Stage de Master 2 Recherche en Informatique

Vérification de programmes assembleur concurrents sur modèle mémoire x86-TSO

Auteur :
David Declerck

Maître de stage :
Dr. Sylvain Conchon

Organisme d'accueil :
Laboratoire de Recherche en Informatique
Université Paris-Sud

Secrétariat - tél : 01 69 15 75 18 Fax : 01 69 15 42 72
courrier électronique : m2-info-nsi.sciences@u-psud.fr

17 mars – 12 septembre 2014

Table des matières

Résumé

Avec le développement des architectures parallèles, la programmation concurrente est une discipline qui devient de plus en plus présente. Tout programmeur non averti est tenté d'adopter le modèle naturel de *cohérence séquentielle*[14] lorsqu'il conçoit des programmes pour ces architectures. Dans ce modèle, un programme concurrent est vu comme l'entrelacement des instructions qui le composent. Bien que ce modèle permette de raisonner facilement sur des programmes concurrents, il s'avère également incorrect. En effet, les optimisations réalisées au sein des microprocesseurs modernes donnent un modèle plus relâché, dit *modèle mémoire faibe*[2], dans lequel les opérations mémoire peuvent prendre effet dans un ordre différent de la séquence d'instructions du programme. S'assurer de la sûreté de tels programme devient alors une tâche particulièrement ardue, d'autant plus que les outils existants ignorent ces particularités.

On propose de vérifier des programmes écrits dans un sous-ensemble du langage d'assemblage x86, en créant un compilateur qui traduit ces programmes vers des systèmes de transitions paramétrés[12]. Cette traduction prend en compte les particularités liées au modèle mémoire, notamment le modèle x86-TSO[17], mis en oeuvre dans les processeurs Intel et AMD. Les systèmes ainsi produits peuvent alors être vérifiés par le *model checker* Cubicle[11]. Par ailleurs, afin de s'assurer que le résultat trouvé par Cubicle puisse être transposé dans la sémantique initiale des programmes assembleur, on prouve par bisimilation l'équivalence de cette sémantique avec celle des systèmes de transition générés pour Cubicle. Le bon fonctionnement de l'outil est démontré par des résultats expérimentaux, permettant de mesurer son efficacité sur différents programmes assembleur concurrents.

Mots clefs

Vérification, concurrence, thread, modèle mémoire faible, processeur, x86, TSO.

CHAPITRE 1

INTRODUCTION

L'omniprésence des multiprocesseurs et des processeurs multi-cœurs impose un style de programmation concurrente pour tirer parti de leur puissance. Le programmeur doit alors gérer plusieurs *threads*, qui s'exécutent en parallèle et accèdent de façon concurrente à une mémoire partagée. La programmation sous ce paradigme est plus subtile que la programmation séquentielle, et raisonner sur de tels programmes n'est pas trivial.

En général, les programmeurs conçoivent leurs algorithmes concurrents en ayant à l'esprit une vision "naturelle" de l'exécution d'un programme, dite *séquentiellement consistante* (SC). Ce modèle est ainsi décrit 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"[14]. En d'autres termes, un programme concurrent est vu comme l'entrelacement des instructions qui compose ses différents *threads*, et qui respecte l'ordre imposé par le code source du programme.

Pour illustrer ce modèle, prenons le programme concurrent suivant écrit dans un pseudo-langage d'assemblage x86 :

Etat initial : x = 0, y = 0	
Thread 1	Thread 2
MOV [x], 1 MOV EAX, [y]	MOV [y], 1 MOV EBX, [x]

Très naturellement, le modèle SC impose qu'à la fin de l'exécution de ce programme, au moins l'un des deux registres EAX ou EBX contienne la valeur 1, quel que soit l'entrelacement réalisé. Cependant, si on exécute ce programme sur un processeur multi-cœurs moderne, par exemple un Intel Core i7, on peut parfois (une fois sur un million) observer un état final tel que les deux registres contiennent la valeur 0.

Dans les faits, le modèle SC ne correspond pas au modèle mis en oeuvre dans les architectures multi-cœurs actuelles. En effet, afin d'améliorer les performances, les processeurs effectuent en interne diverses optimisations, telles que le réordonnancement des instructions, la mise en mémoire tampon des opérations mémoire, la non-atomicité des instructions ou encore l'exécution spéculative. Si ces optimisations sont transparentes pour un programme séquentiel, les programmes concurrents s'en trouvent fortement affectés : il n'y a plus d'ordre total sur les opérations mémoire, et pas de notion de temps global. Chaque *thread* peut donc voir l'effet des opérations mémoire dans un ordre et à des moments différents. On parle alors de *modèles mémoire faibles*[5].

Concrètement, cela revient à relâcher certaines contraintes sur l'ordre des opérations mémoire effectuées par un même *thread*. Différents modèles mémoire existent pour qualifier différentes combinaisons de relâchements. Le plus simple est le relâchement de l'ordre *lecture-après-écriture*, dans lequel une opération de lecture séquencée après une opération d'écriture peut prendre effet avant cette dernière. C'est notamment le cas sur les processeurs Intel et AMD, dans lesquels les écritures en mémoire sont mises dans des tampons de taille inconnue, ce qui retarde leur écriture en mémoire et explique le comportement observé dans l'exemple précédent. On nomme ce modèle Total Store Order (TSO[16]). Si on relâche en plus l'ordre *écriture-après-écriture*, autorisant deux écritures séquencées l'une après l'autre à prendre effet dans l'ordre inverse, on obtient le modèle Partial Store Order (PSO). Enfin, si on relâche en plus les ordres *lecture-après-lecture* et *écriture-après-lecture*, c'est-à-dire si on autorise toutes les opérations mémoire à prendre effet dans un ordre ne respectant pas leur séquence, on obtient le modèle Relaxed Memory Ordering (RMO).

Si ces modèles mémoire permettent d'augmenter l'efficacité des processeurs, certains algorithmes ont parfois besoin d'avoir un comportement en apparence *séquentiellement consistant*. C'est le cas notamment des algorithmes d'*exclusion mutuelle* classiques, tels que Peterson ou de Dekker, que l'on trouve dans la littérature académique. Ces algorithmes ne garantissent plus l'*exclusion mutuelle* lorsqu'ils sont mis en oeuvre sur des *modèles mémoire faibles*. Fort heurseusement, les processeurs fonctionnant sous ces modèles proposent également des mécanismes permettant de retrouver si nécessaire un comportement SC. Il s'agit essentiellement de *barrières mémoire* et d'instructions *atomiques* (préfixées par `lock` en x86), que le programmeur se doit de placer judicieusement dans son programme, afin d'assurer la cohérence des opérations mémoire. Bien entendu, ces mécanismes doivent être utilisés de façon ponctuelle et localisée, ceux-ci impliquant des pénalités en terme de performances.

Ainsi, étant donnée la complexité des algorithmes concurrents, à laquelle s'ajoute la complexité de ces *modèles mémoire faibles*, il devient très difficile de s'assurer qu'un programme concurrent donné soit effectivement correct. Compte tenu de l'explosion combinatoire des entrelacements, une vérification manuelle n'est pas envisageable, et aucun jeu de tests ne peut couvrir tous les cas possibles, d'autant plus que certains comportements de ces modèles mémoire ne sont observables que très rarement. On a donc besoin d'outils permettant de vérifier de façon *automatique* des propriétés de *sûreté* pour ces programmes. Il s'agit de savoir si certains *états dangereux* d'un programme donné, caractérisés par une *valuation* de ses variables, sont *atteignables* depuis son état initial. Par exemple, pour un algorithme d'exclusion mutuelle, un tel état sera caractérisé par deux threads en section critique simultanément. Pour un algorithme de cohérence de cache, ce pourra être un état tel que deux caches possèdent de copies différentes d'une même ligne de cache. Grâce à des travaux précédents[9] que la complexité du problème d'atteignabilité est élevée (non-primitive récursive) pour les modèles TSO et PSO, et que ce problème est *indécidable* pour le modèle RMO (à moins de borner le nombre d'opérations mémoire pouvant être "sautées" lors d'un réordonnancement).

La plupart des outils de vérification existant pour programmes concurrents ignorent le modèle mémoire, et simulent donc un modèle SC. Les quelques outils gérant les *modèles mémoire faibles* permettent davantage la restriction de programmes TSO à des comportements SC plutôt que la vérification effective de ces programmes. L'un des seuls outils de vérification existant est CBMC (Université d'Oxford)[7], qui emploie la technique du *Bounded Model Checking*. Les boucles d'un programme sont déroulées un nombre fini de fois, et le nombre de *threads* est fixé. Le modèle TSO est simulé par dessus le modèle SC en transformant le programme de façon à y intégrer des *tampons* de taille fixe (et petite). Une analyse en avant est ensuite effectuée pour trouver les états non sûrs. Parmi les outils restaurant les comportements SC d'un programme TSO, on peut citer Musketeer (Université d'Oxford)[6], Offence (Université d'Oxford / INRIA)[8], Dfence / Fender / Blender (ETH Zurich)[15], Checkfence (Université de Pennsylvanie)[10], MMChecker (Université de Singapour)[13], ou encore Memorax (Université d'Uppsala)[4]. Ces outils insèrent des *barrières mémoire* et des instructions *atomiques* aux emplacements adéquats pour qu'un programme TSO exhibe les mêmes comportements qu'un programme SC. Le nombre de *barrières* insérées n'est pas toujours optimal.

Nous proposons une nouvelle approche dans la vérification de programmes TSO, qui consiste à vérifier des propriétés de sûreté pour un nombre quelconque de *threads* et des tailles de *tampon* arbitraires. Pour cela, on traduit des programmes écrits en assembleur vers des systèmes de transition[12], dont on vérifie ensuite la sûreté avec le *model checker* Cubicle[11]. Ce *model checker* permet de s'affranchir de la limitation sur la taille des tampons, à laquelle les autres outils sont sujets. Nous travaillerons dans un premier temps sur le modèle mémoire TSO, avec des programmes écrits directement en assembleur x86, ceci afin de développer notre technique. D'autres modèles et langages, notamment le C11[3] qui définit un modèle mémoire très relâché, pourront alors être envisagés dans des travaux ultérieurs.

CHAPITRE 2

APERÇU DE L'APPROCHE

L'idée principale de notre approche est de créer un compilateur qui prend en entrée des programmes écrits dans un sous-ensemble du langage d'assemblage x86 et manipulant des *threads*, et produit en sortie un fichier dans le langage d'entrée du *model checker* Cubicle. Ce compilateur peut générer au choix un *système de transitions* pour le modèle SC ou TSO. Les fichiers ainsi produits peuvent ensuite être analysés par Cubicle, qui détermine si le système qu'ils représentent est sûr ou non en regard des propriétés de sûreté qu'il contient. De plus, on effectue une preuve d'équivalence entre les sémantiques du langage d'assemblage x86 et du système de transitions généré, ce qui garantit que la propriété de sureté vérifiée par Cubicle peut être transposée dans le programme assembleur original.

