Université PIERRE et MARIE CURIE Le : 3/7/2011.

UPMC (Paris VI).

Modélisation et Analyse du protocole de cohérence de caches de la machine multiprocesseur TSAR

Encadrant :

- ✓ D.E.ENCRENAZ.

Réaliser par :

- ✓ Mohamad NAJEM.

PLAN

I- Introduction :

Le projet « Modélisation et Analyse du protocole de cohérence de caches de la machine multiprocesseur TSAR » consiste à modéliser le comportement des éléments de la machine TSAR relatif aux accès aux données en mémoire, afin de vérifier que le protocole de cohérence ne contient pas d'erreur fonctionnelle (deadlock, données erronées,…).

Dans ce rapport je présente le projet TSAR et les différents types de protocole consistaient à assurer la cohérence de communication des mémoires internes aux différentes machines existantes. Puis à expliquer l'outil SPIN qui va me servir à simuler et vérifier les automates du cache L1 et du contrôleur mémoire. Puis je décris les principales étapes de mon projet

II- Description :

Description du projet TSAR :

TSAR qui signifie Tera-Scale Architecture, est une machine multiprocesseur définie dans le cadre d'un projet européen coordonné par la société BULL. La machine intègre jusqu'à 4096 processeurs regroupés en cluster de 4 processeurs avec une mémoire répartie sur l'ensemble des clusters.

Le processeur est de type RISC de 32 bits sans prédictions sur les branchements et peut être un MIPS32, PPC405, SPARC v8 ou bien un ARM7 core.

Chaque processeurs admet deux caches L1 de type Write-through, une pour les instructions et une autre pour les données, et partage la mémoire logiquement. La mémoire étant de taille maximale de 1 Téraoctet (40 bits d'adressage virtuel), est partagée physiquement entre les clusters mais un processeur peut accéder à tout l'espace d'adressage, ce qui rend le temps d'accès à une donnée dépendante de la distance entre le processeur qui fait la demande et la zone mémoire stockant la donnée.

Problématique :

Dans une machine multiprocesseur comme TSAR, il est obligatoire de synchroniser l'accès à une donnée de la mémoire car il y a plusieurs caches qui peuvent simultanément contenir une copie d'un même bloc mémoire.

Par exemple : s'il y a N copies d'un bloc mémoire, et que l'un des processeurs réalise une écriture sur ce bloc, il faut prévenir toutes les autres copies et les invalider. Pour cela un protocole de cohérence est mis en œuvre.

Dans le tableau ci-dessous, vous trouvez un exemple d'exécution d'un système composé de 2 caches et une mémoire, où les valeurs représentent le contenue des cases mémoires indiquées.

Temps	Evènement	Cache C1	Cache C2	Mémoire (X)
0				1
1	C1 : lecture X	1		1
2	C2 : lecture X	1	1	1
3	C1 : Ecriture 0 dans X	0	1	0

Protocole de cohérence :

Un protocole de cohérence est une organisation de l'accès aux blocs mémoires partagés.

Il existe deux différents protocoles basés sur le type de la mémoire : Write-througth et Write-back. Un write-back consiste à avoir une copie exclusive d'un bloc mémoire et la mis à jour au mémoire sera faite lors de la libération du bloc. Par contre, dans un write-through, l'écrite est transmit directement à la mémoire, qui rend la mémoire à jour sur tous ces blocs.

Un processeur peut perdre l'accès à un bloc car il peut être invalidé (écriture) ou évincé (remplacement).

Afin d'assurer cette cohérence, il existe plusieurs protocoles d'invalidation et de mis à jour qui maintiennent l'invariance d'une donnée entre plusieurs lecteurs et écrivains:

Invalidate : Consiste à envoyer des requêtes d'invalidations aux caches qui contiennent des copies d'un bloc mémoire, après une écriture réalisée sur ce bloc.
Update : Consiste à envoyer la nouvelle copie d'un bloc mémoire après une écriture pour tous les caches qui l'admet déjà.

Le nombre de copie sur un même bloc force le choix du protocole qui sera diffusion selon ces deux modes :

- **Multicast (update/invalidate) :**

La cohérence par multicast, est une méthode de diffusion d'un message d'invalidation ou update où on informe tous les caches qui contiennent une copie du bloc mémoire invalide. Cela est réalisé matériellement par la présence des registres pour sauvegarder les adresses des caches qui contiennent des copies adressées à chaque cache pour lui indiquer l'invalidation. Mais ce protocole ne sera plus intéressant en cas d'un grand nombre de copie, comme par exemple : 4096 processeurs dans TSAR, car pour chaque donnée il faut avoir autant de registre que de processeur pouvant simultanément stocker la copie et cela fait augmenter le prix, la taille et diminuer l'efficacité du système.

- **Broadcast (update/invalidate) :**

Ce protocole consiste à informer tous les caches par une diffusion broadcast d'invalidation ou bien update une fois une écriture est signalée. Cela est réalisé matériellement avec une adresse de broadcast commune et connue par tous les caches pour reconnaitre la requête. Ce système mémoire n'a pas besoin de registres pour sauvegarder puisque à chaque fois, la requête d'invalidation est envoyée à tous les caches, ce qui induit une forte charge du réseau.
La diffusion par broadcast est efficace lorsqu'un système mémoire admet un grand nombre de caches et la plus part des caches ont une copie du bloc mémoire à invalider. Mais dans les autres cas, ce système serait moins efficace et plus lent que par multicast.

Protocole de cohérence dans la machine TSAR :

Les réalisateurs du projet TSAR ont intégré un protocole de cohérence spéciale, le DHCCP « Distributed Hybrid Cache Coherence Protocol ». Celui-ci combine deux protocoles déjà vus, où le contrôleur mémoire décide à partir d'un nombre NL (Nombre limite DHCCP threshold < 4096) de copies quel protocole de diffusion à utilisé.
Pour N inférieure à NL un multicast update est envoyé vers toutes les caches qui possèdent une copie. Pour N supérieur à NL un broadcast d'invalidation est envoyé vers tous les caches de la machine.

SPIN et model-checking :

Dans ce projet, je veux créer un modèle de composants de la plate-forme, dont le travail est réalisé avec le langage PROMELA. PROMELA est un langage de programmation impératif qui permet de décrire le comportement de chaque processus d'un système, et leurs différentes interactions. Sous PROMELA, les processus sont asynchrones, communiquant à travers des canaux de communication FIFO bornés en réalisant des transferts de messages dans les canaux (read/write).
Dans mon projet, je modélise un automate d'état par un processus qui s'exécute. L'étape qui suit la programmation d'un automate par PROMELA est de modéliser, tester et vérifier les résultats par model-checking du programme. Ces étapes sont assurées par simulation avec l'outil SPIN. SPIN permet d'étudier un système à état fini et il est bien adapté à une initiation aux problèmes de la vérification d'algorithmes répartis.
Un model-checking, c'est un algorithme (CTL et PLTL) qui représente des propriétés proposées sous forme des formules, où l'outil SPIN simule un système et compare les résultats avec les formules calculées.

III- Etapes du projets :

Dans ce projet, je dois modéliser les automates du système mémoire sur une plate-forme de 3 processeurs puis simuler et vérifier ce protocole. Ces étapes sont réalisées avec l'outil de simulation et de vérification en model checking SPIN.

Plate-forme et canaux de communication:

Dans ce projet, on travaille sur une plate-forme intégrant 3 processeurs et donc 3 caches (instructions et donnés) disposés sur 2 clusters différents. Dans un premier temps, on réduit les caractéristiques pour avoir des caches qui contiennent une seule ligne et une mémoire partagée de 2 adresses de lignes distinctes mappées sur la même ligne de cache.

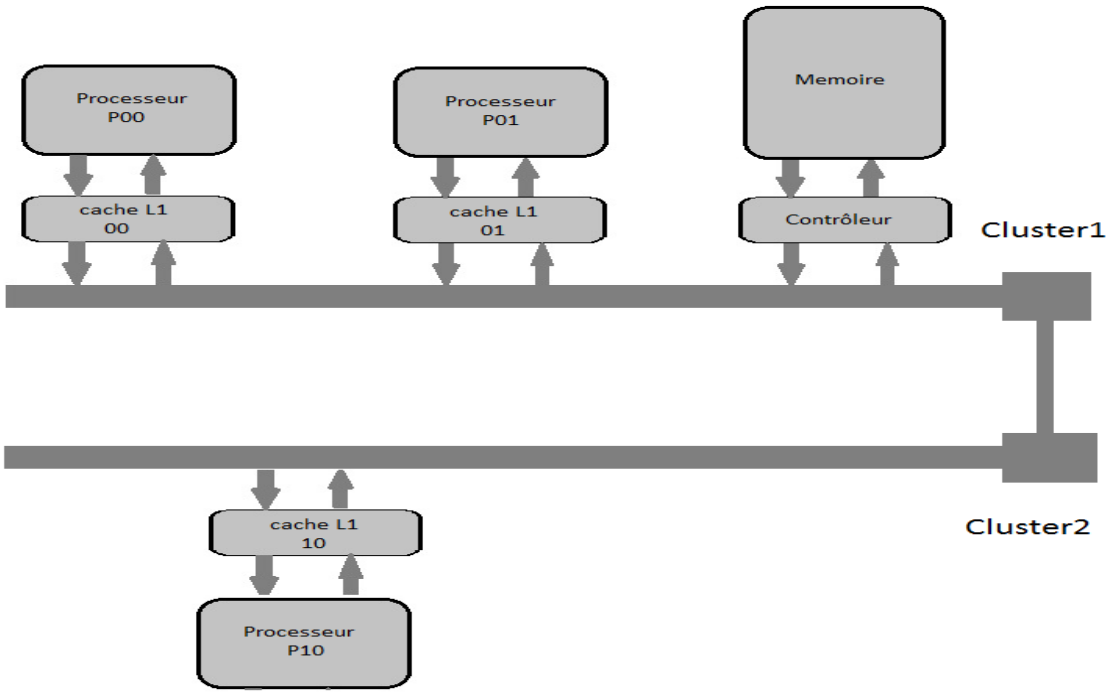


figure1: Plate-forme du projet.

