle fonction a donc sa fenêtre, et les registres sortie de la fonction d'avant sont devenus les registres entrée de celle-ci.

Le schéma pour le retour est symetrique.

Avantages

Ce système a plusieurs avantages:

- De façon évidente, **elle est plus rapide**: l'appel de fonction n'a besoin d'aucune interaction avec une mémoire externe du moment que les données à passer sont plus petits ou égal à 8 registres. Les fonctions sont donc appelées directement.
- Selon le papier “Reducing instruction fetch cost”, ce système combiné avec un fetch “intelligent” de la mémoire résulte en un **abaissement de la puissance consommée** de 53%.

- Il est **extensible**: les fonctions de disent pas à quelle fenêtre passer, mais appellent SAVE et RESTORE –ils ne sont donc pas au courant du nombre total de fenêtres. On peut donc ajouter autant de fenêtres que l'on veut, et les programmes binaires existants commenceront à profiter de cette augmentation directement, sans besoin de recompilation.

Cas extrêmes

Overflow et Underflow

Même si le nombre de fenêtres est élevé, donc même si on peut appeler, rappeler et encore appeler des fonctions les uns à la suite des autres, il y a un moment où une fonction fera un SAVE et qu'il n'y aura plus de fenêtres disponibles. Ceci est appelé un “overflow” et génère une exception (SPILL) qui est attrapée par le système d'exploitation.

Le système d'exploitation est alors supposé sauvegarder des fenêtres (quelques unes ou toutes) dans un endroit (probablement une pile) et les marquer comme étant “disponibles”. Le processeur peut alors satisfaire l'instruction SAVE et la fonction continue à s'exécuter.

Similairement, quand une fonction fait un RESTORE alors que la fenêtre à restaurer n'est plus disponible (est dans la pile), ceci génère l'exception d'underflow (FILL). Le système d'exploitation est alors supposé restaurer des fenêtres (quelques unes ou toutes) et les marquer comme étant “disponibles”.

Gestion des exceptions SPILL et FILL

Quand le système d'exploitation reçoit l'exception SPILL ou FILL, il a à sa disposition plusieurs registres d'état:

- **CWP** (Current Window Pointer): fenêtre courante.
- **CANSAVE**: Le nombre de fenêtres suivant le CWP qui ne sont pas en cours d'utilisation, donc dans lesquels on peut sauvegarder sans qu'il y ait une exception SPILL.
- **CANRESTORE**: Le nombre de fenêtres précédant le CWP qui sont occupés, donc lesquels que l'on peut restaurer sans qu'il y ait une exception FILL.
- **OTHERWIN**: registres hors ceux couverts par CWP, CANSAVE et CANRESTORE que l'on peut utiliser pour sauvegarder ou restaurer des fenêtres sans avoir à passer par la pile.

Ces constantes satisfont toujours l'équation (avec **NWINDOWS** le nombre total de fenêtres): **CANSAVE** + **CANRESTORE** + **OTHERWIN** = **NWINDOWS** – **2**. Deux fenêtres sont réservées pour les TRAP (en effet, Sun n'oublie pas que l'on peut appeler des fonctions C pour appeler les exceptions; qui auront aussi besoin d'appeler des fonctions).

Quand une exception SPILL est reçue, le gestionnaire de trap SunOS sait que:

- Quand l'exception a eu lieu on était dans une fenêtre valide et la prochaine fenêtre avait été marquée comme étant invalide.
- Comme une exception fait décrémenter le CWP, le gestionnaire d'exception est en train d'être executée dans une fenêtre invalide.

Par conséquent, le gestionnaire doit faire en sorte que la fenêtre courante devienne valide:

- Un SAVE est fait pour aller à la fenêtre suivante.
- Cette fenêtre est sauvegardée dans le registre est marquée comme étant valide.
- Un RESTORE est appelé pour retourner à la fenêtre d'avant (le gestionnaire d'exception va ré-incrémenter le CWP une fois que l'exception a été traitée).

Ceci pose donc un premier problème: un appel de fonction peut être direct ou peut prendre beaucoup de temps; et dans certains cas limites où le passage par la pile est nécessaire pour un grand nombre d'appels **SAVE** et **RESTORE**, les pertes de performance sont énormes (exception + passage par pile).

D'un autre côté, la commutation de thread coute plus chère: il faut sauvegarder et restaurer plus de registres (toutes les fenêtres) que ce dont les programmes ont “vraiment” besoin.

Autre usage: fenêtres de threads

Le fait que le temps nécessaire pour appeler une fonction ne soit pas déterministe est un grand problème pour les systèmes d'exploitation temps réel. En outre, on observe que le système de fenêtre crée “par nature” une isolation entre fonctions.