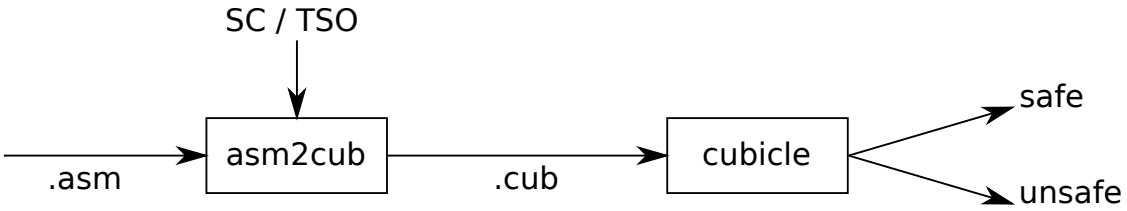


Figure 2.1: Schéma de notre approche

2.1 Sous-ensemble du language supporté

Le sous-ensemble précis du langage supporté par notre outil est décrit en Annexe 1. Toutefois, par soucis de simplicité, nous utiliserons un langage encore plus restreint dans ce rapport. Ce langage permet tout de même de capturer l'essentiel de la complexité du problème des *modèles mémoire faibles*.

integer, n			entier
register, r			registre
variable, x			variable
label, l			étiquette d'instruction (indice du tableau d'instructions)
thread_id, t			identificateur de thread
instruction, i	::=		
	|	mov *r, x*	lecture depuis la mémoire
	|	mov *x, r*	écriture vers la mémoire
	|	mov *r, n*	chargement d'une constante dans un registre
	|	add *r, r'*	ajout du contenu de deux registres
	|	cmp *r, r'*	comparaison du contenu de deux registres
	|	jmp *l*	branchement inconditionel
	|	je *l*	branchement si égal
	|	jne *l*	branchement si différent
	|	mfence	barrière mémoire (TSO uniquement)
processus, p	::=		
	|	*t* : *i array*	thread (tableau d'instructions)
	|	*p* || *p'*	composition parallèle

2.2 Exemple d'algorithme concurrent

Notre exemple type sera l'algorithme de Peterson, qui permet de protéger une *section critique*. Le code est donnée en langage d'assemblage x86 selon la syntaxe Intel (opérande destination à gauche), et peut être compilé avec NASM.

```
enter_1:                            enter_2:
    mov dword [want1], 1                mov dword [want2], 1
    mov dword [turn], 1                 mov dword [turn], 0
wait_1:                             wait_2:
    cmp dword [want2], 1                cmp dword [want1], 1
    jne sc_1                            jne sc_2
    cmp dword [turn], 1                 cmp dword [turn], 0
    je wait_1                           je wait_2
sc_1:                               sc_2:
    ; critical section                  ; critical section

sc_1_end:                           sc_2_end:
    mov dword [want1], 0                mov dword [want2], 0
```

Figure 2.2: Algorithme de Peterson en assembleur x86

Lorsqu'il est exécuté sur des *modèles mémoire faibles*, notamment TSO, deux *threads* peuvent se retrouver simultanément en *section critique*. En effet, les premières écritures vers les variables *want*1 et *want*2 peuvent être encore en attente dans les *tampons* d'écriture lorsqu'on effectue le test pour rentrer en *section critique* ; chaque *thread* croit donc que l'autre n'a pas encore demandé la *section critique* et décide d'y entrer.

2.3 Expression de la propriété de sûreté

On s'intéresse essentiellement à la propriété d'exclusion mutuelle de l'algorithme de Peterson, et de tout algorithme d'exclusion mutuelle en général. On peut exprimer cette propriété en marquant explicitement dans le code source des différents threads des points de programme dont on veut s'assure qu'ils ne puissent pas être atteints simultanéments par des threads différents.

A cette fin, notre outil reconnait les commentaires `; critical section`, tels qu'ils apparaissent dans l'algorithme de Peterson donné en Figure 2.2. Ces commentaires donneront lieu à l'ajout d'instructions spécifiques dans le système de transition généré, afin d'exprimer la propriété d'exclusion mutuelle.

2.4 Le model checker Cubicle

Le model checker Cubicle[11] permet de décrire des systèmes de transitions grâce à des formules logiques. Une transition est composée de sa garde, condition nécessaire à son activation, et ses actions, qui sont des modifications de variables globales.

On choisit par simplicité de représenter ces transitions par des conjonctions de litéraux, contenant deux types de variables : des variables avec apostrophe et sans apostrophe. Une variable sans apostrophe indique une condition nécessaire pour activer la transition. Une variable avec apostrophe indique un résultat de la transition. Par exemple, la formule

$$\exists i.\ \mathtt{T}[i] = \mathsf{true} \wedge \mathtt{X} \leq 100 \wedge \mathtt{X}' = \mathtt{X} + 1 \wedge \mathtt{T}'[i] = \mathsf{false}$$

représente une transition qui peut être prise s'il existe un *thread* i tel que $T[i]$ est vrai et la valeur de X est inférieure ou égale à 100. L'effet de la transition est d'incrémenter la valeur de X et de passer $T[i]$ à faux.

Le lecteur intéressé peut se référer au site web de Cubicle[1] pour plus d'informations sur l'outil et son langage d'entrée.

CHAPITRE 3

Traduction sous le modèle x86-SC

Pour définir le modèle x86-SC, on commence par définir les états manipulés. Puis, on donne la sémantique des processus et des instructions quui agissent sur ces états. On donne ensuite la traduction des états x86-SC sous Cubicle, puis la traduction des instructions. On nomme la sémantique obtenue Cub86-SC. On réalise enfin la preuve d'équivalence de ces deux sémantiques.

3.1 Définition de la sémantique x86-SC

Représentation des états

Pour représenter les états x86-SC, on sépare la mémoire partagée des états locaux des différents threads. Les états locaux sont constitués pour chaque thread de son compteur de programme, son $zeroflag$, et ses registres. Nos états sont donc ainsi définis :

$S = (LS \times M)$	Un état machine x86-SC
$M = (var \mapsto int)\ map$	Une mémoire : dictionnaire des variables vers des entiers
$LS = (tid \mapsto ls)\ map$	Les états locaux des threads : dictionnaire des identificateurs de threads vers leurs états
$ls = (pc \times zf \times Q)$	Un état local à un thread
$Q = (reg \mapsto int)\ map$	Les registres d'un thread : dictionnaire des noms des registres vers des entiers
$pc = int$	Le compteur de programme d'un thread
$zf = int$	Le *zero flag* d'un thread
var	L'ensemble des variables
tid	L'ensemble des identificateurs de threads
reg	L'ensemble des noms de registres

Le registre pc représente le compteur de programme d'un thread. Il situe le point d'exécution courant d'un thread. On choisit pour le représenter d'utiliser les entiers, par simplicité, bien que l'on effectuera aucune opération arithmétique sur ce registre.

Le registre zf est un registre booléen, destiné à reçevoir le résultat de l'instruction de comparaison, mais plus généralement qui doit être positionné à $vrai$ pour toute instruction arithmétique dont le résultat est 0, et à $faux$ dans le cas contraire. On le représentera par un entier, avec la convention usuelle $0 = faux$ et $1 = vrai$.

La machine x86-SC est initialement dans un état S_{init} tel que les registres pc de tous les threads sont initialisés à 0, et tous les autres registres ainsi que la mémoire sont dans un état indéterminé.

Sémantique des processus

Ici, on considère le programme dans son ensemble. Les transitions sont étiquetées par un (ensemble de) processus (tels que définis dans la syntaxe). Quand par applications successives des règles de contexte, on arrive à un thread unique, on restreint l'état à un état contenant uniquement l'état local du thread restant et la mémoire partagée, et on étiquette la transition avec la séquence d'instruction du thread (règle Thread). Cela permet ensuite d'appliquer les règles relatives aux instructions.

Principe :
$S \xrightarrow{p} S'$ l'état S devient S' par une action du programme p

$$\frac{(LS, M) \xrightarrow{p1} (LS', M')}{(LS, M) \xrightarrow{p1||p2} (LS', M')} \textsc{CtxLeft} \qquad \frac{(LS, M) \xrightarrow{p2} (LS', M')}{(LS, M) \xrightarrow{p1||p2} (LS', M')} \textsc{CtxRight}$$

$$\frac{LS(t) = ls \quad (ls, M) \xrightarrow{I} (ls', M')}{(LS, M) \xrightarrow{t:I} (LS[t \mapsto ls'], M')} \textsc{Thread}$$

Sémantique des instructions

Grâce à la règle règle Thread, on se situe maintenant au niveau d'un unique thread. Les transitions sont étiquetées par le tableau d'instructions du thread. Les états sont réduits à l'état local du thread exécutant l'instruction et à la mémoire partagée (les états locaux des autres threads sont "masqués"). A chaque instruction correspond une ou deux règles, selon les cas.

Principe :
$(ls, M) \xrightarrow{I} (ls', M')$ le thread exécute une instruction de I sur l'état (ls, M) qui devient (ls', M')

$$\frac{I(pc) = \mathsf{mov}\ r, x \quad M(x) = n}{((pc, zf, Q), M) \xrightarrow{I} ((pc + 1, zf, Q[r \mapsto n]), M)} \textsc{Read}$$

$$\frac{I(pc) = \mathsf{mov}\ x, r \quad Q(r) = n}{((pc, zf, Q), M) \xrightarrow{I} ((pc + 1, zf, Q), M[x \mapsto n])} \textsc{Write}$$

$$\frac{I(pc) = \mathsf{mov}\ r, n}{((pc, zf, Q), M) \xrightarrow{I} ((pc + 1, zf, Q[r \mapsto n]), M)} \textsc{Const}$$

$$\frac{I(pc) = \mathsf{add}\ r, r' \quad Q(r) + Q(r') = n}{((pc, zf, Q), M) \xrightarrow{I} ((pc + 1, \mathsf{iszero}(n), Q[r \mapsto n]), M)} \textsc{Add}$$

$$\frac{I(pc) = \mathsf{cmp}\ r, r' \quad Q(r) - Q(r') = n}{((pc, zf, Q), M) \xrightarrow{I} ((pc + 1, \mathsf{iszero}(n), Q), M)} \textsc{Cmp}$$

$$\frac{I(pc) = \mathsf{jmp}\ l}{((pc, zf, Q), M) \xrightarrow{I} ((l, zf, Q), M)} \textsc{Jmp}$$

$$\frac{I(pc) = \mathsf{je}\ l \quad zf = 1}{((pc, zf, Q), M) \xrightarrow{I} ((l, zf, Q), M)} \textsc{JeTrue}$$

$$\frac{I(pc) = \mathsf{je}\ l \quad zf = 0}{((pc, zf, Q), M) \xrightarrow{I} ((pc + 1, zf, Q), M)} \textsc{JeFalse}$$

$$\frac{I(pc) = \mathsf{jne}\ l \quad zf = 0}{((pc, zf, Q), M) \xrightarrow{I} ((l, zf, Q), M)} \textsc{JneTrue}$$

$$\frac{I(pc) = \mathsf{jne}\ l \quad zf = 1}{((pc, zf, Q), M) \xrightarrow{I} ((pc + 1, zf, Q), M)}\ \textsc{JneFalse}$$

On peut également rajouter une règle abstraite pour signifier le fait qu'un thread a atteint la fin de sa séquence d'instruction et ne pourra plus avancer :

$$\frac{pc \geq |I|}{((pc, zf, Q), M) \xrightarrow{I} \not\rightarrow}\ \textsc{End}$$

3.2 Traduction de la sémantique x86-SC vers Cubicle : Cub86-SC

Représentation des états x86-SC sous Cubicle

Un état x86-SC est représenté sous Cubicle par un ensemble de variables correspondant aux variables partagées ainsi qu'aux états locaux de tous les threads du programme.