Les composants de la plate-forme communiquent entre eux par des requêtes dans des canaux de communication qui sont partagés entre 3 différentes couples S-D (source - destinataire).

Proc-L1:

PL1DTREQ (Processor to L1 Data Requeste): Canal qui sert à transmettre les demandes du processeur d'une donnée vers la cache. Et ces demandes peuvent être une de des 4 : DT_RD (read) et DT_WR (write). L'adresse mémoire des données sera transmise avec ces demandes.

L1PDTACK (L1 to Processor Data Acknowledge): Canal qui s'occupe de répondre aux requêtes transmises pas le PL1DTREQ. En cas de demande de lecture, la donnée sera

transmise dans ce canal. Le type de requête peut être : ACK_DT_RD (lecture validée), ACK_DT_WR (écriture validé).

L1 – Contrôleur Mémoire :

L1MCDTREQ (L1 to Memory Controller Data Request): Canal transmettant les requêtes de lecture en cas de MISS et d'écriture du cache L1 au contrôleur de la mémoire : RD (read), WR (écrire).

MCL1DTACK (Memory Controller to L1 Data Acknowledge): Canal transmettant la réponse au requête de lecture/écriture (**L1MCDTREQ**) et qui peut contenir les donnés : ACK_RD (lecture validé) et ACK_WR (écriture validé).

L1MCCUREQ (L1 to Memory Controller Clean Up Request): La mémoire cache L1 peut transmettre une requête CLNUP (Clean up) pour vider des blocs mémoires cela à travers ce canal qui lui, transmet la demande et l'adresse mémoire à vider.

MCL1CUACK (Memory Controller to L1 Clean Up Acknowledge): La réponse à la requête de nettoyage ACK_CLNUP d'un bloc de mémoire sera transmise par ce canal.

MCL1CPREQ (Memory Controller to L1 Coherence Protocol Request) : Le protocole de cohérence est assuré par les requêtes d'invalidation et d'update géré par le contrôleur dans ce canal de communication. Les requêtes peuvent varier selon l'état du bloc dans son cache et être une des 3 suivantes : B_INV (Broadcast invalidation), M_INV et M_UP (Multicast invalidation, update).

L1MCCPACK (L1 to Memory Controller Coherence Protocol Acknowledge): Ce canal contient un message de retour qui représente la réponse à l'opération demandé par le canal de cohérence. Et les requêtes sont : ACK_B_INV (Broadcast d'invalidation validé), ACK_M_INV (multicast d'invalidation validé) et ACK_M_UP (multicast update validé).

Contrôleur mémoire – Mémoire :

MCMEMDTREQ (Memory Controller to Memory Data Request) : Le contrôleur mémoire organise les demandes de lecture et écriture en mémoire en envoyant les commandes dans ce canal (GET/PUT).
MEMMCDTACK (Memory to Memory Controller Data Acknowledge) : La réponse du mémoire vers des requêtes de lecture ou bien d'écriture sera envoyée par ce canal vers le contrôleur mémoire (ACK_GET et ACK_PUT).

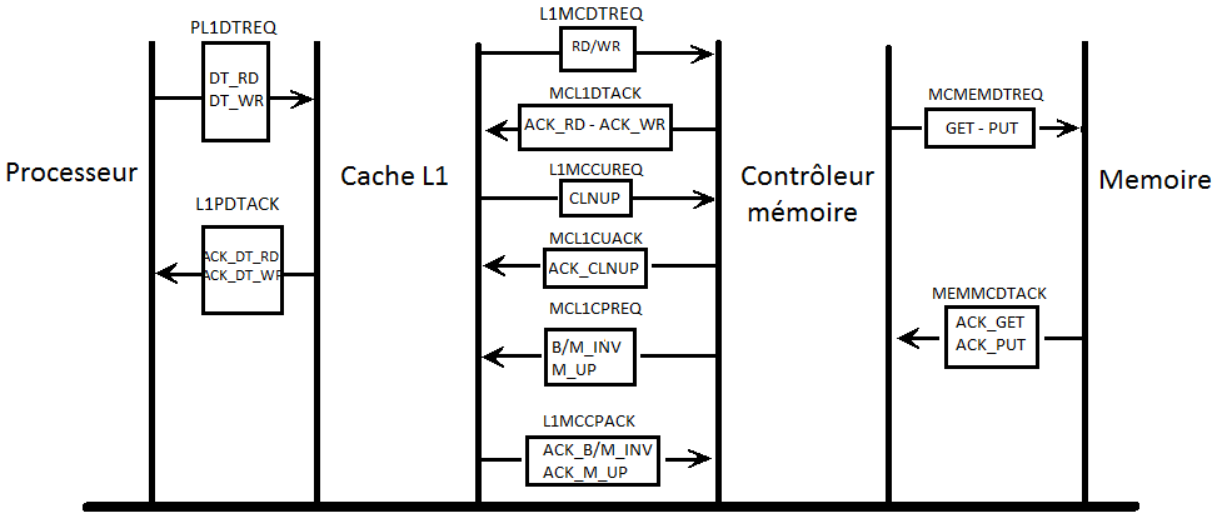


figure2: Canaux de communication.

Spécification de la réalisation :

Dans ce projet, je dispose d'un automate pour le cache L1, un automate pour le contrôleur mémoire et l'architecture de la plate-forme qui ont été proposés par mon encadrant et l'équipe de recherche du laboratoire LIP6 à l'université PIERRE ET MARIE CURIE après une discussion entre eux (voir annexe).
Je pars des modèles simples à réaliser afin de les tester et les vérifier avant de les assembler puis de les complexifier. En premier temps, on se limite aux requêtes de lecture et d'écriture (RD/WR) gérées par le processeur.
On partage le travail en plusieurs réalisations :

1. Description de la plate-forme en langage PROMELA. Dans cette partie, je réalise l'écriture des deux automates en langage PROMELA, en veillant à réduire la domaine de définition des variables afin de limiter l'explosion du nombre d'états.
2. Test élémentaire du système :
   a- Test du cache L1.
   b- Test du modèle du contrôleur mémoire.
   c- Test du système entier.
3. Vérification.

a. Test du cache L1 :

Pour valider le modèle créé, je teste le comportement de l'automate décrit. Ce test se fait sur deux parties : la première consiste à tester les requêtes de lecture et d'écriture dans les différentes états du cache (Miss, vide, hit,…) et pour cela on modélise un processeur qui demande les requêtes du tableau ci-dessous successivement et en remplaçant le contrôleur mémoire par un modèle qui répond par ACK_RD sur les requêtes de lecture et ACK_WR sur les requêtes d'écriture (voir figure 3).

T	PL1DTREQ	Succession des états cache L1	V	L1PDTACK
0	Read(X)	Empty->MISS(X) -> ATT_D(X) ->V(X)	X	ACK_RD
1	Read(X)	V(X)	X	ACK_RD
2	Write(X)	V(X) ->ATT_WR(X) ->V(X)	X	ACK_RD
3	Write(Y)	V(X) ->ATT_WR(Y) ->V(X)	X	ACK_RD
4	Read(Y)	V(X) ->MISS(Y) -> ATT_D(Y) ->V(Y)	Y	ACK_WR
5	Read(Y)	V(Y)	Y	ACK_WR
6	Write(X)	V(Y) ->ATT_WR (X) ->V(Y)	Y	ACK_RD
7	Write(Y)	V(Y) ->ATT_WR (Y) ->V(Y)	Y	ACK_RD

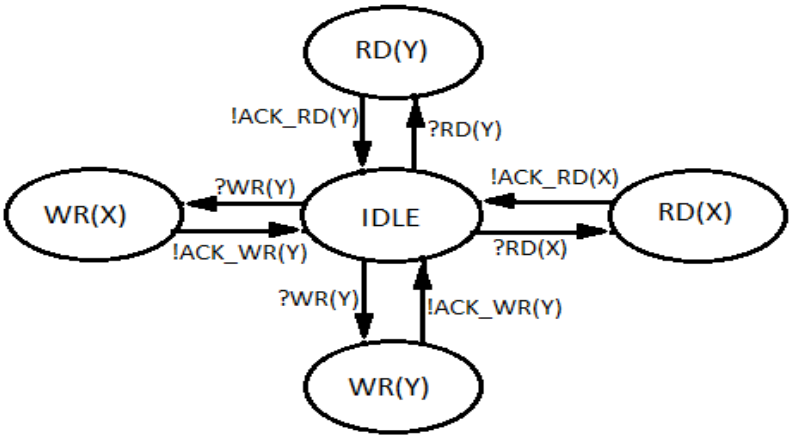


figure3: automate test 1.