Le papier “Determinism in RTOS Design for SPARC-like Architectures” décrit un système d'exploitation où chaque thread fonctionne dans sa propre fenêtre, donc où chaque appel SAVE ou RESTORE génère une exception. Pour permettre ceci:

- **CANSAVE** et **CANRESTORE** valent toujours 0 (c'est le RTOS les met à 0),
- Pour que l'équation reste vraie **OTHERWIN** vaut **NWINDOWS – 4**,
- Les registres **WSTATE** et **CLEANWIN** sont utilisés pour l'allocation des registres aux divers threads.

Chaque thread a donc à sa disposition 16 registres (8 registres préxifés **i** et 8 préfixés **l**, les registres **o** étant partagés entre fenêtres adjacentes on ne peut pas les utiliser) et une mémoire partagée de 8 registres. Ceci crée donc un système multithreadé en même temps rapide (commutation entre threads avec un coup moindre) et déterministe.

Fenêtres de registres ou allocation globale ?

Comme on l'a précisé à l'introduction, le système qui s'oppose au système de fenêtres de registres est le système d'allocation globale: dans ce système, tous les registres sont à la disposition et donc c'est le programmeur (enfin... l'optimiseur du compilateur dans le cas général) s'occupe de l'allocation des registres entre divers fonctions. Ceci pose plusieurs problèmes:

- Le problème d'**allocation de registres** se réduit au problème de coloriage de graphe, qui est **NP-complet**. La plupart des optimiseurs font donc des choix plus ou moins au hasard, ce qui crée un code plus ou moins optimisé (il est en effet fort probable que le même programme compilé deux fois ne donne pas le même code binaire).
- Il y a des cas où une allocation des registres n'est pas possible: le cas typique est le cas où le programmeur passe en argument un pointeur vers une fonction et qu'il l'appelle. Comme le compilateur ne peut pas détecter quels fonctions risquent d'être appelés, il ne peut pas décider quels registres sauvegarder / restaurer (généralement il sauvegarde et restaure toutes les registres).
- Le programme compilé est compilé pour un nombre donné de registres. Donc, si on ajoute des registres au processeur, on doit aussi recompiler le programme pour qu'il puisse en prendre compte.

D'un autre côté, on avait remarqué que les performances du système de fenêtres dépendent de comment les overflow et underflow sont gérés dans le système d'exploitation et que l'on s'attend à une baisse des performances si le programme en question est fortement multithreadé. Comparons maintenant les performances que l'on a des deux côtés:

Processor	AMD Athlon MP	Intel Itanium	Sun UltraSPARC II	Sun UltraSPARC III
System or Motherboard	Epox 8KHA+	HP i2000	Sun Enterprs 450	Sun Blade 2050
Clock Rate	1.6GHz	800MHz	480MHz	1,050MHz
External Cache	None	4MB	8MB	8MB
164.gzip	811	332	165	433
175.vpr	423	376	212	460
176.gcc	460	407	232	577
181.mcf	343	402	356	659
186.crafty	942	356	175	558
197.parser	621	296	211	488
252.eon	1280	370	209	527
253.perlbmk	982	320	247	540
254.gap	776	256	171	372
255.vortex	1017	459	304	738
256.bzip2	554	334	237	629
300.twolf	534	449	243	570
SPECint_base2000	677	358	225	537
SPECfp_base2000	588	703*	274	701

Figure: Caractéristiques et performances de divers systèmes
Microprocessor Report, Cahners, 2002

On observe que “le gagnant” en performances dépend du type d'application:

- Dans des applications de type compilateur, placement-routage, de compression de données et d'optimisation combinatoire, le système de fenêtres de registres est de loin le gagnant. Ces applications sont tous monothreadées et font beaucoup d'appels imbriqués de fonctions.
- Dans des applications de type jeux, visualisation informatique ou encore le langage PERL, le système d'allocation globale et bien plus rapide que le système de fenêtres de registres. Ces applications ont tous un point commun: ils sont fortement multithreadés.
- Finalement, les deux systèmes ont des performances comparables pour une application de gestion de base de données. Cette application est en même temps multitheadée et fait beaucoup d'appels de fonctions.

On peut donc dire que le gain en performances de cette méthode dépend du type d'application: on gagne si l'application est monothreadée et qu'elle fait beaucoup d'appels de fonctions; et on y perd du moment que l'application est multithreadée.

Références

Voici les références utilisées dans ce rapport, dans l'ordre alphabétique:

- Determinism in RTOS Design for SPARC-like Architectures, University of Alcala
- Exploiting SPARC Buffer Overflow vulnerabilities, UNF
- Reducing instruction fetch cost, Florida State University, 2005
- Register Renaming, Ecole Polytechnique Fédérale de Lausanne
- Register Renaming, Wikipedia
- Register Windows, Barney Maccabe, 1996
- Register Windows, Nikos Drakos, University of Leeds
- Register Windows, University of New Mexico
- Register Windows, Wikipedia
- Register Windows vs. Register Allocation, David W. Wall, Digital Equipment Corporation
- SPARC traps under SunOS, Jim Moore, 1997
- UltraSparc Unleashes SPARC Performance, Microprocessor Report, 1994
- wof.S (Linux Sparc window overflow handler), David S. Miller, 1995