

Un modèle de mémoire axiomatique

Jade Alglave & Luc Maranget

INRIA

June 3, 2009

Cohérence Séquentielle (SC)

Un modèle de programmation concurrente agréable pourrait correspondre au principe de cohérence séquentielle énoncé par Leslie Lamport en 1979:

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. [Lam79]

Sur un exemple

iw3.1	proc:0	proc:1
poi:0	$y = 1$	$x = 1$
poi:1	$r1 = x$	$r1 = y$
Initial state: $x = y = 0$		

Suivant le principe de cohérence séquentielle, on s'attend à trois sorties possibles:

Allowed: $0:r1 = 0 \wedge 1:r1 = 1$
Allowed: $0:r1 = 1 \wedge 1:r1 = 0$
Allowed: $0:r1 = 1 \wedge 1:r1 = 1$

Expérimentalement

Faisons tourner notre exemple, sur une machine à 4 cœurs POWER5:

```
{0:GPR6=1; 1:GPR6=1; 0:GPR4=x; 1:GPR4=x;  
 0:GPR5=y; 1:GPR5=y; x=0; y=0;}
```

P0		P1	;
stw GPR6,0,GPR5		stw GPR6,0,GPR4	;
lwz GPR1,0,GPR4		lwz GPR1,0,GPR5	;

exists (0:GPR1=0 /\ 1:GPR1=0)

Condition exists (0:GPR1=0 /\ 1:GPR1=0) is validated

Modèles de mémoire relâchés

Pour améliorer leurs performances, les processeurs modernes ont un modèle de mémoire faible, c'est-à-dire relâché par rapport au modèle de cohérence séquentielle:

For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. [Lam79]

Ce résultat est autorisé dans un modèle non SC:

Allowed: $0:r1 = 0 \wedge 1:r1 = 0$

Problèmes

- ▶ Pour écrire des programmes concurrents corrects en assembleur, il faut comprendre les garanties fournies par le modèle de mémoire
- ▶ Les documentations sont peu claires, et surtout ambiguës, puisqu'écrites en langue naturelle
- ▶ Ce problème s'étend aux langages de plus haut niveau, puisque l'algorithme d'exclusion mutuelle de Peterson casse sur des architectures modernes

Notre démarche

- ▶ Lire et tenter de comprendre les documentations des architectures [Pow07, ARM08, ARM, int07]
- ▶ En extraire un modèle de mémoire formel:
 - ▶ décrit en Coq, pour faire des preuves
 - ▶ implémenté en Caml, pour simuler la machine
- ▶ Tester intensivement que notre modèle formel est cohérent vis-à-vis de ce que les implémentations de l'architecture exhibent
- ▶ Itérer

Préambule à la construction du modèle

Condition de cohérence du modèle:

- ▶ Notre modèle formel doit être cohérent vis-à-vis de l'architecture
- ▶ Il doit donc permettre au moins autant de comportements que ceux exhibés par une machine qui implémente l'architecture

Évènements

- ▶ Un programme est une suite d'instructions;
- ▶ Cependant, nos objets de base sont les *évènements* en lecture (R) et écriture (W) engendrés par ces instructions: les instructions ne sont pas considérées atomiques;
- ▶ On donne une sémantique aux instructions en termes d'évènements *mémoire* en lecture et écriture.

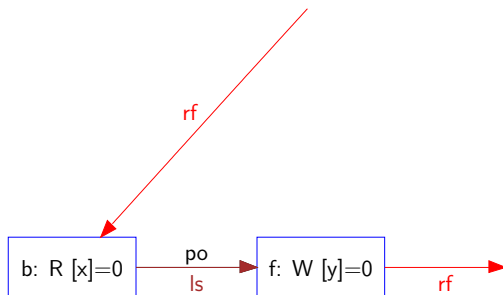
b: R [x]=1

f: W [y]=0

Dépendances locales

L'on tient compte des dépendances engendrées par les accès registres, que l'on consigne dans une relation dp . Par exemple:

ls	proc:0
poi:0	r1 = x
poi:1	y = r1



Structures d'évènements

On rassemble les évènements et les dépendances locales au sein d'une *structure d'évènements*:

$$E \triangleq (\text{events}, dp)$$

Notre sémantique du jeu d'instructions permet de définir un ensemble de structures d'évènements associés à chaque programme.