Puis on remplace le contrôleur par un deuxième modèle qui teste les invalidations dans les différents états du cache. Ce test est réalisé sur chaque mot mémoire, et dans le tableau ci-dessous vous trouvez un exemple de test sur un des deux adresses mémoires avec l'exécution des requêtes du processeur avant celles de la mémoire.

T	Proc -> L1	MC -> L1	Etat du L1	V
0	##	M/B_INV(X/Y)	Empty	Empty
1	Read(X)	M/B_INV(Y)	Empty -> MISS(X) -> ATT_D(X)	Empty
2	##	ACK_RD	ATT_D(X) -> V(X)	X
3	##	M/B_INV(Y)	V(X)	X
4	Write(X)	M/B_INV(Y)	V(X) -> ATT_WR(X)	X
5	##	M/B_INV(X)	ATT_WR(X) (état différente)	Empty
6	##	M/B_INV(X/Y)	ATT_WR(X)	Empty
7	##	ACK_WR	ATT_WR(X) -> Empty	Empty
8	Write(Y)	M/B_INV(X/Y)	Empty -> ATT_WR(X)	Empty

Le composant qui assure le test de ces états, doit respecter la succession d'états décrite dans la figure4. Dans ces états on teste le comportement de l'automate du cache L1 pour les états propres à la valeur X ; pour tester tout l'automate il faut dupliquer l'automate en remplaçant les adresses X par Y.

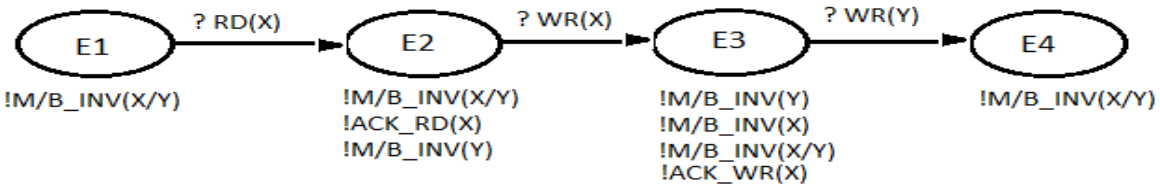


figure4: automate test2.

b. Test du modèle du contrôleur mémoire :

Après la validation du modèle de l'automate de cache L1, je réalise des tests élémentaires sur les différentes transitions entre les états du contrôleur mémoire. Le contrôleur mémoire sera testé pour deux types de requêtes : les demandes de lectures/d'écritures et les requêtes de synchronisations qui sont assurées par les 3 types suivants : M_INV(X) (multicast invalidation), B_INV (broadcast invalidation) et M_UP (multicast update).

Test élémentaire du contrôleur mémoire pour une adresse X ou bien Y fixe pour tout le test.

NB : CACHE_TH = 2 ; // pour n_cache >2 on passe en broadcast.

Temps	Cache L1	C_id	Contrôleur mémoire	N_cache	L_cache	Mémoire
0	WR	0	Empty -> ATT_WR	0	NULL	-
1	WR	1	ATT_WR	0	NULL	-
2	-	-	ATT_WR -> V_M_D	0	NULL	ACK_GET
3	RD	0	V_M_D	1	0	-
4	RD	1	V_M_D	2	0,1	-
5	CLNUP	1	V_M_D	1	0	-
6	WR	1	V_M_D -> ATT_UP	1	0	-
7	ACK_M_UP	0	V_M_D	1	0	-
8	RD	1	V_M_D	2	0,1	-
9	RD	2	V_B_D	3	NULL	-
10	RD	3	V_B_D	4	NULL	-
11	CLNUP	3	V_B_D	3	NULL	-
12	WR	2	V_B_D-> ATT_B_INV	3	NULL	-
13	ACK_B_INV	0	ATT_B_INV	3	NULL	-
14	ACK_B_INV	1	ATT_B_INV	3	NULL	-
15	ACK_B_INV	2	ATT_B_INV -> ATT_PUT	0	NULL	-
16	-	-	ATT_PUT -> Empty	0	NULL	ACK_PUT
17	RD	0	Empty -> ATT_RD	0	NULL	-
18	-	-	ATT_RD -> V_M	1	0	ACK_GET
19	RD	1	V_M	2	0,1	-

20	CLNUP	1	V_M	1	0	-
21	RD	1	V_M	2	0,1	-
22	RD	2	V_B	3	NULL	-
23	RD	3	V_B	4	NULL	-
24	CLNUP	3	V_B	3	NULL	-
25	WR	2	V_B -> ATT_B_INV	3	NULL	-
26	ACK_B_INV	0	ATT_B_INV	3	NULL	-
27	ACK_B_INV	1	ATT_B_INV	3	NULL	-
28	ACK_B_INV	2	ATT_B_INV -> ATT_PUT	3	NULL	-
29	-	-	ATT_PUT -> Empty	0	NULL	ACK_PUT
30	RD	0	Empty -> ATT_RD	0	NULL	-
31	WR	1	ATT_RD -> ATT_RD_WR	0	NULL	-
32	WR	2	ATT_RD_WR	0	NULL	-
33	-	-	ATT_RD_WR -> V_M_D	1	0	ACK_GET
34	RD	1	V_M_D	2	0,1	-
35	RD	2	V_M_D -> V_B_D	3	NULL	-
36	WR	3	V_B_D -> ATT_B_INB	3	NULL	-
37	ACK_B_INV	0	ATT_B_INV	3	NULL	-
38	ACK_B_INV	1	ATT_B_INV	3	NULL	-
39	ACK_B_INV	2	ATT_B_INV -> ATT_PUT	3	NULL	-
40	-	-	ATT_PUT -> Empty	0	NULL	ACK_PUT
41	RD	0	Empty -> ATT_RD	0	NULL	-
42	-	-	ATT_RD -> V_M	1	0	ACK_GET
43	RD	1	V_M	2	0,1	-
44	WR	2	V_M -> ATT_UP	2	0,1	-
45	ACK_M_UP	0	ATT_UP	2	0,1	-
46	ACK_M_UP	1	ATT_UP -> V_M_D	2	0,1	-

c. Test du système entier :

Dans cette phase, je place les deux modèles validés dans un même programme, pour vérifier le comportement de tout le système. A la place du processeur, on réalise un générateur aléatoire de requête de lecture et d'écriture puis on simule le système puis on vérifie qu'il n'y a pas un mauvais fonctionnement (requêtes invalides, deadlock, …) en faisant un examen visuel sur les résultats des simulations et en utilisant des automates observateurs pour la phase de vérification avec model-cheking.
Puis je décris la plate-forme en assemblant 3 générateurs (processeurs) avec leurs caches et le contrôleur mémoire avec une mémoire de 2 valeurs. Je teste pour valider le comportement du système entier.

3. Vérification :

Après la réussite de toutes les étapes décrites ci-dessus, j'arrive à la phase attendue qui est le but du projet et qui le valide. Dans cette étape, il faut vérifier le protocole de cohérence proposé par les réalisateurs du projet TSAR.
On définit les propriétés attendues du système :

- Absence de blocage : Le système pourra toujours évoluer : il ne reste pas bloqué dans un état sans pouvoir progresser.
- Toute requête de lecture et d'écriture demandée par le processeur, finira par être fournie.
- Validité du compteur de copies dans le contrôleur mémoire à des instants d'observations pertinents.
- Toutes écritures ou lectures postées reçoit la dernière valeur écrite : En cas de multiprocesseur, cette propriété est vraie pour les requêtes qui sont émient après l'acquittement de l'écriture (causalité).

On lance le modèl-checker qui va vérifier ces propriétés sur le système décrit et nous retourner les résultats. On traite ces résultats pour retirer une conclusion du système. En cas de réponse positive, on vérifie que le système répond bien à ce qu'on a défini comme propriété, sinon les comportements du système sont en contradiction avec les propriétés non vérifiées : il faut analyser les contre-exemples pour expliquer le résultat de vérification.