Les compteurs de programme de chaque thread sont représentés par autant de variables PC que de threads. Il sont d'un type énuméré loc, dont les éléments sont de la forme L_0, ... L_l. Le nombre de ces éléments peut être déterminé statiquement à la compilation : il est fonction de la longueur de la plus longue séquence d'instructions.

Les $zero\ flag$ de chaque thread sont représentés par autant de variables ZF que de threads. A l'inverse de x86, ce ne sont pas des booléens mais des entiers, avec l'équivalence $n = 0 \equiv vrai$ et $n \neq 0 \equiv faux$. Cela vient du fait que la valeur de ce flag dépend du résultat d'une opération arithmétique (cf. add et cmp), et qu'il aurait fallu des transitions supplémentaires pour convertir ce résultat en booléen avec Cubicle. Pour éviter cela, on préfère stocker directement le résultat de la dernière opération arithmétique dans ZF. Cela ne permet toutefois pas de comportements supplémentaires, car les flags sont testés en comparant avec 0, il n'y a donc que deux issues possibles : $vrai$ ou $faux$. On dira que deux flags ZF sont équivalents s'ils sont soit égaux à 0 tous les deux, soit différents de 0 tous les deux. Deux états Cub86-SC sont donc équivalents si toutes les variables autres que ZF sont identiques et si leurs flags ZF sont équivalents.

Les registres de chaque thread sont représentés par autant de variables R que de threads et de registres, et sont de type entier.

Les variables partagées sont représentées par autant de variables X, et sont de type entier.

Un état Cub86-SC est donc de la forme : $\{PC_1, ..., PC_p, ZF_1, ..., ZF_p, R1_1, ..., Rm_p, X1, ..., Xn\}$, où p est le nombre de threads, m le nombre de registres et n le nombre de variables.

L'état initial du système $C_{init} = \mathcal{T}(S_{init})$ est donné par $\{PC_1 = L_0, ..., PC_p = L_0\}$; toutes les autres variables sont dans un état indéterminé.

Traduction des états x86-SC vers Cub86-SC

On définit la fonction de traduction des états $\mathcal{T}$ qui traduit des états x86-SC vers les états Cub86-SC. Cette fonction comporte des sous-fonctions pour traduire les différentes composantes d'un état.

$$
\begin{array}{rcll}
\mathcal{T}(LS, M) & = & \mathcal{T}_{LS}(LS) \cup \mathcal{T}_M(M) & \\
\mathcal{T}_M(M) & = & \{X_1 = M(x_1), ..., X_n = M(x_n)\} & avec\ x_1, ..., x_n \in dom(M) \\
\mathcal{T}_{LS}(LS) & = & \mathcal{T}_{ls}(t_1, LS(t_1)) \cup ... \cup \mathcal{T}_{ls}(t_p, LS(t_p)) & avec\ t_1, ..., t_p \in dom(LS) \\
\mathcal{T}_{ls}(t, (pc, zf, Q)) & = & \{PC_t = L_{pc}, ZF_t = \mathsf{iszero}\,[1](zf), & \\
 & & \quad R1_t = Q(r_1), ..., Rm_t = Q(r_m)\} & avec\ r_1, ..., r_m \in dom(Q)
\end{array}
$$

Exemple :
$\mathcal{T}([t_1 \mapsto (4, 1, [r_1 \mapsto 8, r_2 \mapsto 1]), t_2 \mapsto (2, 0, [r_1 \mapsto 7, r_2 \mapsto 5])], [x \mapsto 4, y \mapsto 2])$
$= \{PC_1 = L_4, ZF_1 = 0, R1_1 = 8, R2_1 = 8, PC_2 = L_2, ZF_2 = 1, R1_2 = 7, R2_2 = 5, X = 4, Y = 2\}$

La fonction $\mathcal{T}$ est réversible, et on écrira $\overline{\mathcal{T}}$pour signifier que l'on traduit d'un état Cub86-SC vers un état x86-SC.

Compilation des instructions vers Cub86-SC

On définit la fonction de compilation $\mathcal{C}$ qui prend en entrée un identificateur de thread, une instruction, et le numéro de l'instruction dans le tableau (équivalent du compteur de programme). La fonction renvoie un ensemble de transitions Cubicle équivalent à l'instruction.

$$
\begin{array}{rcll}
\mathcal{C}(t\ ;\ \mathsf{mov}\ r, x\ ;\ i) & = & \{PC_t = L_i \wedge R'_t = X \wedge PC'_t = L_{i+1}\} & \textsc{TRead} \\
\mathcal{C}(t\ ;\ \mathsf{mov}\ x, r\ ;\ i) & = & \{PC_t = L_i \wedge X' = R_t \wedge PC'_t = L_{i+1}\} & \textsc{TWrite} \\
\mathcal{C}(t\ ;\ \mathsf{mov}\ r, n\ ;\ i) & = & \{PC_t = L_i \wedge R'_t = n \wedge PC'_t = L_{i+1}\} & \textsc{TConst} \\
\mathcal{C}(t\ ;\ \mathsf{add}\ r1, r2\ ;\ i) & = & \{PC_t = L_i \wedge R1'_t = R1_t + R2_t \wedge & \textsc{TAdd} \\
 & & \quad ZF'_t = R1_t + R2_t \wedge PC'_t = L_{i+1}\} & \\
\mathcal{C}(t\ ;\ \mathsf{cmp}\ r1, r2\ ;\ i) & = & \{PC_t = L_i \wedge ZF'_t = R1_t - R2_t \wedge PC'_t = L_{i+1}\} & \textsc{TCmp} \\
\mathcal{C}(t\ ;\ \mathsf{jmp}\ l\ ;\ i) & = & \{PC_t = L_i \wedge PC'_t = L_l\} & \textsc{TJmp} \\
\mathcal{C}(t\ ;\ \mathsf{je}\ l\ ;\ i) & = & \{PC_t = L_i \wedge ZF_t = 0 \wedge PC'_t = L_l\} & \textsc{TJeTrue} \\
 & \cup & \{PC_t = L_i \wedge ZF_t \neq 0 \wedge PC'_t = L_i\} & \textsc{TJeFalse} \\
\mathcal{C}(t\ ;\ \mathsf{jne}\ l\ ;\ i) & = & \{PC_t = L_i \wedge ZF_t \neq 0 \wedge PC'_t = L_l\} & \textsc{TJneTrue} \\
 & \cup & \{PC_t = L_i \wedge ZF_t = 0 \wedge PC'_t = L_i\} & \textsc{TJneFalse}
\end{array}
$$

Par exemple, la première transition requiert que le compteur de programme du thread t soit égal à L_i pour être activée. Cette condition est suffisante pour savoir quelle actions effectuer : en effet, comme l'instruction à la position i dans le tableau d'instructions du thread correspond à $mov\ r,\ x$, la transition a forcément été obtenue par compilation de cette instruction. Les actions de cette transitions consistent à mettre dans la variable R_t la valeur de la variable X, et à mettre à jour le compteur de programme avec l'étiquette L_{i+1}.

Afin de compiler l'ensemble des instructions d'un thread, on définit la fonction de compilation récursive $\mathcal{C}_t$ qui prend en entrée un identificateur de thread, un tableau d'instructions, et l'indice à

1. La fonction iszero renvoie 1 si son argument vaut 0, et 0 dans le cas contraire

partir duquel commencer la compilation. La fonction renvoie un ensemble de transitions Cubicle équivalent à la séquence d'instructions définie par le tableau.

$$\mathcal{C}_t(t\ ;\ I\ ;\ i) \quad = \quad \mathcal{C}(t\ ;\ I(i)\ ;\ i) \cup \mathcal{C}_t(t\ ;\ I\ ;\ i+1)$$

On appelle cette fonction de compilation sur l'ensemble des threads qui composent le programme (et avec initialement $i = 0$ pour chaque thread) afin d'obtenir un système de transitions Cubicle équivalent au programme x86-SC donné en entrée.

Le lecteur intéressé peut consulter l'Annexe 2, qui représente le résultat de la traduction du programme Peterson sur la sémantique x86-SC.

Traduction de la propriété de sûreté

On a vu dans le chapitre précédent que la propriété de sûreté était exprimée par des commentaires spécifiques insérés dans le code source des différents threads. Notre outil collecte pour chaque thread l'ensemble des points de programmes correspondant à ces commentaires, puis génère une ou plusieurs formules permettant d'exprimer l'exclusion mutuelle.

Par exemple, pour un programme avec deux threads t_1 et t_2, et deux points de programme pc_{crit1} dans t_1 et pc_{crit2} dans t_2, la formule générée est la suivante :

$$\{PC_{t1} = L_{crit1} \wedge PC_{t2} = L_{crit2}\}$$

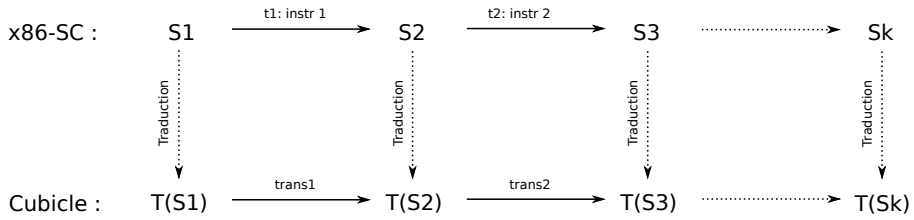
Si cette formule est vraie, alors l'exclusion mutuelle n'est pas assurée.

3.3 Equivalence des sémantiques

On cherche maintenant à prouver que notre schéma de traduction est correct, c'est à dire, que les deux sémantiques sont équivalentes. Pour cela, on cherche à établir une bisimulation entre les programmes x86-SC et leur traduction Cub86-SC. Par simulation avant, on vérifie que tous les comportements des programmes x86-SC sont inclus dans les comportements de leur équivalent Cub86-SC. Puis, par simulation arrière, on vérifie que tous les comportements des programmes Cub86-SCe sont inclus dans les comportements de leurs équivalents x86-TSO.

On note que les règles CtxLeft, CtxRight et Thread matérialisent un ordonnancement de l'exécution des threads entièrement non déterministe, qui correspond naturellement à l'ordonnancement réalisé par Cubicle. On considère donc que l'ordonnancement est équivalents dans les deux sémantiques. Notre preuve ne vérifiera donc pas ces règles.

Simulation avant : x86-SC vers Cub86-SC



Théorème 1. La sémantique Cub86-SC simule la sémantique x86-SC, c'est à dire, $\forall S_1$, si $S_1 \xrightarrow{*} S_k$ alors $\forall C_1 \in \mathcal{T}(S_1)$, $\exists C_k \in \mathcal{T}(S_k)$ tel que $C_1 \xrightarrow{*} C_k$.