Témoins d'exécution

Pour former un témoin d'exécution, l'on se munit de trois relations sur les évènements.

Deux d'entre elles sont postulées:

- ▶ rf : relation entre une lecture et l'écriture qui fournit sa valeur
- ▶ ws : ordre total des écritures à une même case (*coherence*)

La dernière est déduite:

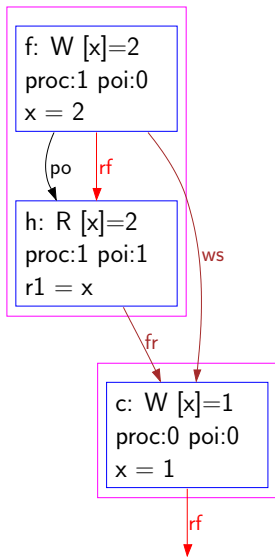
- ▶ fr :

$$e_r \xrightarrow{fr} e_w \triangleq \exists e'_w, e'_w \xrightarrow{rf} e_r \wedge e'_w \xrightarrow{ws} e_w$$

Ainsi:

$$X \triangleq (ws, rf)$$

Sur un exemple



rlns	proc:0	proc:1
poi:0	$x = 1$	$x = 2$
poi:1		$r1 = x$

Critère de cohérence séquentielle

Dans ce contexte, nous avons établi un critère pour décider si une exécution est *SC*:

- ▶ l'union *hb* des relations *ws*, *rf*, *fr* et de l'ordre du programme *po* doit être acyclique,
- ▶ ce qui signifie que l'on peut construire un ordre total des évènements compatible avec l'ordre du programme.

Ainsi:

$$sc\ check\ E\ X \triangleq acyclic\ ((po\ E) \cup (hb\ E\ X))$$

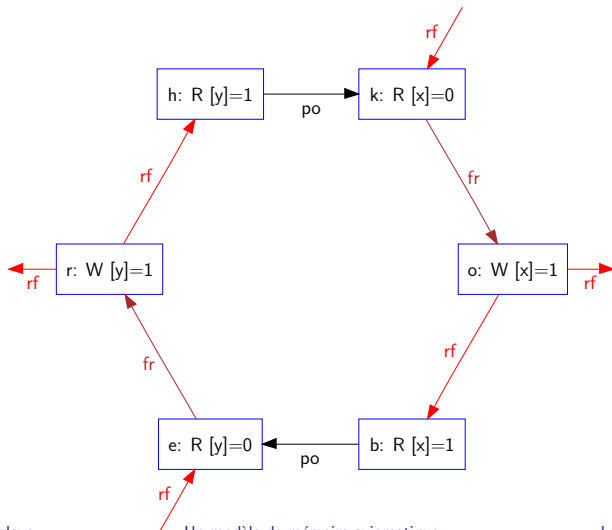
$$\forall E\ X, sc\ check\ E\ X \rightarrow \exists so, sc\ exec\ E\ so \wedge final\ X = final\ so$$

Alors ?

IRIW	proc:0	proc:1	proc:2	proc:3
poi:0	$r1 = x$	$r3 = y$	$x = 1$	$y = 1$
poi:1	$r2 = y$	$r4 = x$		
Initial state: $x = y = 0$				

Une exécution non SC

IRIW	proc:0	proc:1	proc:2	proc:3
poi:0	r1 = x	r3 = y	x = 1	y = 1
poi:1	r2 = y	r4 = x		
Initial state: x = y = 0				



Comportement uniprocasseur

- ▶ On suppose qu'un processeur conserve l'ordre du programme pour les accès à une même case.
- ▶ On se munit de la relation pio pour représenter l'ordre de sortie du processeur: il s'agit de l'ordre du programme pour les accès à une même case;
- ▶ Ainsi, l'union hb des flèches de communication ws , fr et rf doit être compatible avec l'ordre de sortie du processeur pio .