Validation du projet :

La validation du projet porte sur la fourniture :

- Deux descriptions des automates L1 et contrôleur mémoire testé par simulation.
- L'expression des propriétés de cohérence.
- Le résultat de vérification par model-checking pour la plate-forme, que la vérification ait aboutie (ce que l'on espère) ou non (si l'explosion combinatoire est trop rapide).
- Enfin, si j'ai le temps, analyse des résultats, éventuellement simplification de la plate-forme pour contourner les problèmes d'explosion combinatoire.

IV- Réalisation :

Modélisation en promela :

Pour modéliser les composants de la machine TSAR, je représente chaque état par un label dans le code « promela ». Puis je défini les fonctions de transitions de chaque état. Etant donné que l'automate se comporte comme un automate de « MEALY », la sortie de l'automate sera modifiée seulement quand il y a une transition, et la transition est réalisée par l'exécution d'une instruction de saut vers un label désigné (goto label).

1. Structure de message :

La communication entre les différents modèles est faite à l'aide des canaux. Pour cela je fixe une structure qui représente un message par les champs suivants :

- **Mtype type :** Type du message (DT_RD, WR, M_UP, ACK…).
- **Bit addr :** L'adresse de la case mémoire.
- **Bit val :** La valeur de la case mémoire.
- **Byte cache_id :** Identifiant du cache.

Cette structure se trouve dans le fichier « lib_project.h ».

2. Modèle du cache L1 :

Je dispose d'un automate (voir annexe) qui représente le comportement du Cache L1 dans la machine TSAR.

Le modèle du cache L1 est écrit en PROMELA dans le fichier « CacheL1.h ».

a- Prototype :

Proctype CacheL1 (chan PL1DTREQ, L1PDTACK, MCL1CPREQ, L1MCCPACK, L1MCCUREQ, MCL1CUACK; byte c_id);

Les canaux PL1DTREQ, L1PDTACK, MCL1CPREQ, L1MCCPACK, L1MCCUREQ et MCL1CUACK comme entrées/sorties de l'automate.
C_id : identifiant du cache.

b- Etats :

Pour modéliser cet automate, on définit d'abord les labels qui correspondent aux états :

- **Empty** : Cache ne contient pas de données.
- **ATT_WR1_X** : Attente de l'acquittement de l'écriture de X. Etat associé à Empty (Cache vide).
- **ATT_WR1_Y** : Attente de l'acquittement de l'écriture de Y. Etat associé à Empty (Cache vide).
- **MISS_X** : Cache n'admet pas la donnée à l'adresse X.
- **ATT_D_X** : Attente de l'acquittement de la lecture de X.
- **CL_R_X** : X est invalidé avant la réception de l'acquittement.
- **V_X** : Cache contient la valeur de l'adresse X.
- **ATT_WR2_X** : Attente de l'acquittement de l'écriture de X. Etat associé à V_X (Cache contient X).
- **ATT_WR2_Y** : Attente de l'acquittement de l'écriture de Y. Etat associé à V_X (Cache contient X).
- **MISS_Y** : Cache n'admet pas la donnée à l'adresse Y.
- **ATT_D_Y** : Attente de l'acquittement de la lecture de Y.

- **CL_R_Y** : Y est invalidé avant la réception de l'acquittement.
- **V_Y** : Cache contient la valeur de l'adresse Y.
- **ATT_WR3_X**: Attente de l'acquittement de l'écriture de X. Etat associé à V_Y (Cache contient Y).
- **ATT_WR3_Y**: Attente de l'acquittement de l'écriture de Y. Etat associé à V_Y (Cache contient Y).

c- Fonction de transition :

De chaque état l'automate attend sur un nombre défini de canaux, pour détecter la transition d'un état à un autre. Par ex : De l'état « EMPTY » vers l'état « MISS_X », il faut lire un message de lecture de X qui vient du processeur (canal PL1DTREQ).

On distingue 2 types de transitions :

- Une transition liée aux requêtes de lecture/écriture du processeur et leurs acquittements du contrôleur mémoire.
- Une transition liée aux requêtes qui correspondent au protocole de cohérence implémenté par le système (Invalidation et UPDATE).

d- Variables internes :

Cette machine à état admet des variables locales qui servent à tester les requêtes, et une case mémoire qui contient la copie d'une adresse mappée en mémoire.

- **Byte v_cache** : Case mémoire interne qui contient la valeur d'une adresse (X ou bien Y).
- **Bit v_cache_valide** : bit de validité de la case mémoire interne.
- **Bit v_addr** : variable qui contient l'adresse associée à la donnée dans le cache.
- **Bit Vcl** : bit de validité du CLNUP. Ce bit interdit la soumission d'une deuxième CLNUP sur une adresse avant la réception de l'acquittement de la première.
- **Msg m** : structure d'un message, qui sert à lire/écrire dans les canaux.

Les variables internes sont modifiées lors d'une transition.

e- Hypothèses :

Pour bien effectuer les différentes transitions et définir les états où le cache conserve son propre état, plusieurs hypothèses ont été prises en considération dans ce modèle après la discutions avec mon encadrant et les responsables du projet TSAR :

- Les invalidations sont traitées dans tous les états du modèle.
- Les requêtes de multicast UPDATE sont prises dans tous les états. En notant que dans les états « EMPTY », « ATT_WR1_X » et « ATT_WR1_Y » on ne doit pas recevoir cette requête.

- Consommation des requêtes d'acquittement des CLNUP dans tous les états.
- Cache L1 n'attend pas l'acquittement de CLNUP.
- Cache L1 n'est autorisé à envoyer une requête n+1 de CLNUP que si elle a déjà reçue l'acquittement de CLNUP de n.

La première version de modèle de cache L1 prend en considération toutes ces propriétés, mais après la simulation de la plate-forme à 3 processeurs, je détecte un blocage du système lié à la deuxième hypothèse. En fait cette hypothèse ne permet pas au système de fonctionner puisque les requêtes d'éviction (CLNUP) seront considérés comme réponse aux requêtes de cohérence émise par le contrôleur mémoire (Multicast update « M_UP »). Un acquittement à la requête « M_UP » doit avoir lieu si le cache admet la copie. Par contre un cache peut recevoir un « M_UP » dans un état où il n'admet pas la copie, puisque le modèle du cache n'attend pas l'acquittement des évictions (CLNUP) donc il peut y avoir un cas où le cache envoie un « CLNUP » vers le contrôleur mémoire et ce dernier n'a pas encore traiter cette requête.

Scénario de contradiction de la deuxième propriété :

Voici un scénario qui bloque le canal d'acquittement des requêtes de cohérence. Dans cette figure, on peut remarquer que si le contrôleur mémoire émet une requête M_UP, il attend qu'une seule réponse. Si le cache envoie un « CLNUP », le contrôleur décrémente le nombre de réponse qui doit attendre et donc un acquittement qui suit le cleanup va rester dans le canal des requêtes et bloquer le système après un certain temps.

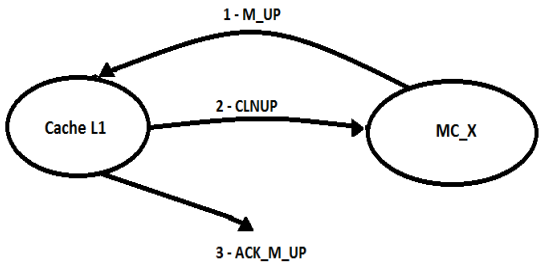


figure ? : scénario de blocage du canal d'acquitement des requêtes de cohérence.

Solution proposée :

Je modifie le modèle du cache d'une façon à tester l'adresse désigné par la requête de cohérence M_UP. Si l'automate du cache est dans un état où il admet la valeur d'une adresse en mémoire et la requête M_UP désigne la même adresse, le modèle répond à cette requête.

f- Réalisation des testes élémentaires :

Les tests décrits dans la partie « Spécification » assurent tous les transitions possibles entre les états. Les deux types de requêtes décrites ci-dessus seront testés séparément.

Requête de lecture/écriture :

Pour vérifier la première séquence de test, je définis les modèles minimaux ci-dessous dans le fichier « CacheL1_v2_bis.pml » :

Test_P1 : Ce modèle monodirectionnel, remplace le processeur, pour envoyer des requêtes bien définies.

Test_C1 : Ce modèle remplace le contrôleur mémoire, et répond aux requêtes de lecture/écriture sans traitement des invalidations et des updates.

Je simule le fichier « promela » avec ces composants et je vérifie avec mon encadrant que les transitions obtenues sont équivalentes aux ceux qui sont prévues dans le tableau (voir test du Cache L1 dans la partie Spécification).

Requête d'invalidation et d'update :

Cette séquence de test est testée par les modèles minimaux ci-dessous dans le fichier « CacheL1_v3_bis.pml » :

Test_P2 : Modèle qui remplace le processeur. On envoie des requêtes définies pour forcer l'automate du cache L1 dans un état, afin de tester les transitions liées au protocole de cohérence.