Démonstration. On procède par récurrence sur la longueur k de la dérivation $S_1 \xrightarrow{*} S_k$ et analyse par cas sur la première règle appliquée.

Hypothèse de récurrence : on suppose que pour toute dérivation de longueur $k-1$, le Théorème 1 est vrai, c'est à dire qu'on a $\forall C_2 \in \mathcal{T}(S_2), \exists C_k \in \mathcal{T}(S_k)$ tel que $C_2 \xrightarrow{*} C_k$.

On montre alors grâce à une analyse par cas sur les différentes règles que $\forall S_1$, si $S_1 \to S_2$, alors $\forall C_1 \in \mathcal{T}(S_1)$, $\exists C_2 \in \mathcal{T}(S_2)$ tel que $C_1 \to C_2$.

- Cas mov r, x : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q)$ et $M(x) = n$, le thread t exécute l'instruction mov r, x. Par la règle READ, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q[r \mapsto n])], M)$. Depuis l'état Cub86-SC $C_1 = \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$ et $C_1(X) = n$, le système prend la transition TREAD, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, R_t \mapsto n] = \mathcal{T}(S_2)$.

- Cas mov x, r : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q)$ et $Q(r) = n$, le thread t exécute l'instruction mov x, r. Par la règle WRITE, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q)], M[x \mapsto n])$. Depuis l'état Cub86-SC $C_1 = \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$ et $C_1(R_t) = n$, le système prend la transition TWRITE, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, X \mapsto n] = \mathcal{T}(S_2)$.

- Cas mov r, n : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q)$, le thread t exécute l'instruction mov r, n. Par la règle CONST, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q[r \mapsto n])], M)$. Depuis l'état Cub86-SC $C_1 = \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, le système prend la transition TCONST, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, R_t \mapsto n] = \mathcal{T}(S_2)$.

- Cas add $r1, r2$: depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q)$, $Q(r_1) = n_1$ et $Q(r_2) = n_2$, le thread t exécute l'instruction add $r1, r2$. Par la règle ADD, on obtient un état $S_2 = (LS[t \mapsto (pc+1, \mathsf{iszero}(n_1+n_2), Q[r1 \mapsto n_1+n_2])], M)$. Depuis l'état Cub86-SC $C_1 = \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, $C_1(R1_t) = n_1$ et $C_1(R2_t) = n_2$, le système prend la transition TADD, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, ZF_t \mapsto n_1+n_2, R1_t \mapsto n_1+n_2] \equiv \mathcal{T}(S_2)$.

- Cas cmp $r1, r2$: depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q)$, $Q(r_1) = n_1$ et $Q(r_2) = n_2$, le thread t exécute l'instruction cmp $r1, r2$. Par la règle CMP, on obtient un état $S_2 = (LS[t \mapsto (pc+1, \mathsf{iszero}(n_1-n_2), Q)], M)$. Depuis l'état Cub86-SC $C_1 = \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, $C_1(R1_t) = n_1$ et $C_1(R2_t) = n_2$, le système prend la transition TCMP, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, ZF_t \mapsto n_1-n_2] \equiv \mathcal{T}(S_2)$.

- Cas jmp l : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q)$, le thread t exécute l'instruction jmp l. Par la règle JMP, on obtient un état $S_2 = (LS[t \mapsto (l, zf, Q)], M)$. Depuis l'état Cub86-SC $C_1 = \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, le système prend la transition TJMP, et on obtient un état $C_2 = C_1[PC_t \mapsto L_l] = \mathcal{T}(S_2)$.

- Cas je l : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q)$, le thread t exécute l'instruction je l. On a alors deux sous-cas :

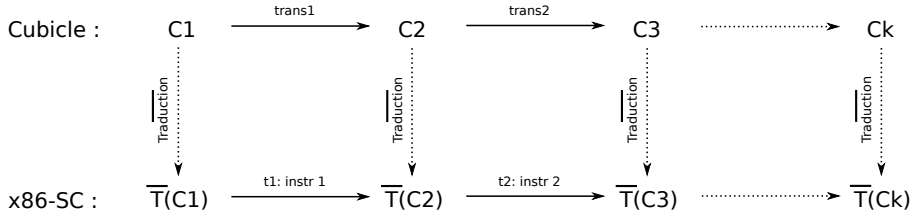
  - Branchement pris quand $zf = 1$: par la règle JETRUE, on obtient un état $S_2 = (LS[t \mapsto (l, zf, Q)], M)$. Depuis l'état Cub86-SC $C_1 = \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, le système prend la transition TJETRUE, et on obtient un état $C_2 = C_1[PC_t \mapsto L_l] = \mathcal{T}(S_2)$.

– Branchement non pris quand $zf = 0$: par la règle JeFalse, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q)], M)$. Depuis l'état Cub86-SC $C_1 = \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, le système prend la transition TJeFalse, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}] = \mathcal{T}(S_2)$.

– Cas jne l : similaire à je l (inversion du flag)

□

Simulation arrière : Cub86-SC vers x86-SC



Théorème 2. La sémantique x86-SC simule la sémantique Cub86-SC, c'est à dire, $\forall C_1$, si $C_1 \xrightarrow{*} C_k$ alors $\forall S_1 \in \overline{\mathcal{T}}(C_1)$, $\exists S_k \in \overline{\mathcal{T}}(C_k)$ tel que $S_1 \xrightarrow{*} S_k$.

Démonstration. On procède par récurrence sur la longueur k de la dérivation $C_1 \xrightarrow{*} C_k$ et analyse par cas sur la première règle appliquée.

Hypothèse de récurrence : on suppose que pour toute dérivation de longueur $k-1$, le Théorème 2 est vrai, c'est à dire qu'on a $\forall S_2 \in \overline{\mathcal{T}}(C_2), \exists S_k \in \overline{\mathcal{T}}(C_k)$ tel que $S_2 \xrightarrow{*} S_k$.

On montre alors grâce à une analyse par cas sur les différentes règles que $\forall C_1$, si $C_1 \to C_2$, alors $\forall S_1 \in \overline{\mathcal{T}}(C_1)$, $\exists S_2 \in \overline{\mathcal{T}}(C_2)$ tel que $S_1 \to S_2$.

– Cas TRead : pour un thread t donné, depuis l'état Cub86-SC C_1 tel que $C_1(PC_t) = L_{pc}$ et $C_1(X) = n$, le système prend la transition TRead, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, R_t \mapsto n]$. Depuis l'état x86-SC $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q)$ et $M(x) = n$, le thread t exécute l'instruction $I(pc) =$ mov r, x. Par la règle Read, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q[r \mapsto n])], M) = \overline{\mathcal{T}}(C_2)$.

– Cas TWrite : pour un thread t donné, depuis l'état Cub86-SC C_1 tel que $C_1(PC_t) = L_{pc}$ et $C_1(R_t) = n$, le système prend la transition TWrite, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, X \mapsto n]$. Depuis l'état x86-SC $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q)$ et $Q(r) = n$, le thread t exécute l'instruction $I(pc) =$ mov x, r. Par la règle Write, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q)], M[x \mapsto n]) = \overline{\mathcal{T}}(C_2)$.

– Cas TConst : pour un thread t donné, depuis l'état Cub86-SC C_1 tel que $C_1(PC_t) = L_{pc}$, le système prend la transition TConst, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, R_t \mapsto n]$. Depuis l'état x86-SC $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q)$, le thread t exécute l'instruction $I(pc) =$ mov r, n. Par la règle Const, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q[r \mapsto n])], M) = \overline{\mathcal{T}}(C_2)$.

– Cas TAdd : pour un thread t donné, depuis l'état Cub86-SC C_1 tel que $C_1(PC_t) = L_{pc}$, $C_1(R1_t) = n_1$ et $C_2(R2_t) = n_2$, le système prend la transition TAdd, et on obtient

un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, ZF_t \mapsto n_1 + n_2, R1_t \mapsto n_1 + n_2]$. Depuis l'état x86-SC $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q)$, $Q(r_1) = n_1$ et $Q(r_2) = n_2$, le thread t exécute l'instruction $I(pc) = \mathsf{add}\ r1, r2$. Par la règle ADD, on obtient un état $S_2 = (LS[t \mapsto (pc + 1, \mathsf{iszero}(n_1 + n_2), Q[r1 \mapsto n_1 + n_2])], M) = \overline{\mathcal{T}}(C_2)$.

– Cas TCMP : pour un thread t donné, depuis l'état Cub86-SC C_1 tel que $C_1(PC_t) = L_{pc}$, $C_1(R1_t) = n_1$ et $C_2(R2_t) = n_2$, le système prend la transition TCMP, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, ZF_t \mapsto n_1 - n_2]$. Depuis l'état x86-SC $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q)$, $Q(r_1) = n_1$ et $Q(r_2) = n_2$, le thread t exécute l'instruction $I(pc) = \mathsf{cmp}\ r1, r2$. Par la règle CMP, on obtient un état $S_2 = (LS[t \mapsto (pc + 1, \mathsf{iszero}(n_1 - n_2), Q)], M) = \overline{\mathcal{T}}(C_2)$.

– Cas TJMP : pour un thread t donné, depuis l'état Cub86-SC C_1 tel que $C_1(PC_t) = L_{pc}$, le système prend la transition TJMP, et on obtient un état $C_2 = C_1[PC_t \mapsto L_l]$. Depuis l'état x86-SC $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q)$, le thread t exécute l'instruction $I(pc) = \mathsf{jmp}\ l$. Par la règle JMP, on obtient un état $S_2 = (LS[t \mapsto (l, zf, Q)], M) = \overline{\mathcal{T}}(C_2)$.

– Cas TJETRUE : pour un thread t donné, depuis l'état Cub86-SC C_1 tel que $C_1(PC_t) = L_{pc}$ et $C_1(ZF_t) = 0$, le système prend la transition TJETRUE, et on obtient un état $C_2 = C_1[PC_t \mapsto L_l]$. Depuis l'état x86-SC $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q)$ et $zf = 1$, le thread t exécute l'instruction $I(pc) = \mathsf{je}\ l$. Comme $zf = 1$, c'est la règle JETRUE qui s'applique, et on obtient un état $S_2 = (LS[t \mapsto (l, zf, Q)], M) = \overline{\mathcal{T}}(C_2)$.

– Cas TJEFALSE : pour un thread t donné, depuis l'état Cub86-SC C_1 tel que $C_1(PC_t) = L_{pc}$ et $C_1(ZF_t) \neq 0$, le système prend la transition TJEFALSE et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}]$. Depuis l'état x86-SC $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q)$ et $zf = 0$, le thread t exécute l'instruction $I(pc) = \mathsf{je}\ l$. Comme $zf = 0$, c'est la règle JEFALSE qui s'applique, et on obtient un état $S_2 = (LS[t \mapsto (pc + 1, zf, Q)], M) = \overline{\mathcal{T}}(C_2)$.