Ainsi:

$$e_1 \xrightarrow{pio} e_2 \triangleq e_1 \xrightarrow{po} e_2 \wedge loc\ e_1 = loc\ e_2$$
$$uniproc\ E\ X \triangleq acyclic\ ((pio\ E) \cup (hb\ E\ X))$$

Causalité

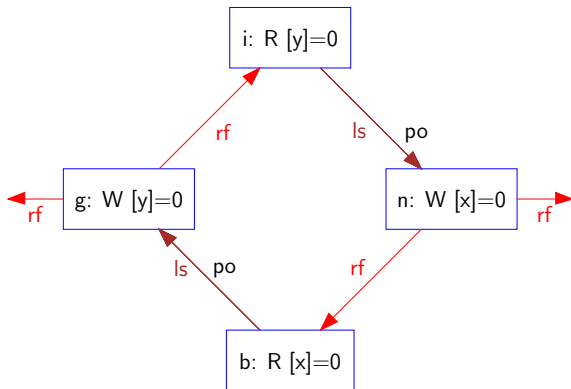
- ▶ Certaines *rf* postulées sont d'emblée incohérentes;
- ▶ En effet, elles sont incompatibles avec les dépendances locales.

Ainsi:

$$\text{causality } E \ X \triangleq \text{acyclic } ((rf \ X) \cup (dp \ E))$$

Sur un exemple

caus	proc:0	proc:1
poi:0	r1 = x	r2 = y
poi:1	y = r1	x = r2



Validité d'une exécution

Pour décider de la validité d'une exécution,

- ▶ l'on vérifie que le comportement *uniprocasseur* et la *causalité* sont respectés;
- ▶ l'on considère les relations *globales*, à savoir *ws* et *fr* ; leur union, notée *ghb*, doit être acyclique.

Ainsi:

$$ghb X \triangleq (ws X) \cup (fr X)$$

$$valid E X \triangleq uniproc E X \wedge causality E X \wedge acyclic (ghb X)$$

Rf globales

- ▶ En fonction des architectures, les *rf* peuvent ne pas être globales, ce qui permet de modéliser par exemple les write buffers.
- ▶ Si toutes les *rf* sont globales, l'on retrouve les modèles à écriture atomique [AM06].
- ▶ Dans la suite, l'on se place dans un cadre où les *rf* ne sont pas globales.

Recouvrer la cohérence séquentielle ?

- ▶ Les barrières sont supposées fournir un mécanisme pour restaurer SC.
- ▶ Une autre relation globale représente l'action des barrières: ab .
- ▶ L'on étend la relation ghb à ab dans la définition de validité d'une exécution.

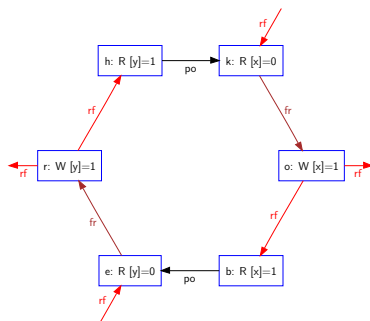
Ainsi:

$$X \triangleq (ws, rf, ab)$$

$$ghb X \triangleq (ws X) \cup (fr X) \cup (ab X)$$

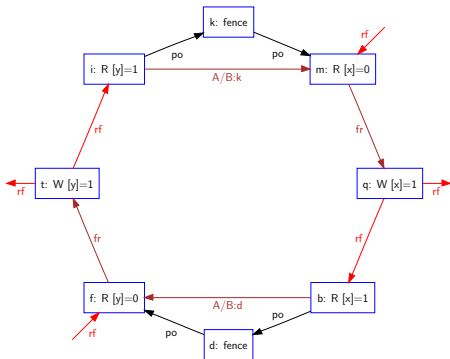
Un modèle fort des barrières

IRIW	proc:0	proc:1	proc:2	proc:3
poi:0	r1 = x	r3 = y	x = 1	y = 1
poi:1	r2 = y	r4 = x		
Initial state: x = y = 0				
Allowed: $0:r1=1 \wedge 0:r2=0 \wedge 1:r3=1 \wedge 1:r4=0$				