Test_C2 : Modèle qui remplace le contrôleur mémoire. Il envoie des requêtes d'invalidation et d'update avant qu'il acquitte les demandes de lecture et d'écriture.

Dans ce système, on vérifie que tout les transitions liées au protocole de cohérence gérées par le contrôleur mémoire sont effectuées. Je simule le système et on observe les différents états de l'automate du cache L1. Ces différentes transitions ont été validées avec mon encadrant. Après ces 2 tests effectués sur le modèle du cache L1, on peut garantir que le modèle crée se comporte bien comme il a été défini par les responsables du projet dans le schéma donné.

3. Modèle du contrôleur mémoire :

Pour le contrôleur mémoire, je réalise 2 modèles qui fonctionnent en parallèle dans le système, un pour l'adresse X et l'autre pour Y.

a- Prototype :

Proctype MC_X (chan L1MCDTREQ, MCL1DTACK, L1MCCUREQ, MCL1CUACK, MCL1CPREQ, L1MCCPACK, MCMEMDTREQ, MEMMCDTACK) ;

Proctype MC_Y (chan L1MCDTREQ, MCL1DTACK, L1MCCUREQ, MCL1CUACK, MCL1CPREQ, L1MCCPACK, MCMEMDTREQ, MEMMCDTACK) ;

Les canaux L1MCDTREQ, MCL1DTACK, L1MCCUREQ, MCL1CUACK, MCL1CPREQ, L1MCCPACK, MCMEMDTREQ et MEMMCDTACK sont les entrées/ sorties de l'automate de chaque modèle.

b- Etat :

Je décris les différents états de l'automate du modèle pour l'adresse X. Et pour le modèle de Y, il suffit de remplacer « X » par « Y ».

- **Empty_X** : Le contrôleur mémoire n'admet pas la valeur de l'adresse mémoire X.

- **ATT_WR_X** : Etat d'attente de lecture d'un bloc mémoire. Le contrôleur mémoire reçoit une écriture sur l'adresse X, et dans cet état une demande du bloc mémoire qui contient X est effectuée.

- **ATT_RD_X** : Etat d'attente de lecture d'un bloc mémoire. Le contrôleur mémoire reçoit une lecture sur l'adresse X, et dans cet état une demande du bloc mémoire qui contient X est effectuée.

- **ATT_RD_WR_X** : Etat d'attente de lecture d'un bloc mémoire. Le contrôleur reçoit une écriture après une lecture qui n'a pas encore été servie par la mémoire.

- **V_M_D_X** : Le contrôleur mémoire admet une copie de la valeur de l'adresse mémoire X qui n'est pas propre (dirty : une écriture été effectuée sur cette adresse et le contrôleur mémoire n'a pas encore écrit en mémoire « Write-back»). Le contrôleur mémoire est en état multicast pour le protocole de cohérence.

- **V_M_X** : Le contrôleur mémoire admet une copie de la valeur de l'adresse mémoire X qui est propre (aucune écriture n'a été effectuée sur cette adresse). Le contrôleur mémoire est en état multicast pour le protocole de cohérence.

- **ATT_UP_X** : Etat d'attente des acquittements des caches après un multicast update. C'est l'état transitoire entre V_M_X et V_M_D_X, c.a.d une écriture à été reçue par le contrôleur mémoire après un nombre limité de lecture qui garde multicast comme protocole de cohérence.

- **V_B_X** : Le contrôleur mémoire admet une copie propre de la valeur de l'adresse mémoire X. Le contrôleur mémoire est en état broadcast pour le protocole de cohérence.

- **V_B_D_X** : Le contrôleur mémoire admet une copie dirty de la valeur de l'adresse mémoire X. Le contrôleur mémoire est en état broadcast pour le protocole de cohérence.

- **ATT_B_INV_X** : Etat d'attente des acquittements des caches après un broadcast d'invalidation.

- **ATT_PUT_X** : Etat d'attente de l'acquittement de l'écriture du bloc en mémoire.

c- Fonction de transition :

On distingue plusieurs types de transitions d'états :

- Transition liée aux premières requêtes de lecture/écriture envoyée par un cache qui initialise l'état du contrôleur mémoire. Et ces transitions sont : Du Empty vers ATT_WR_X et ATT_RD_X.
- Transition liée à la réception d'une requête d'écriture qui modifie l'état de la valeur dans le cache de propre vers dirty. Ces transitions sont : V_M_X vers V_M_D_X et V_B_X vers V_B_D_X.
- Transition liée au changement de protocole de cohérence. Et cela dépend du nombre de cache qui admet une copie. Pour un nombre n de cache supérieure à NL le cache change d'état de V_M_X vers V_B_X et de V_M_D_X vers V_B_D_X.
- Transition liée à la réception d'une requête d'écriture qui force le contrôleur mémoire à envoyer le bloc vers la mémoire et initialiser son état.

Les mêmes transitions dans le contrôleur mémoire pour l'adresse Y, mais en changeant les états (remplaçant X par Y).

d- Variable interne :

- **Byte src** : Variable temporelle pour sauvegarder l'identifiant du cache source avant de l'ajouter au tableau de copie.
- **Byte v_x** : valeur de la case mémoire d'adresse X.
- **Byte c_id_tmp** : Variable temporelle pour sauvegarder l'identifiant du cache.
- **Byte cpt** : compteur interne.
- **Byte cpt_rep** : Compteur de réponse en cas de M_UP et B_INV.
- **Byte c_id[CACHE_TH]** : tableau qui sauvegarde les identifiants des caches qui ont une copie de X en cas de multicast.
- **Bit v_c_id[CACHE_TH]** : bit de validité de l'identifiant sauvegardé dans c_id.
- **Byte n_cache** : compteur de copie.

Ces variables changent de valeurs à chaque transition.

NB : Je définis dans le fichier « memory_controler.h » la constante CACHE_TH avec la valeur 2, et qui représente la valeur limite qui force le basculement de multicast vers broadcast dans le contrôleur.

e- Hypothèse :

Pour le contrôleur mémoire, je définis avec mon encadrant plusieurs hypothèses :

- Un contrôleur mémoire pour X et un contrôleur mémoire pour Y qui fonctionnent en parallèle. Les deux adresses X et Y n'entrent pas en collision dans le contrôleur mémoire.
- Les deux adresses X et Y appartiennent à 2 blocs mémoires différentes.

- Les requêtes de lecture/écriture ne seront traitées qu'après tous les acquittements et les cleanups.
- Le contrôleur mémoire consomme le message de broadcast invalidation après la réception de tous les acquittements.

f- Réalisation des tests élémentaires :

Dans la partie spécification, j'ai défini une suite de tests qui valide toutes les transitions possibles de chaque état de l'automate du contrôleur mémoire. Ces tests sont réalisés par la simulation d'un système qui comporte les composants suivants (minim_model.h):

Minim_L1 : Modèle qui remplace l'automate du cache L1. Il envoie des requêtes définies dans le tableau de la spécification, qui force les transitions dans le contrôleur mémoire.

Minim_MEM : Modèle qui remplace la mémoire. Il répond aux requêtes de GET et PUT du contrôleur mémoire après un petit délai (voir 5.).

Je simule le système avec les 3 modèles, et je valide avec mon encadrant toutes les transitions prévues pour l'automate du contrôleur.

Comme ces modèles sont généraliste, on test le modèle du contrôleur mémoire pour Y et X par le même système, mais en modifiant le paramètre du modèle « Minim_L1 » et remplaçant MC_X par MC_Y (modèle de X par modèle de Y).

4. Modèle du processeur :

Pour compléter la plate-forme, j'ai crée un modèle qui remplace le processeur, et il génère des requêtes de lecture/écriture sur X et Y aléatoirement, et attend l'acquittement de ces requêtes. Le modèle « proc » est défini dans le fichier « minim_modèle.h » et son automate est représenté par le schéma ci-dessous.

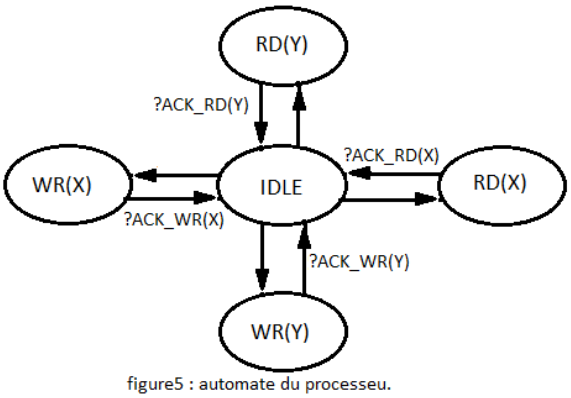


figure5 : automate du processeu.

5. Modèle de la mémoire :

J'utilise le modèle « Minim_MEM » crée pour tester le contrôleur mémoire. C'est un modèle qui répond aux requêtes de lecture/écriture de bloc mémoire (GET/PUT) envoyées par les 2 contrôleurs mémoire X et Y (voir schéma ci-dessous).