– Cas TJNETRUE : similaire à TJETRUE (inversion du flag)

– Cas TJNEFALSE : similaire à TJEFALSE (inversion du flag)

□

CHAPITRE 4

Traduction sous le modèle x86-TSO

Pour définir la modèle x86-TSO, on se base sur le modèle x86-SC. En effet, une grande partie des définitions est commune aux deux sémantiques. On ne présentera donc dans ce chapitre que les aspects de x86-TSO qui diffèrent de x86-SC.

4.1 Aperçu du modèle x86-TSO

Le modèle x86-TSO est représenté par la machine abstraite donnée en Figure 4.1. On dispose d'une mémoire partagée par tous les threads. Chaque thread dispose par ailleurs d'un tampon d'écriture qui lui est propre. Lorsqu'un thread veut écrire en mémoire, l'écriture est systématiquement placée dans son tampon. De façon asynchrone et non déterministe, les écritures en attente dans les tampons sont propagées en mémoire (on appelle cela "synchronisation"). Lorsqu'un thread veut lire depuis la mémoire, il commence par vérifier si la variable demandée est présente dans son tampon, auquel cas il lit la valeur la plus récente pour cette variable depuis le tampon. Si la variable n'est pas présente dans son tampon, il va la chercher directement en mémoire. On dispose également d'une instruction mfence, qui lorsqu'elle est utilisée, bloque le thread qui l'exécute jusqu'à ce que son tampon soit vide. L'utilisation judicieuse de cette primitive permet de "réparer" des algorithmes fonctionnant sous le modèle SC et cassés sous le modèle TSO. On dispose par ailleurs d'un mécanisme de verrou global, permettant de donner un comportement atomique aux instructions de type *lecture-modification-écriture*. Dans notre approche, on ne s'intéressera pas à ces verrous, mais uniquement aux tampons et à l'instruction mfence.

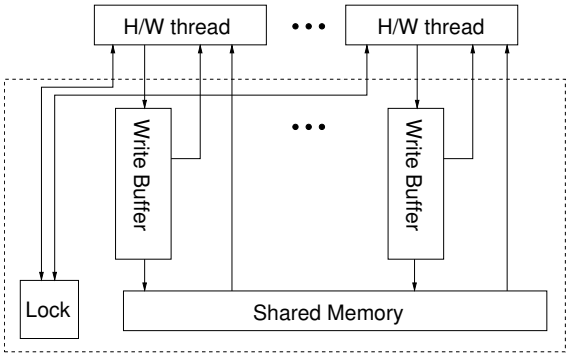


Figure 4.1: Machine abstraite du modèle x86-TSO[16]

4.2 Les tampons TSO

Un tampon TSO peut être vu comme une sorte de file FIFO contenant des paires $(var \times int)$. On prendra comme convention que la tête de la file se situe à gauche, et la queue à droite. Une opération d'écriture place en tête de file la variable et la valeur à écrire. Une opération de synchronisation retire de la queue de la file une variable et une valeur, et met à jour la variable en mémoire avec la valeur. Il y a cependant quelques différences avec une vraie file FIFO. Premièrement, on dispose d'une d'une opération en lecture seule permettant de lire l'occurrence la plus récente d'une variable

dans la file, ce qui permet de réaliser l'opération de lecture depuis le tampon. Deuxièmement, on dispose d'une opération en lecture seule permettant de tester si une variable est présente ou non dans le tampon.

Définition 1. Un tampon TSO est une file FIFO supportant les opérations suivantes :
- is_empty : vérifie que la file est vide
- not_in : vérifie que qu'une variable n'est pas dans la file [1]
- peek : récupère l'occurence la plus récente d'une variable dans la file
- push_front : ajoute une variable en tête de file
- pop_back : récupère et retire la variable en fin de file

4.3 Définition de la sémantique x86-TSO

Représentation des états

Les états sont similaires à x86-SC, la seule différence étant l'ajout des tampons d'écriture dans les états locaux de chaque thread.

$S = (LS \times M)$	Un état machine x86-TSO
$M = (var \mapsto int)\ map$	Une mémoire : dictionnaire des variables vers des entiers
$LS = (tid \mapsto ls)\ map$	Les états locaux des threads : dictionnaire des identificateurs de threads vers leurs états
$ls = (pc \times zf \times Q \times B)$	Un état local à un thread
$Q = (reg \mapsto int)\ map$	Les registres d'un thread : dictionnaire des noms des registres vers des entiers
$B = (var \times int)\ queue$	Le tampon d'écriture TSO d'un thread
$pc = int$	Le compteur de programme d'un thread
$zf = int$	Le *zero flag* d'un thread
var	L'ensemble des variables
tid	L'ensemble des identificateurs de threads
reg	L'ensemble des noms de registres

La machine x86-TSO est initialement dans un état S_{init} tel que les registres pc de tous les threads sont initialisés à 0, les tampons de tous les threads sont vides, et tous les autres registres ainsi que la mémoire sont dans un état indéterminé.

Notations pour la manipulation des tampons TSO

Au niveau de la sémantique x86-TSO, on utilise les notations suivantes pour manipuler les tampons :

$x \in B$	Vrai si au moins une paire contenue dans B concerne x
$x \notin B$	Vrai si aucune paire contenue dans B ne concerne x
$B = \varnothing$	Vrai si B est vide
$B_1 +\!\!+ B_2$	Concaténation de B_1 et B_2
$(x, n) +\!\!+ B$	Apposition de (x, n) en tête de B
$B +\!\!+ (x, n)$	Apposition de (x, n) en queue de B

Ces notations permettent d'exprimer les opérations décrites par la Définition 1.

1. on définit l'opération comme un test de non-appartenance, pour simplifier la traduction sous Cubicle

Sémantique des instructions

La plupart des instructions ont la même sémantique qu'avec x86-SC, on peut donc garder les mêmes règles (en se contentant de laisser les tampons non modifiés). Seules les instructions de lecture et d'écriture en mémoire ont une sémantique différente. Il y a par ailleurs une instruction supplémentaire : mfence.

$$\frac{I(pc) = \mathsf{mov}\ r, x \quad x \notin B \quad M(x) = n}{((pc, zf, Q, B), M) \xrightarrow{I} ((pc+1, zf, Q[r \mapsto n], B), M)}\ \textsc{ReadMem}$$

$$\frac{I(pc) = \mathsf{mov}\ r, x \quad B = B_1 +\!\!+ (x, n) +\!\!+ B_2 \quad x \notin B_1}{((pc, zf, Q, B), M) \xrightarrow{I} ((pc+1, zf, Q[r \mapsto n], B), M)}\ \textsc{ReadBuf}$$

$$\frac{I(pc) = \mathsf{mov}\ x, r \quad Q(r) = n}{((pc, zf, Q, B), M) \xrightarrow{I} ((pc+1, zf, Q, (x, n) +\!\!+ B), M)}\ \textsc{WriteBuf}$$

$$\frac{I(pc) = \mathsf{mfence} \quad B = \varnothing}{((pc, zf, Q, B), M) \xrightarrow{I} ((pc+1, zf, Q, B), M)}\ \textsc{Mfence}$$

Synchronisation tampon / mémoire

De façon asynchrone, un tampon peut basculer la plus ancienne des écritures qu'il contient vers la mémoire. On peut l'exprimer grâce à une règle qui ne dépend pas des registres *pc* mais uniquement de l'état des tampons.

$$\frac{\exists t.\ LS(t) = (pc, zf, Q, B +\!\!+ (x, n))}{(LS, M) \to (LS[t \mapsto (pc, zf, Q, B)], M[x \mapsto n])}\ \textsc{WriteMem}$$

4.4 Les tampons TSO sous Cubicle

Représentation des tampons TSO sous Cubicle

Cubicle ne proposant pas de type *file*, on choisit de représenter un tampon TSO grâce à un tableau B dont les cases contiennent des *records* à deux champs [2].

Le premier champ X, indique s'il y a une écriture en attente ou non et quelle est la variable concernée. On utilise pour cela un type énuméré dont les valeurs sont de la forme V_{Xn}. Il y aura autant de valeurs que de variables partagées dans le programme (on peut les déterminer statiquement à la compilation). On ajoute également la valeur *Empty*, pour matérialiser une case du tampon vide. Initialement, les tampons sont vides, donc $\forall j.\ B[j].X = Empty$. Le second champ V indique la valeur à écrire ; il est donc de type entier.

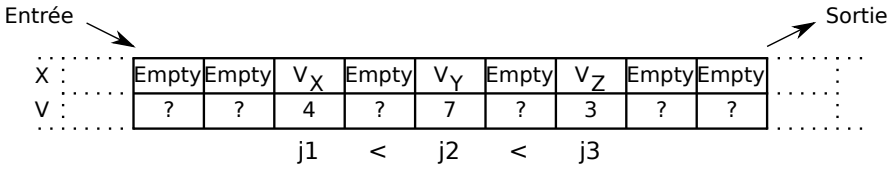
En x86-TSO, les tampons peuvent être d'une taille *arbitrairement grande*. Pour exprimer cela sous Cubicle, les tableaux sont indicés par un type infini (le type proc) pour lequel on dispose d'une relation d'ordre. Algorithmiquement, on utilise habituellement un tableau circulaire, avec deux pointeurs vers les éléments en tête et en queue. L'ajout ou le retrait d'éléments dans la file se fait en incrémentant les pointeurs. Toutefois, le type proc ne dispose pas de l'opération successeur nécessaire pour cet incrémentation. On pourrait tout à fait simuler le successeur grâce à des formules logiques,

2. on utilise des tableaux de *records* par simplicité, mais Cubicle ne supportant pas les *records*, il faudra les représenter par autant de tableaux que de champs

mais les formules engendrées seraient alors particulièrement complexes, et leur analyse par Cubicle particulièrement couteuse. On préfère donc donc adopter une représention plus relâchée des files, où l'on s'autorise la présence de trous dans les tableaux. Cela ne pose aucun problème, puisqu'on a seulement besoin de situer les éléments les uns par rapport aux autres pour que le tampon se comporte correctement. On peut ici faire une analogie avec une file d'attente de personnes dans une boulangerie : si un client quitte la file, on a un "trou" ; si au moment de servir un client, le vendeur rencontre un "trou", il passe simplement au client suivant. Bien entendu, dans notre cas, les "trous" ne peuvent être créés qu'à l'ajout d'un élément. Plus formellement :

Définition 2. Une file ou tampon TSO est un tableau B qui est soit :
- vide : $\forall j.\ B[j].X = Empty$
- non vide, et on a :
  - un élément en tête à l'indice j_{head} tel que $B[j_{head}].X \neq Empty$ et $\forall j.\ j < j_{head} \implies B[j].X = Empty$
  - un élément en queue à l'indice j_{tail} tel que $B[j_{tail}].X \neq Empty$ et $\forall j.\ j_{tail} < j \implies B[j].X = Empty$
  - $j_{head} \leq j_{tail}$

Le schéma suivant montre un exemple d'une telle file, représentant le tampon TSO $[(x,4),(y,7),(z,3)]$:



Encodage des opérations sur les tampons TSO sous Cubicle

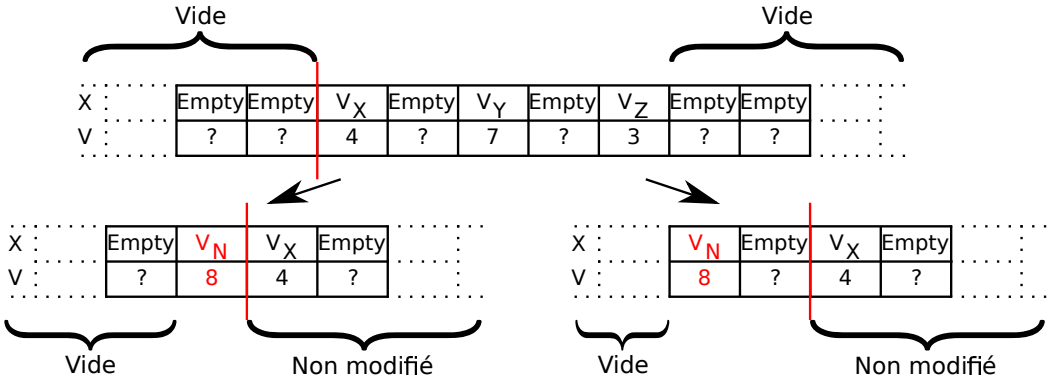
Les opérations décrites par la Définition 1 peuvent être réalisées sous Cubicle grâce aux formules suivantes [3] :

$$\begin{aligned}
\mathsf{is_empty}\ B &\triangleq \{\forall j.\ B[j].X = Empty\} \\
\mathsf{not_in}\ B\ V_X &\triangleq \{\forall j.\ B[j].X \neq V_X\} \\
\mathsf{peek}\ B\ V_X\ R &\triangleq \{\exists j_1.\ B[j_1].X = V_X \wedge \\
&\quad \forall j_2.\ (j_2 < j_1 \implies B[j_2].X \neq V_X) \wedge \\
&\quad R' = B[j_1].V\} \\
\mathsf{push_front}\ B\ V_X\ R &\triangleq \{\exists j_1.\ B[j_1].X = Empty \wedge \\
&\quad \forall j_2.\ (j_2 < j_1 \implies B[j_2].X = Empty) \wedge \\
&\quad B[j_1].X' = V_X \wedge B[j_1].V' = R\} \\
\mathsf{pop_back}\ B\ V_X\ R &\triangleq \{\exists j_1.\ B[j_1].X = V_X \wedge \\
&\quad \forall j_2.\ (j_1 < j_2 \implies B[j_2].X = Empty) \wedge \\
&\quad B[j_1].X' = Empty \wedge R' = B[j_1].V\}
\end{aligned}$$

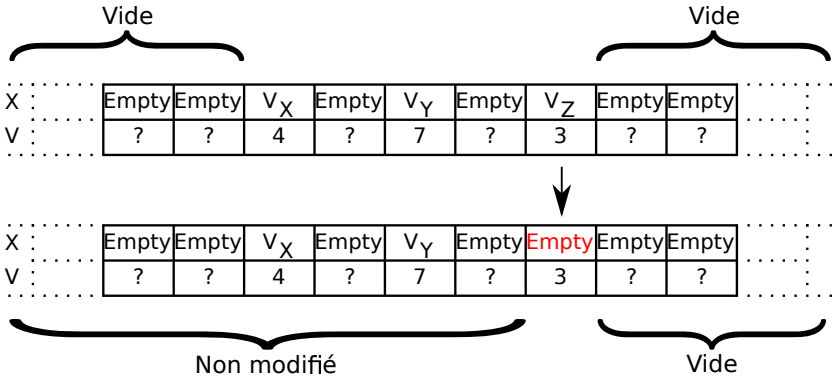
Le schéma suivant représente deux résultats possibles de l'opération push_front. Partant de la file du dessus, avec V_X la variable en tête de file, on peut insérer la nouvelle variable V_N n'importe où à gauche du trait rouge (la formule indique qu'il faut une case vide telle que toutes les autres cases à gauche soient elles aussi vides). On peut par exemple l'insérer immédiatement à gauche, et on obtient la file en bas à gauche. Ou on peut l'insérer en laissant une case vide entre lui et

3. les formules doivent être écrites en CNF sous Cubicle, $A \implies B$ s'écrira $\neg A \vee B$

l'élement en tête de file, et on obtient la file en bas à droite. Dans tous les cas, tous les éléments à gauche de la nouvelle tête de file sont bien vides, et les éléménts à droite restent inchangés.



Le schéma suivant représente le résultat de l'opération pop_back. Partant de la file du dessus, on peut retirer la variable V_Z depuis la fin de file (il s'agit de la variable telle que toutes les autres cases à droite sont vides). On obtient la file en bas, avec V_Y comme nouvelle fin de file. En effet, tous les éléments à droite de V_Y sont bien vides, et les éléménts à gauche sont inchangés.



Propriétés des tampons TSO sous Cubicle

On a besoin pour la suite de garantir que les tampons TSO sous Cubicle possèdent bien certaines propriétés de préservation des éléments. On énonce ces propriétés, suivies de leurs démonstrations. Les démonstrations en question sont simples à réaliser, il s'agit d'une simple inspection des règles.

Lemme 1. L'opération push_front ne modifie aucun élément à droite de j_{head} (c'est à dire les éléments d'indice j tel que $j_{head} < j$)

Démonstration.

- depuis une file vide : l'élément j_{head} n'existant pas, le lemme est vrai par défaut
- depuis une file non-vide : push_front modifie une et une seule case, à l'indice $j_{newhead}$, telle que $j_{newhead}$ est vide et toutes les case à sa gauche le sont également. Puisqu'on part d'une file conforme avec au moins un élément, on a forcément $j_{newhead} < j_{head} \leq j_{tail}$. On a donc que les cases à droite de j_{head} ne sont pas modifiées.

□

Lemme 2. L'opération pop_back ne modifie aucun élément (strictement) à gauche de j_{tail} (c'est à dire les éléments d'indice j tel que $j < j_{tail}$)

Démonstration.
- depuis une file vide : l'opération ne s'appliquant pas, le lemme est vrai par défaut
- depuis une file non-vide : pop_back modifie une et une seule case, qui doit être non vide et telle que toutes les case à sa droite sont vides. Puisqu'on part d'une file conforme avec au moins un élément, la seule case correspondant à cette description est précisément la case j_{tail}. Cette case étant la seule case modifiée, on a donc naturellement que les cases à gauche de j_{tail} ne sont pas modifiées.

□

Lemme 3. Les opérations sur les files telles que décrites par la Définition 1 préservent les éléments entre j_{head} et j_{tail}

Démonstration.
- opérations is_empty, not_in et peek : ces opérations ne modifiant pas la file, les éléments entre j_{head} et j_{tail} sont préservés
- opération push_front :
  - depuis une file vide : la file ne contenant pas d'élément j_{head}, le lemme est vrai par défaut
  - depuis une file non-vide : le Lemme 1 garantit que l'on écrit jamais à droite de j_{head}, et puisque $j_{head} < j_{tail}$, les éléments entre j_{head} et j_{tail} sont bien préservés
- opération pop_back :
  - depuis une file vide : l'opération ne s'appliquant pas, le lemme est vrai par défaut
  - depuis une file non-vide : le Lemme 2 garantit que l'on écrit jamais à gauche de j_{tail}, et puisque $j_{head} < j_{tail}$, les éléments entre j_{head} et j_{tail} sont bien préservés

□

Equivalence des tampons TSO sous Cubicle

Puisque notre définition des tampons TSO sous Cubicle autorise des trous dans les tampons, un même tampon TSO peut donc avoir plusieurs traductions Cubicle équivalentes, selon les emplacements des trous. On a donc besoin de définir une relation d'équivalence, binaire, réflexive, transitive et symmétrique, qui indique si deux tampons TSO sous Cubicle sont équivalents ou non. Cette relation définit une classe d'équivalence pour les tampons.

Pour définir cette relation, on s'aide d'une fonction $filter\_non\_empty$, qui prend en entrée une tampon TSO Cubicle, et renvoie l'unique séquence ordonnée des éléments non-vides qu'il contient. On définit alors l'équivalence de deux tampons $B1$ et $B2$ comme suit :

$$B1 \equiv B2 \quad \triangleq \quad filter\_non\_empty(B1) = filter\_non\_empty(B2)$$

4.5 Traduction de la sémantique x86-TSO vers Cubicle : Cub86-TSO

Représentation des états x86-TSO sous Cubicle

On reprend les états Cub86-SC définis précédement pour x86-SC, auxquels on ajoute les tampons de chaque thread. Les états sont maintenant de la forme :
$\{PC_1, ..., PC_p, ZF_1, ..., ZF_p, R1_1, ..., Rm_p, X1, ..., Xn, B_1[j_1], ..., B_p[j_{kp}]\}$, où p est le nombre de processus, m le nombre de registres, n le nombre de variables, j sont les indices des tampons d'écriture et k les tailles de chaque tampon.

L'état initial du système $C_{init} = \mathcal{T}(S_{init})$ est donné par $\{PC_1 = L_0, ..., PC_p = L_0, \forall j.B_1[j].X = Empty, ..., B_p[j].X = Empty\}$.

Traduction des états x86-TSO vers Cub86-TSO

On définit la fonction de traduction des états $\mathcal{T}$ qui traduit des états x86-TSO vers les états Cub86-TSO :

$$
\begin{array}{rcll}
\mathcal{T}(LS, M) & = & \mathcal{T}_{LS}(LS) \cup \mathcal{T}_M(M) & \\
\mathcal{T}_M(M) & = & \{X_1 = M(x_1), ..., X_n = M(x_n)\} & avec\ x_1, ..., x_n \in dom(M) \\
\mathcal{T}_{LS}(LS) & = & \mathcal{T}_{ls}(t_1, LS(t_1)) \cup ... \cup \mathcal{T}_{ls}(t_p, LS(t_p)) & avec\ t_1, ..., t_p \in dom(LS) \\
\mathcal{T}_{ls}(t, (pc, zf, Q, B)) & = & \{PC_t = L_{pc}, ZF_t = \mathsf{iszero}(zf), & \\
 & & \quad R1_t = Q(r_1), ..., Rm_t = Q(r_m)\} \cup \mathcal{T}_B(t, B) & avec\ r_1, ..., r_m \in dom(Q) \\
 & & \text{Soient } j_1, j_2, ..., j_k \text{ des indices tels que } j_1 < j_2 < ... < j_k, & \\
\mathcal{T}_B(t, B)) & = & \{B_t[j_1].X = V_{fst(B(1))}, ..., BX_t[j_k] = V_{fst(B(k))}, & \\
 & & \quad B_t[j_1].V = snd(B(1)), ..., BV_t[j_k] = snd(B(k))\} \cup & \\
 & & \{B_t[j].X = Empty \mid \forall j.\ j \neq j_1 \wedge ... \wedge j_k\} &
\end{array}
$$

Les indices $j_1, j_2, ..., j_k$ étant choisis arbitrairement, cette dernière fonction $\mathcal{T}_B$ peut donner plusieurs traductions équivalentes d'un même tampon TSO. De ce fait, la fonction $\mathcal{T}$ peut donner plusieurs traductions équivalentes d'un même état x86-TSO. On dira que deux états Cub86-TSO sont équivalents si toutes leurs variables autres que B_t sont identiques et si leurs leurs tampons B_t sont équivalents selon la relation définie en section précédente. On considèrera donc qu'on manipule des classes d'équivalence d'états, plutôt que des états individuels.