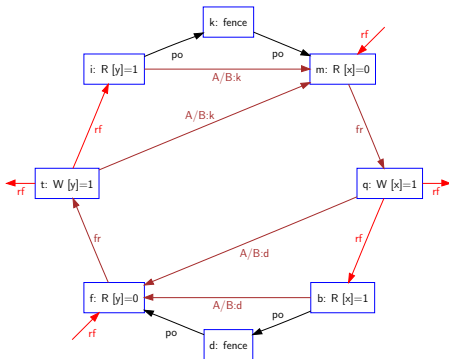
Sans cumulativité

IRIW	proc:0	proc:1	proc:2	proc:3
poi:0	r1 = x	r3 = y	x = 1	y = 1
poi:1	fence	fence		
poi:2	r2 = y	r4 = x		
Initial state: x = y = 0				
Allowed: 0:r1=1 \wedge 0:r2=0 \wedge 1:r3=1 \wedge 1:r4=0				



Avec cumulativité

IRIW	proc:0	proc:1	proc:2	proc:3
poi:0	r1 = x	r3 = y	x = 1	y = 1
poi:1	fence	fence		
poi:	r2 = y	r4 = x		
Initial state: x = y = 0				
Forbidden: 0:r1=1 \wedge 0:r2=0 \wedge 1:r3=1 \wedge 1:r4=0				



Une barrière forte

$$e_1 \xrightarrow{ABTh\ b} e_2 \triangleq$$

$$e_1 \xrightarrow{fenced\ b} e_2 \vee (Base)$$

$$e_1 \xrightarrow{ABTh\ b} w_2 \wedge w_2 \xrightarrow{rf} e_2 \vee (Right : e_2\ R)$$

$$e_1 \xrightarrow{rf} r_1 \wedge r_1 \xrightarrow{ABTh\ b} e_2 \vee (Left : e_1\ W)$$

On démontre qu'une telle barrière permet de restaurer la cohérence séquentielle:

$$\forall E\ X,$$

$$(\forall e_1 e_2, (e_1 \xrightarrow{po} e_2) \rightarrow (e_1 \xrightarrow{fenced} e_2)) \rightarrow$$

$$\exists so, sc\ exec\ E\ so \wedge final\ X = final\ so$$

Expérimentalement

Cependant, sur un 4 cœurs POWER5:

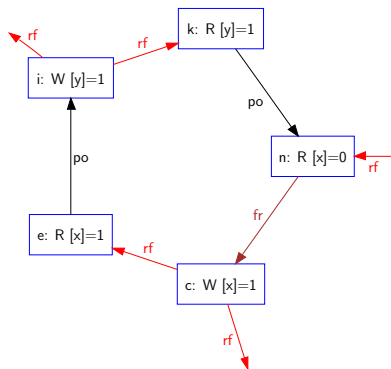
IRIW	proc:0	proc:1	proc:2	proc:3
poi:0	r1 = x	r3 = y	x = 1	y = 1
poi:1	sync	sync		
poi:2	r2 = y	r4 = x		
Initial state: x = y = 0				

Allowed: $r1 = 1 \wedge r2 = 0 \wedge r3 = 1 \wedge r4 = 0$

Faut-il revenir à un modèle de barrières sans cumulativité ?

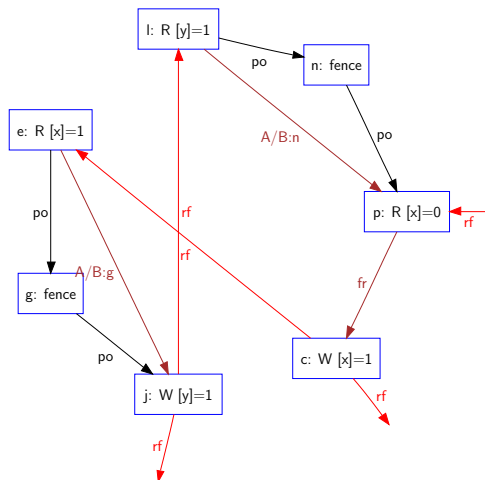
WRC

WRC	proc:0	proc:1	proc:2
poi:0	$x = 1$	$r1 = x$	$r2 = y$
poi:1		$y = 1$	$r3 = x$
Initial state: $x = y = 0$			
Allowed: $1:r1=1 \wedge 2:r1=1 \wedge 2:r3=0$			