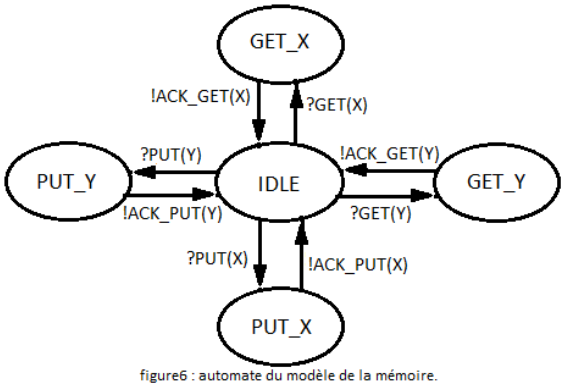


figure6 : automate du modèle de la mémoire.

Propriété de safety :

1. Plateforme 1 proc :

a- Simulation et vérification sans observateurs :

Après la création des différents composants de la plate-forme à 1 processeur sans observateurs, je simule le système jusqu'à profondeur de 10000 steps. En observant les résultats avec les yeux, on peut déduire que le système se comporte comme il est prévu sans blocage. Je passe le modèle checker de l'outil SPIN sur la plate-forme et il arrive à vérifier pour tous les combinaisons d'état possible (environ 700000 états) qu'il n'y a pas de dead-lock dans le système.

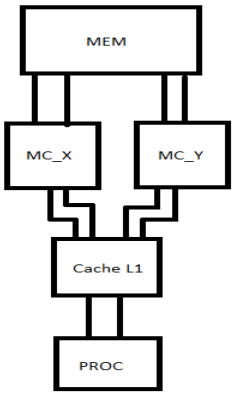


figure ? : Plate-forme à 1 processeur

b- Simulation et vérification avec observateurs :

1. Observateurs :

Pour vérifier des propriétés sur la plate-forme, je crée deux observateurs. Dans un premier temps, le rôle des observateurs était de regarder les requêtes transmises sur les canaux de communications entre le processeur et le cache L1. Mais en simulant le système, on déduit que l'observateur ne détecte pas

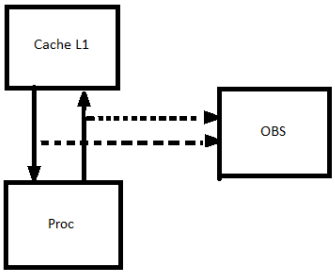


figure ? : Solution 1, observateur sans consomation de message.

toutes les requêtes, et existe des cas où l'observateur peut rater une requête. Pour cela les résultats ne deviennent plus acceptables.

La lecture de l'observateur n'est pas consommatrice alors que celle du cache l'est : selon l'ordre d'exécution des lectures dans le canal (d'abord l'observateur ou d'abord le cache), l'observateur observe la donnée avant sa consommation par le cache ou bien la rate.

Je modifie le modèle du processeur pour ajouter deux canaux qui transmettent les requêtes du processeur vers l'observateur. Et donc le processeur sera bloqué tant que l'observateur n'a pas récupéré son ancienne requête (Fifo à profondeur 1) avant d'émettre la nouvelle vers le cache L1.

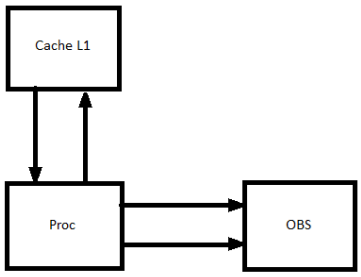


figure ? : Solution2 : observateur avec consomation

- Read after write :

J'ai créé un observateur qui vérifie que tous les requêtes de lecture consécutive à une écriture sur une adresse précise (X ou bien Y) seront acquittées par la dernière valeur écrite (attention : n'est pas applicable en multiprocesseur). Et il sera représenté par l'automate suivant :

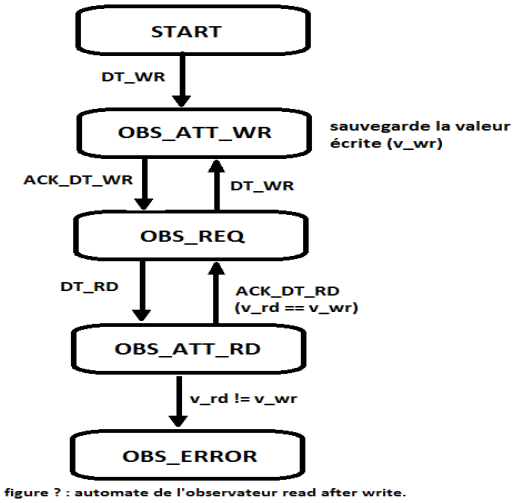


figure ? : automate de l'observateur read after write.

L'observateur commence à indiquer sont point de départ qui est la première écriture (DT_WR). Il sauvegarde la valeur écrite avec l'attente de l'acquittement et il passe dans l'état d'observation des requêtes. Dans cet état il détecte deux types de requêtes : Soit des écritures et cela réinitialise l'automate, soit des lectures et dans ce cas il attend l'acquittement. Après l'acquittement de lecture, il vérifie si la valeur retourner est égale à la valeur sauvegarder lors de l'écriture, et sinon il passe dans un état d'erreur où on place une asserte pour arrêter le système.

- Read after read :

J'ai créé un observateur qui vérifie que toutes les requêtes de lecture successives sur une même adresse (X ou bien Y) sans écriture entre elles, seront acquittées toujours par la même valeur. Et il sera représenté par l'automate suivant :

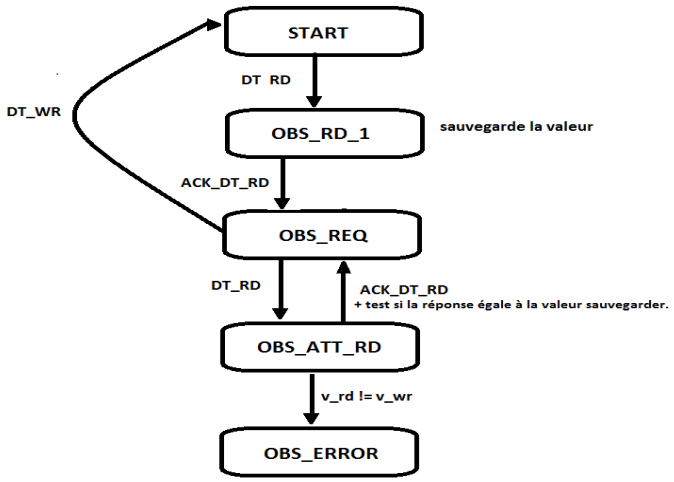


figure ? : schéma de l'automate de l'observateur read after read.

L'observateur « read after read » détecte la première requête de lecture envoyée par le processeur vers le cache où il passe dans un état d'attente de l'acquittement de lecture afin de sauvegarder la valeur lue, puis l'observateur passe dans l'état où il observe toutes les requêtes émises par le processeur. En cas des requêtes de lecture, l'observateur attend l'acquittement puis vérifie que la valeur retournée est équivalente à la valeur sauvegardée, sinon il passe en état d'erreur où un assert est placé dans le code pour informer le système que la propriété n'est pas valide. Enfin, en cas des requêtes d'écriture l'observateur attend l'acquittement de l'écriture et passe à l'état de départ pour détecter une nouvelle lecture.

2. Simulation et vérification :

Je simule la plate-forme à 1 processeur avec observateur pour une profondeur de 10000 états. En observant de la trace d'exécution, je peux déduire que le système fonctionne normalement et aucun blocage n'est détecté.

Pour la phase de vérification avec modèle checking, la mémoire disponible (4 Go) n'était pas suffisamment grande pour engendrer tous les états du système en mode exhaustive. Spin arrive à vérifier partiellement le système sans détection de blocage ou bien des asserts pour 10^7 états. Je passe l'outil de vérification avec l'option de simplification des états (hash-compact) de SPIN pour le même espace mémoire. J'obtiens aussi des résultats partiaux mais pour 3.5*10^7 états sans assert détectés.

c- Conclusion sur les propriétés :

Le résultat de simulation et de vérification obtenu sur la plate-forme à 1 processeur décrit ci-dessus (avec et sans observateurs), on peut déduire les propriétés suivantes :

a- Absence de blocage du système proc + cache + contrôleur mémoire.
b- Chaque requête de lecture/écriture finit par être servie (le processeur ne se bloque jamais).

Et on peut déduire des résultats partiaux concernant les propriétés à vérifiées par les observateurs sur la portion de l'espace d'état analysé (3.5*10^7 états et profondeur de 300000 transitions):

a- Toutes les lectures successives qui suivent une écriture se terminent et retournent la dernière valeur écrites.
b- Toutes les lectures successives sans écriture entre eux se terminent et retournent la même valeur.