Compilation des instructions vers Cub86-TSO

La plupart des instructions se compilent comme en x86-SC. Seules les instructions effectuant des accès mémoire ont une traduction spécifique. Plus exactement, leur traduction est obtenue en combinant la transition x86-SC correspondant et une opération sur le tampon TSO. On définit la fonction de compilation $\mathcal{C}$ qui prend en entrée un identificateur de thread, une instruction, et le numéro de l'instruction dans le tableau (équivalent du compteur de programme). La fonction renvoie un ensemble de transitions Cubicle équivalent à l'instruction.

$$
\begin{array}{rcll}
\mathcal{C}(t\ ;\ \mathsf{mov}\ r, x\ ;\ i) & = & \{PC_t = L_i \wedge \forall j.\ B_t[j].X \neq V_X \wedge & \text{TReadMem} \\
 & & \quad R'_t = X \wedge PC'_t = L_{i+1}\} \cup & = \text{TRead} + \mathsf{not\_in} \\
 & & \{PC_t = L_i \wedge \exists j_1.\ B_t[j_1].X = V_X \wedge & \text{TReadBuf} \\
 & & \quad \forall j_2.\ (j_2 < j_1 \implies B_t[j_2].X \neq V_X) \wedge & = \text{TRead} + \mathsf{peek} \\
 & & \quad R'_t = B_t[j_1].V \wedge PC'_t = L_{i+1}\} & \\
\mathcal{C}(t\ ;\ \mathsf{mov}\ x, r\ ;\ i) & = & \{PC_t = L_i \wedge \exists j_1.\ B_t[j_1].X = Empty \wedge & \text{TWriteBuf} \\
 & & \quad \forall j_2.\ (j_2 < j_1 \implies B_t[j_2].X = Empty) \wedge & = \text{TWrite} + \mathsf{push\_front} \\
 & & \quad B_t[j_1].X' = V_X \wedge B_t[j_1].V' = R_t \wedge & \\
 & & \quad PC'_t = L_{i+1}\} & \\
\mathcal{C}(t\ ;\ \mathsf{mfence}\ ;\ i) & = & \{PC_t = L_i \wedge \forall j.\ B_t[j].X = Empty \wedge & \text{TMfence} \\
 & & \quad PC'_t = L_{i+1}\} & = \mathsf{is\_empty}
\end{array}
$$

Compilation de la synchronisation mémoire

On définit une fonction de compilation $\mathcal{C}_S$, qui prend en paramètre un identificateur de thread et une variable. La fonction génère la transition permettant le vidage de la variable spécifiée depuis le tampon vers la mémoire. Il s'agit de la combinaison de la transition x86-SC TWrite et de

l'opération pop_back sur le tampon TSO. Cette fonction de compilation doit être appelée pour tous les threads du programme et sur toutes les variable partagées (élément connu à la compilation).

$$\begin{array}{lcll} \mathcal{C}_S(t\ ;\ x) & = & \{\exists j_1.\ B_t[j_1].X = V_X \wedge & \textsc{TWriteMem} \\ & & \forall j_2.\ (j_1 < j_2 \implies B_t[j_2].X = Empty) \wedge & = \textsc{TWrite} + \mathsf{pop\_back} \\ & & X' = B_t[j_1].V \wedge B_t[j_1].X' = Empty\} & \end{array}$$

Le lecteur intéressé peut consulter l'Annexe 3, qui représente le résultat de la traduction du programme Peterson sur la sémantique x86-TSO.

Propriété des transitions obtenues

Théorème 3. Les transitions Cub86-TSO obtenues par traduction des instructions x86-TSO préservent les éléments entre j_{head} et j_{tail} dans les files / tampons TSO

Démonstration. Les transitions Cub86-TSO générées ayant été obtenues en combinant les transitions Cub86-SC avec les opérations sur les files, elles ne manipulent les tampons TSO que par l'interface conforme définie par le Lemme 3. De ce fait, elles préservent les éléments entre j_{head} et j_{tail}. □

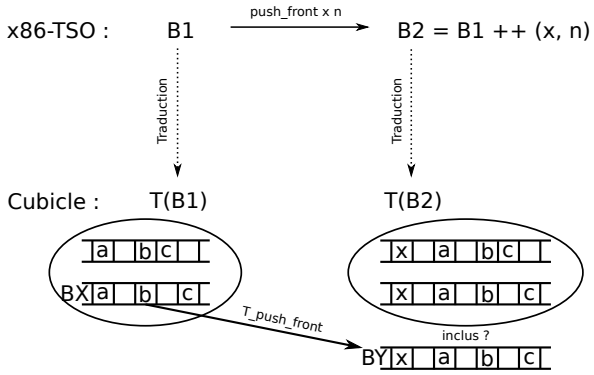
4.6 Equivalence des sémantiques

Pour prouver la correction de notre traduction, on se base sur le même schéma de preuve que précédement, qui consiste à établir une bissimulation entre les deux sémantiques, en prenant en compte la possibilité qu'une sychronisation mémoire ait lieu à n'importe quel moment.

Prérequis : équivalence des tampons TSO

La présence de tampons TSO rend la preuve plus délicate. En effet, partant d'un tampon et de sa traduction sous Cubicle, on veut s'assurer de l'équivalence des tampons obtenus lorsque le système avance d'un pas de chaque côté de la sémantique.

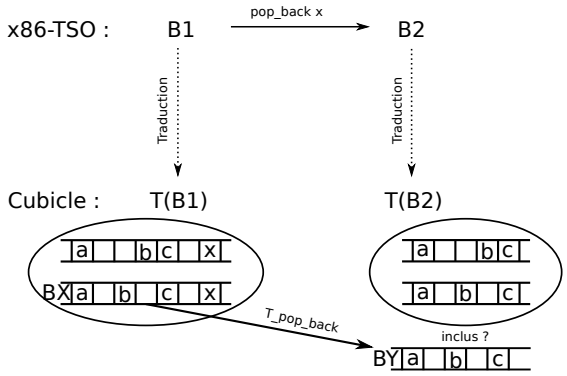
Prenons le cas de l'opération push_front :



On part d'un tampon TSO $B1$, qui se traduit en un tampon BX quelconque parmi l'ensemble $\mathcal{T}(B1)$. Si on applique l'opération push_front sur $B1$, on obtient un tampon $B2$. Lorsqu'on applique la transition T_push_front sur BX, on obtient un tampon BY. On veut montrer que le tampon BY est bien inclus dans l'ensemble des traductions possibles pour $B2$, soit $\mathcal{T}(B2)$.

Grâce au Lemme 3, on sait que tous les éléments de BY situés après la tête sont rigoureusement égaux à ceux de BX situés aux mêmes indices. BY ne contient qu'un seul élément en plus, situé en tête. Le tampon $B2$ est égal au tampon $B1$ auquel on a ajouté un élément en tête. Etant donnée notre fonction de traduction $\mathcal{T}$, les éléments contenus dans l'ensemble $\mathcal{T}(B2)$ sont donc de la même forme que les éléments de $\mathcal{T}(B1)$, avec un élément en plus en tête. Cet ensemble contient donc forcément BY.

Pour ce qui est de l'opération pop_back :

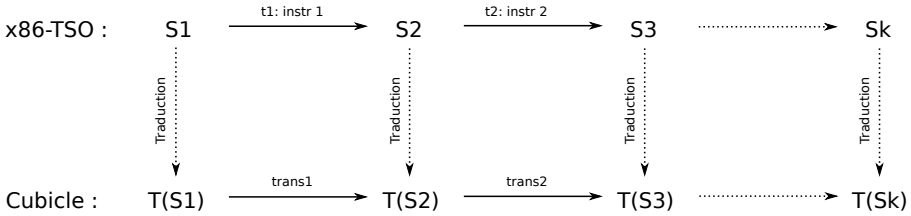


On part d'un tampon TSO $B1$, qui se traduit en un tampon BX quelconque parmi l'ensemble $\mathcal{T}(B1)$. Si on applique l'opération pop_back sur $B1$, on obtient un tampon $B2$. Lorsqu'on applique la transition T_pop_back sur BX, on obtient un tampon BY. On veut montrer que le tampon BY est bien inclus dans l'ensemble des traductions possibles pour $B2$, soit $\mathcal{T}(B2)$.

Grâce au Lemme 3, on sait que tous les éléments de BY sont rigoureusement égaux à ceux de BX situés aux mêmes indices, à l'exception de la queue de BX, qui est vide dans BY. Le tampon $B2$ est égal au tampon $B1$ auquel on a retiré l'élément en queue. Etant donnée notre fonction de traduction $\mathcal{T}$, les éléments contenus dans l'ensemble $\mathcal{T}(B2)$ sont donc de la même forme que les éléments de $\mathcal{T}(B1)$, avec l'élément de queue en moins. Cet ensemble contient donc forcément BY.

Grâce à cette simulation de la sémantique des tampons TSO par Cubicle, on peut maintenant montrer facilement la bissimulation entre les sémantiques x86-TSO et Cub86-TSO.

Simulation avant : x86-TSO vers Cub86-TSO



Théorème 4. La sémantique Cub86-TSO simule la sémantique x86-TSO, c'est à dire, $\forall S_1$, si $S_1 \xrightarrow{*} S_k$ alors $\forall C_1 \in \mathcal{T}(S_1)$, $\exists C_k \in \mathcal{T}(S_k)$ tel que $C_1 \xrightarrow{*} C_k$.

Démonstration. On procède par récurrence sur la longueur k de la dérivation $S_1 \xrightarrow{*} S_k$ et analyse par cas sur la première règle appliquée.

Hypothèse de récurrence : on suppose que pour toute dérivation de longueur $k-1$, le Théorème 4 est vrai, c'est à dire qu'on a $\forall C_2 \in \mathcal{T}(S_2), \exists C_k \in \mathcal{T}(S_k)$ tel que $C_2 \xrightarrow{*} C_k$.

On montre alors grâce à une analyse par cas sur les différentes règles que $\forall S_1$, si $S_1 \to S_2$, alors $\forall C_1 \in \mathcal{T}(S_1)$, $\exists C_2 \in \mathcal{T}(S_2)$ tel que $C_1 \to C_2$.

- Cas mov r, x : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q, B)$, le thread t exécute l'instruction mov r, x. On a alors deux sous-cas :

  - $x \notin B$: on a $M(x) = n$, on applique la règle ReadMem, et on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q[r \mapsto n], B)], M)$. Depuis tout état Cub86-TSO $C_1 \equiv \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, $C_1(X) = n$ et $\forall j.\ C_1(B_t[j].X) \neq V_X$, le système prend la transition TReadMem, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, R_t \mapsto n] \equiv \mathcal{T}(S_2)$.