WRC : sans cumulativité

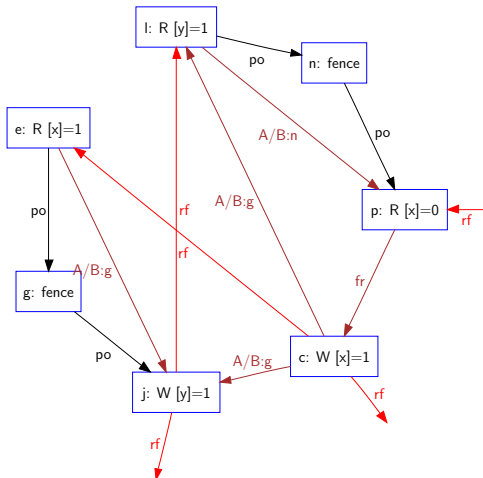
WRC	proc:0	proc:1	proc:2
poi:0	x = 1	r1 = x	r2 = y
poi:1		fence	fence
poi:2		y = 1	r3 = x
Initial state: x = y = 0			
Allowed: 1:r1=1 \wedge 2:r1=1 \wedge 2:r3=0			



WRC : avec cumulativité

Interdit par la documentation [Pow07]

WRC	proc:0	proc:1	proc:2
poi:0	x = 1	r1 = x	r2 = y
poi:1		fence	fence
poi:		y = 1	r3 = x
Initial state: x = y = 0			
Forbidden: 1:r1=1 \wedge 2:r1=1 \wedge 2:r3=0			



Un modèle plus réaliste

$$e_1 \xrightarrow{ABDk \ b} e_2 \triangleq$$
$$e_1 \xrightarrow{\text{fenced } b} e_2 \vee (\text{Base})$$
$$e_1 \xrightarrow{ABDk \ b} w_2 \wedge w_2 \xrightarrow{rf} e_2 \vee (\text{Right} : e_2 \ R)$$
$$\text{write } e_2 \wedge e_1 \xrightarrow{rf} r_1 \wedge r_1 \xrightarrow{ABDk \ b} e_2 \ (\text{LeftW} : e_1 \ W \ \text{and} \ e_2 \ W)$$

Ce modèle est, pour l'instant, validé par l'expérience.

Conclusion

Pour l'instant:

- ▶ Un critère de cohérence séquentielle
- ▶ Un modèle fort des barrières permettant de restaurer SC
- ▶ Un modèle réaliste des barrières validé par l'expérience

Ensuite:

- ▶ Vers un placement des barrières
- ▶ Vers les locks



Arvind and J.-W. Maessen.

Memory model = instruction reordering + store atomicity.

In *Proc. ISCA 2006*, 2006.



ARM.

ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition), April 2008.



ARM.

ARM Barrier Litmus Tests and Cookbook, February 2008.



Intel 64 Architecture Memory Ordering White Paper.

August 2007.



L. Lamport.

How to make a multiprocessor computer that correctly executes multiprocess programs.

IEEE Trans. Computers, 28(9):690–691, 1979.



Power ISA Version 2.05.

October 2007.

Une barrière forte

```
Inductive ABTh (E:Event_struct) (X:Execution_witness) :
  Rln Event :=
  | Base : forall e1 b e2,
    barriered E e1 b e2 -> ABTh E X e1 e2
  | Right : forall e1 b w2 r2,
    ABTh E X e1 b w2 /\ (rf X) w2 r2 -> ABTh E X e1 r2
  | Left : forall w1 r1 b e2,
    (rf X) w1 r1 /\ ABTh E X r1 b e2 -> ABTh E X w1 e2.
```

Un modèle plus réaliste

```
Inductive ABDk (E:Event_struct) (X:Execution_witness) :  
  Rln Event :=  
  | Base : forall e1 b e2,  
    barriered E e1 b e2 -> ABDk E X e1 e2  
  | Right : forall e1 b w2 r2,  
    ABDk E X e1 b w2 /\ (rf X) w2 r2 -> ABDk E X e1 r2  
  | LeftW : forall w1 r1 b w2,  
    write_to_shared E w2 /\  
    (rf X) w1 r1 /\ ABDk E X b r1 w2 -> ABDk E X w1 w2.
```

WRC complet

WRC	proc:0	proc:1	proc:2
poi:0	x = 1	r1 = x	r2 = y
poi:1		fence	fence
poi:2		y = 1	r3 = x
Initial state: x = y = 0			
Forbidden: 1:r1=1 \wedge 2:r1=1 \wedge 2:r3=0			