2. Plateforme 3 proc :

Je construis une plate-forme à 3 processeurs, 3 caches, 1 contrôleur pour l'adresse X, 1 contrôleur pour l'adresse Y et 1 mémoire. Puisque la mémoire n'a pas servie à supporter la plate-forme à 1 processeur, je simplifie la plate-forme pour avoir :

- Un processeur qui demande requêtes de lecture et d'écriture sur X et Y aléatoirement (modèle décrit ci-dessus).
- Un processeur qui demande des requêtes de lecture sur X.
- Un processeur qui demande des requêtes de lecture et d'écriture sur Y.

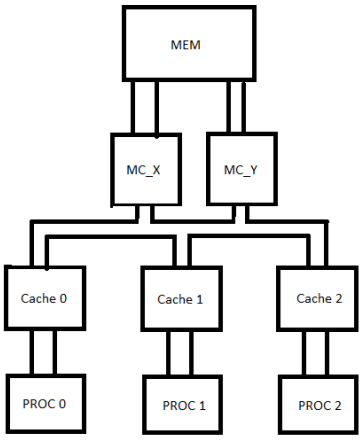


figure ? : Plate-forme à 3 processeur.

Ces simplifications n'affectent pas la généralité des résultats de vérification que l'on peut obtenir : Puisqu'il n'y a pas de différence entre les mapping des 2 adresses X et Y, donc les 2 adresses sont symétriques c-à-d chaque propriété conclue sur l'une est forcément conclue sur l'autre.

a- Simulation et vérification sans observateurs :

Après la simulation de la plate-forme avec Spin pour une profondeur de 10000 états, et l'observation du comportement du système, je peux déduire que la plate-forme fonctionne bien et sans blocage des processeurs ni des contrôleurs de mémoire.

Afin de vérifier les absences de blocages dans la plate-forme, je travail avec mon encadrant sur une machine à 16Go de mémoire, on lance l'outil de vérification et ca dure plusieurs

heures, pour finir avec des résultats partiel sur un espace d'état de 5*10^8 états sans détection des asserts ni blocage du système.

b- Simulation et vérification avec observateurs :

Les observateurs utilisés sont identiques à celui de la plate-forme à 1 processeur, mais on associe à chaque processeur 2 canaux pour envoyer ces propres requêtes (6 canaux d'entrée par observateur).

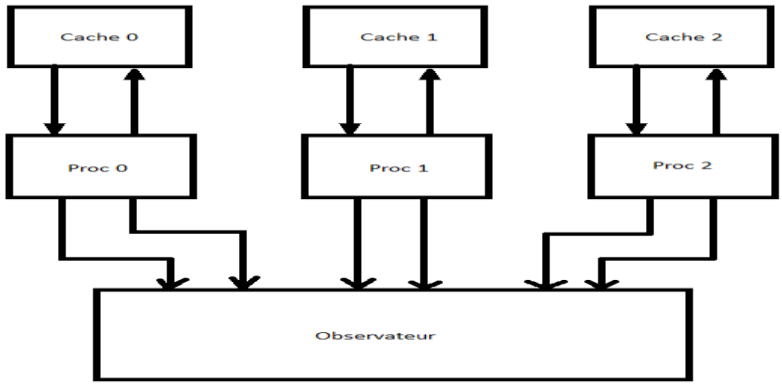


Figure ? : Observateur dans la plate-forme à 3 proc.

Pour l'observateur « read after read », il détecte la première lecture envoyée par n'importe quel processeur, il sauvegarde la valeur acquittée par le cache associé, puis il observe les requêtes émises par tous les processeurs afin de tester en cas de lectures leurs valeurs de retours si elles sont équivalentes à celles qui a été lue initialement.

L'observateur « read after write » détecte son état de départ qui est la première écriture effectuée par n'importe quel processeur, puis il observe les requêtes émises par tous les processeurs. En cas de lecture, il teste si la valeur acquittée est égale à la valeur sauvegardée au départ.

Je simule la plate-forme à 3 processeurs sous SPIN, un assert a été signalé dans la profondeur 3500 steps. Un cas d'incohérence était détecté par simulation où la propriété de « read after write » ne sera plus valide.

c- Conclusion sur les propriétés :

Après la simulation des 2 plates-formes (avec et sans observateurs), et la vérification incomplète de la première, on peut déduire des propriétés partielles sur le système pour la portion d'états examiné :

- Absence de blocages au niveau du système complet.
- Toute lecture qui suit une écriture ne sera pas forcément acquittée par une valeur égale la dernière valeur écrite (à jour). Et cela dépend du temps de réponse sur l'écriture qui est supérieur au temps de réponse de la lecture si le cache admet la copie (voir ci-dessous) :

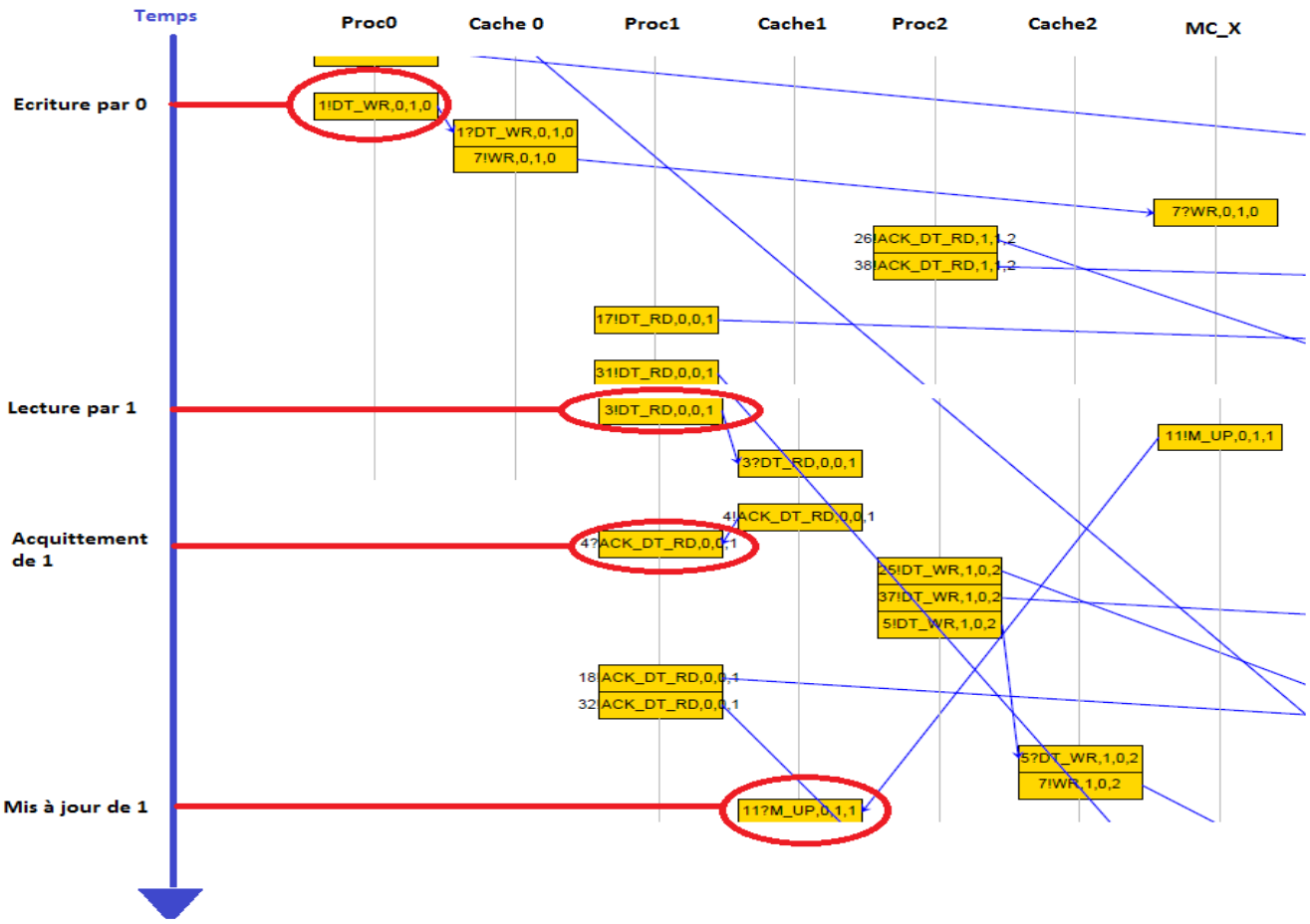


figure ? : simulation

Dans cette figure, on a la trace d'exécution de la plate-forme, où on remarque bien l'instant où le processeur demande l'écriture sur l'adresse X et l'instant où le processeur 1 demande la lecture de cette adresse. L'écriture sera transmis vers le cache et puis vers la mémoire qui elle-même doit gérer la cohérence de tous les copies avant d'acquitter l'écriture. Par contre, Le cache 1 admet une copie de X donc la requête de lecture est transmis vers le cache qui va répondre directement au processeur. La valeur retournée par le cache 1 n'est pas à jour, puisque le cache n'a pas encore été informé de l'écriture de 0.

L'observateur « read after write » du cas monoprocesseur n'est pas transposable au cas multiprocesseur : il ne permet pas de distinguer les dépendances causales entre read/write sur des processeurs différents (alors que les dépendances sur un unique processeur sont causales).

Matériellement cette incohérence ne peut pas être résolue. Une solution logicielle est nécessaire par la prise des verrous par le système d'exploitation pour créer des dépendances entre les différents accès à une même adresse et donc avoir l'accès exclusif pour les écrivains et bloquer les processeurs lecteurs dans une boucle d'attente active. Après l'acquittement des écritures, les écrivains libèrent le verrou et donc cela assure que tous les copies on été mis à jour avant n'importe quelle lecture.

V- Résultat et conclusion:

Après mon travail sur ce projet, je fournis :

- Les modèles du cache L1, des Contrôleur mémoire pour X et pour Y et des composants minimaux (processeur et mémoire) en langage PROMELA.
- Les propriétés conclus sur ce système décrit ci-dessus.
- Les testes élémentaires et les résultats de la vérification du model-cheker de l'outil SPIN.

Dans ce projet on détecte quelque hypothèse sur les modèles qui induit une mal fonctionnement du système et on peut considérer ces résultats comme un signal aux responsables du projet TSAR. On arrive à vérifier quelque propriété sur le système entier et les autres ne pourront pas être vérifiés avec les mêmes outils à cause de l'explosion mémoire que j'ai eu lors de mon travail. Un travail supplémentaire sur ce projet est nécessaire pour terminer les vérifications en utilisant des outils plus fréquents et qui utilisent une technique de réduction de la zone mémoire réservé pour toute l'espace d'états du système.

Annexe

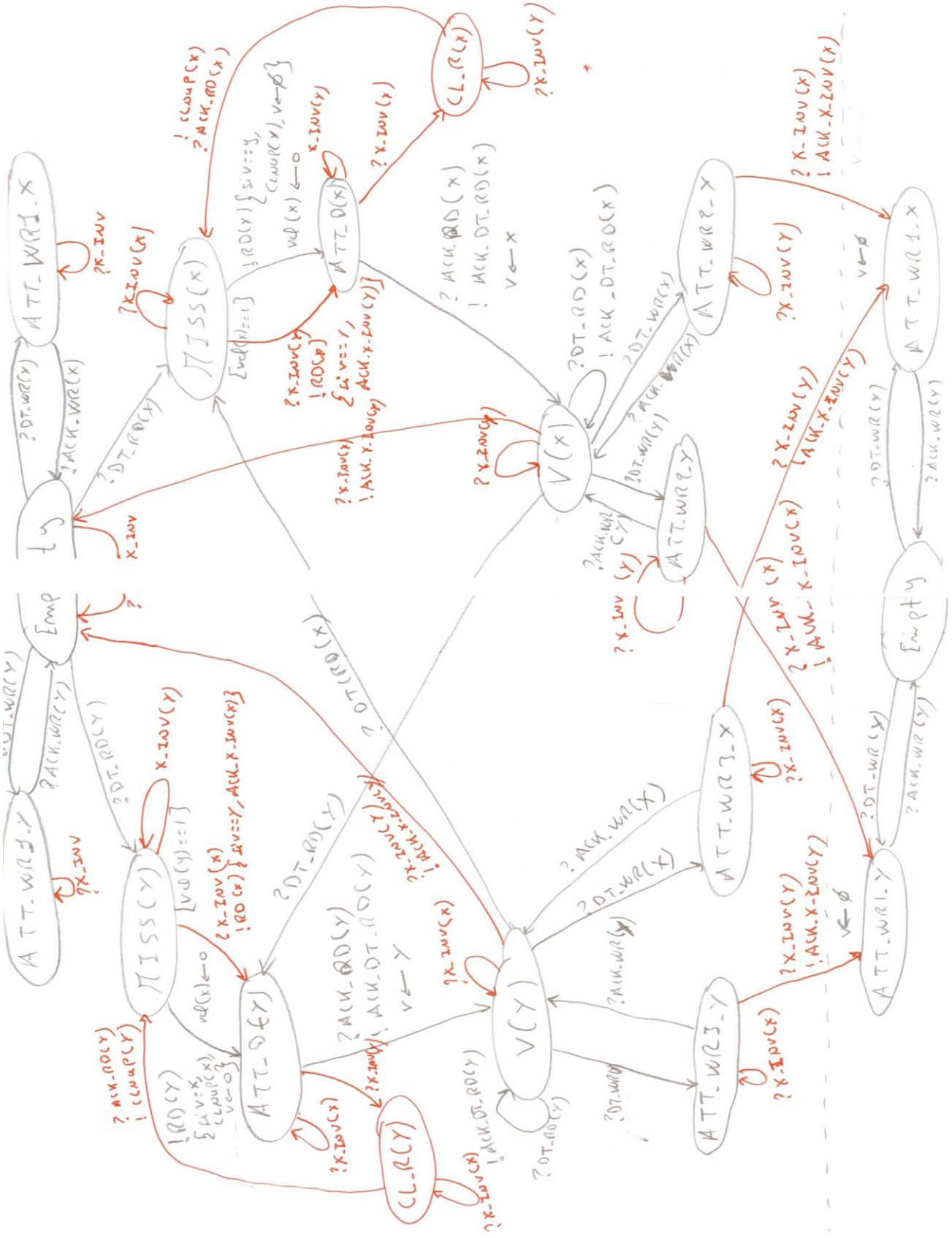
Les fichiers qui contiennent les modèles et les testes réalisé avec les vérifications se trouvent dans le répertoire « PSESI_TSAR » fournit avec le rapport. Vous trouvez dans ce répertoire les sous répertoire ci-dessous :

- Test_code1 : Ce répertoire contient les testes élémentaire réalisé sur le modèle du cache L1. Mais ce modèle est directement écrit dans les fichiers « .pml ». Vous obtenez les résultats des tests en simulant ces fichiers.
- Test_code2 : Ce répertoire contient les premiers tests réalisés sur le modèle du cache L1 pour les requêtes de lecture et d'écriture. Ce modèle se trouve dans le fichier « CacheL1.h » qui est inclus dans le fichier « CacheL1_v2_bis.pml ». Vous obtenez les résultats des tests en simulant ce fichier.
- Test_code2_bis : Ce répertoire contient les tests réalisés sur le modèle du cache L1 pour les requêtes de cohérence. Ce modèle se trouve dans le fichier « CacheL1.h » qui est inclus dans le fichier « CacheL1_v3_bis.pml ». Vous obtenez les résultats des tests en simulant ce fichier.
- Test_code3 : Dans ce répertoire il ya le fichier « test_MC_X.pml » qui teste le contrôleur mémoire pour l'adresse X et qui est écrite dans le fichier « Memory_controller.h ». Vous obtenez les résultats des tests en simulant le fichier « .pml ».
- Test_code4 : Dans ce répertoire il ya le fichier « test_MC_Y.pml » qui teste le contrôleur mémoire pour l'adresse Y et qui est écrite dans le fichier « Memory_controller.h ». Vous obtenez les résultats des tests en simulant le fichier « .pml ».
- Test_plat1 : Dans ce répertoire, le fichier « plat1.pml » écrit la plate-forme à 1 processeur sans observateur où vous simuler et vérifier avec l'outil SPIN ce fichier pour obtenir les résultats décrite dans le projet. Les observateurs sont décrites dans le fichier « obs.h » et les modèles des composants minimaux dans le fichier « minim_model.h ».
- Test_plat1_v2 : Dans ce répertoire, le fichier « plat1.pml » écrit la plate-forme à 1 processeur avec observateur où vous simuler et vérifier avec l'outil SPIN ce fichier pour obtenir les résultats décrite dans le projet. Les observateurs sont décrites dans le fichier « obs.h » et les modèles des composants minimaux dans le fichier « minim_model.h ».
- Test_plat2 : Dans ce répertoire, le fichier « plat2.pml » écrit la plate-forme à 3 processeurs sans observateur où vous simuler et vérifier avec l'outil SPIN ce fichier pour obtenir les résultats décrite dans le projet. Les observateurs sont décrites dans le fichier « obs.h » et les modèles des composants minimaux dans le fichier « minim_model.h ».
- Test_plat2_v2 : Dans ce répertoire, le fichier « plat2.pml » écrit la plate-forme à 3 processeurs avec observateur où vous simuler et vérifier avec l'outil SPIN ce fichier pour obtenir les résultats décrite dans le projet. Les observateurs sont décrites dans le

fichier « obs.h » et les modèles des composants minimaux dans le fichier « minim_model.h ».

NB : Les fichiers headers « .h » où j'ai codé les modèles sont identiques dans tous les sous répertoires sauf dans quelque uns le fichier « obs.h » est modifié pour généraliser le modèle en gardant le même nom comme prototype. Attention à ne pas écraser se fichier !

Automate du cache L1 :



Automate du Contrôleur mémoire :