  - $x \in B$: on a $B = B_1 +\!\!+ (x, n) +\!\!+ B_2$ avec $x \notin B_1$, on applique la règle ReadBuf, et on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q[r \mapsto n], B)], M)$. Depuis tout état Cub86-TSO $C_1 \equiv \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, $C_1(B_t[j_1].X) = V_X$, $C_1(B_t[j_1].V) = n$ et $\forall j_2.\ j_2 < j_1 \implies C_1(B_t[j_2].X) \neq V_X$, le système prend la transition TReadBuf, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, R_t \mapsto n] \equiv \mathcal{T}(S_2)$.

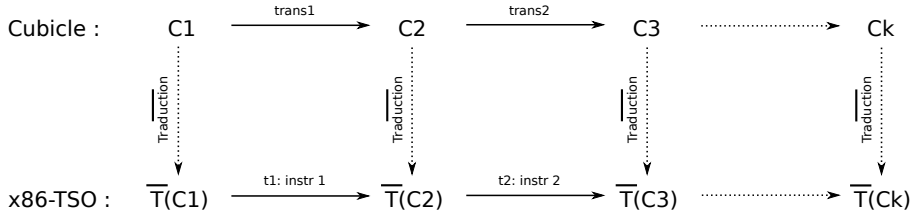
- Cas mov x, r : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q, B)$ et $Q(r) = n$, le thread t exécute l'instruction mov x, r. Par la règle WriteBuf, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q, (x, n) +\!\!+ B)], M)$. Depuis tout état Cub86-TSO $C_1 \equiv \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$, $C_1(R_t) = n$, $C_1(B_t[j_1].X) = Empty$ et $\forall j_2.\ j_2 < j_1 \implies C_1(B_t[j_2].X) = Empty$, le système prend la transition TWriteBuf, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, B_t[j_1].X \mapsto V_X, B_t[j_1].V \mapsto n] \equiv \mathcal{T}(S_2)$.

- Cas synchronisation : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q, B +\!\!+ (x, n))$, le thread t effectue une synchronisation. Par la règle WriteMem, on obtient un état $S_2 = (LS[t \mapsto (pc, zf, Q, B)], M[x \mapsto n])$. Depuis tout état Cub86-TSO $C_1 \equiv \mathcal{T}(S_1)$ tel que $C_1(B_t[j_1].X) = V_X$, $C_1(B_t[j_1].V) = n$ et $\forall j_2.\ j_1 < j_2 \implies C_1(B_t[j_2].X) = Empty$, le système prend la transition TWriteMem, et on obtient un état $C_2 = C_1[B_t[j].X \mapsto Empty, X \mapsto n] \equiv \mathcal{T}(S_2)$.

- Cas mfence : depuis l'état $S_1 = (LS, M)$ avec $LS(t) = (pc, zf, Q, B)$ et $B = \varnothing$, le thread t exécute l'instruction mfence. Par la règle Mfence, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q, B)], M)$. Depuis tout état Cub86-TSO $C_1 \equiv \mathcal{T}(S_1)$ tel que $C_1(PC_t) = L_{pc}$ et $\forall j.\ C_1(B_t[j].X) = Empty$, le système prend la transition TMfence, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}] \equiv \mathcal{T}(S_2)$.

□

Simulation arrière : Cub86-TSO vers x86-TSO



Théorème 5. La sémantique x86-TSO simule la sémantique Cub86-TSO, c'est à dire, $\forall C_1$, si $C_1 \xrightarrow{*} C_k$ alors $\forall S_1 \in \overline{\mathcal{T}}(C_1)$, $\exists S_k \in \overline{\mathcal{T}}(C_k)$ tel que $S_1 \xrightarrow{*} S_k$.

Démonstration. On procède par récurrence sur la longueur k de la dérivation $C_1 \xrightarrow{*} C_k$ et analyse par cas sur la première règle appliquée.

Hypothèse de récurrence : on suppose que pour toute dérivation de longueur $k-1$, le Théorème 5 est vrai, c'est à dire qu'on a $\forall S_2 \in \overline{\mathcal{T}}(C_2), \exists S_k \in \overline{\mathcal{T}}(C_k)$ tel que $S_2 \xrightarrow{*} S_k$.

On montre alors grâce à une analyse par cas sur les différentes règles que $\forall C_1$, si $C_1 \to C_2$, alors $\forall S_1 \in \overline{\mathcal{T}}(C_1)$, $\exists S_2 \in \overline{\mathcal{T}}(C_2)$ tel que $S_1 \to S_2$.

- Cas TReadMem : pour un thread t donné, depuis tout état Cub86-TSO C_1 tel que $C_1(PC_t) = L_{pc}$, $C_1(X) = n$ et $\forall j.\ C_1(B_t[j].X) \neq V_X$, le système prend la transition TReadMem, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, R_t \mapsto n]$. Depuis l'état x86-TSO $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q, B)$ et $M(x) = n$, le thread t exécute l'instruction $I(pc) = \mathsf{mov}\ r, x$. Comme $x \notin B$, c'est la règle ReadMem qui s'applique, et on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q[r \mapsto n], B)], M) = \overline{\mathcal{T}}(C_2)$.

- Cas TReadBuf : pour un thread t donné, depuis tout état Cub86-TSO C_1 tel que $C_1(PC_t) = L_{pc}$, $C_1(B_t[j_1].X) = V_X$, $C_1(B_t[j_1].V) = n$ et $\forall j_2.\ j_1 \leq j_2 \vee C_1(B_t[j_2].X) \neq V_X$, le système prend la transition TReadBuf, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, R_t \mapsto n]$. Depuis l'état x86-TSO $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q, B)$ et $B = B_1 +\!\!+ (x, n) +\!\!+ B_2$ tel que $x \notin B_1$, le thread t exécute l'instruction $I(pc) = \mathsf{mov}\ r, x$. Comme $x \in B$, c'est la règle ReadBuf qui s'applique, et on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q[r \mapsto n], B)], M) = \overline{\mathcal{T}}(C_2)$.

- Cas TWriteBuf : pour un thread t donné, depuis tout état Cub86-TSO C_1 tel que $C_1(PC_t) = L_{pc}$, $C_1(R_t) = n$, $C_1(B_t[j_1].X) = Empty$ et $\forall j_2.\ j_1 \leq j_2 \vee C_1(B_t[j_2].X) = Empty$, le système prend la transition TWriteBuf, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}, B_t[j_1].X \mapsto V_X, B_t[j_1].V \mapsto n]$. Depuis l'état x86-TSO $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q, B)$ et $Q(r) = n$, le thread t exécute l'instruction $I(pc) = \mathsf{mov}\ x, r$. Par la règle WriteBuf, on obtient un état $S_2 = (LS[t \mapsto (pc+1, zf, Q, B +\!\!+ (x, n))], M) = \overline{\mathcal{T}}(C_2)$.

- Cas TWriteMem : pour un thread t donné, depuis tout état Cub86-TSO C_1 tel que $C_1(B_t[j_1].X) = V_X$, $C_1(B_t[j_1].V) = n$ et $\forall j_2.\ j_2 \leq j_1 \vee C_1(B_t[j_2].X) = Empty$, le système prend la transition TWriteMem, et on obtient un état $C_2 = C_1[X \mapsto n, B_t[j_1].X = Empty]$. Depuis l'état x86-TSO $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q, B +\!\!+ (x, n))$, le thread t effectue une synchronisation. Par la règle WriteMem, on obtient un état $S_2 = (LS[t \mapsto (pc, zf, Q, B)], M[x \mapsto n]) = \overline{\mathcal{T}}(C_2)$.

– Cas TMFENCE : pour un thread t donné, depuis tout état Cub86-TSO C_1 tel que $C_1(PC_t) = L_{pc}$ et $\forall j.\ C_1(B_t[j].X) = Empty$, le système prend la transition TMFENCE, et on obtient un état $C_2 = C_1[PC_t \mapsto L_{pc+1}]$. Depuis l'état x86-TSO $S_1 = (LS, M) = \overline{\mathcal{T}}(C_1)$ avec $LS(t) = (pc, zf, Q, B)$ et $B = \varnothing$, le thread t exécute l'instruction $I(pc) = \mathsf{mfence}$. Par la règle MFENCE, on obtient un état $S_2 = (LS[t \mapsto (pc + 1, zf, Q, B)], M) = \overline{\mathcal{T}}(C_2)$.

□

Chapitre 5

Résultats

Nous avons utilisé notre outil pour vérifier divers programmes types rencontrés dans la littérature relative aux modèles mémoire faibles (le lecteur intéressé peut se référer à [5] et [16] pour plus d'informations sur ces programmes). Ces programmes minimalistes ont été conçus spécifiquement pour exhiber des différence de comportement bien précises sur des modèles mémoire faibles spécifiques. Nous l'avons bien entendu également utilisé pour vérifer l'algorithme de Peterson.

Le tableau suivant montre les résultats obtenus en vérifiant ces programmes avec Cubicle (avec l'option -brab 2), aussi bien sous le modèle SC que sous le modèle TSO. La plateforme de test est équipée d'un processeur Intel Xeon W5580 8-cœurs @ 3.20 GHz et dispose de 24 Go de RAM. Dans chaque cas, on donne le résultat trouvé (safe ou unsafe), le nombre de noeuds explorés, le nombre d'invariants découverts et le temps d'exploration. Le symbole ☠ dans la colonne des invariants indique que la propriété de sûreté n'est pas vérifiée.

	SC				TSO			
	Rés.	Noeuds	Inv.	Temps	Rés.	Noeuds	Inv.	Temps
`sb.asm`	safe	38	3	0,24s	unsafe	89	☠	0,51s
`sb_fixed.asm`	safe	52	3	0,35s	safe	241	15	1,95s
`rwc.asm`	safe	46	4	0,23s	unsafe	151	☠	0,92s
`rwc_fixed.asm`	safe	56	4	0,30s	safe	244	21	1,92s
`wrc.asm`	safe	76	10	0,40s	safe	191	29	1,88s
`iriw.asm`	safe	97	11	0,40s	safe	353	26	3,40s
`mp3.asm`	safe	90	7	0,42s	safe	347	22	3,11s
`peterson.asm`	safe	192	20	5,62s	unsafe	660	☠	5,40s
`peterson_fixed.asm`	safe	257	20	7,88s	safe	5873	1	8m38

On remarque que les programmes SB, RWC et Peterson, qui étaient corrects en SC, deviennent incorrects sous TSO. Grâce à l'ajout d'instructions mfence judicieusement placées, ces programmes ont pu être "réparés", de façon à redevenir corrects sous TSO.

Tous ces programmes sont aisément vérifiés sous le modèle SC. Sous le modèle TSO, on observe que le nombre de noeuds explorés par Cubicle est beaucoup plus important, et de ce fait le temps d'exploration est plus long. On observe également que plus d'*invariants* sont trouvés : en effet, le mécanisme de recherche d'*invariants* de Cubicle a pu découvrir des *invariants* sur le contenu des *tampons*, c'est d'ailleurs ce qui lui permet d'aboutir à un résultat. Le cas de l'algorithme de Peterson corrigé est le plus intéressant. En effet, l'espace d'exploration de ce programme est considérable sous le modèle TSO, d'où un temps d'exploration nettement plus long. Sans les tampons à trous et diverses optimisations réalisées dans la modélisation, on ne serait pas parvenus à ce résultat. On voit donc qu'on s'approche des limites de cette méthode, et que le passage à l'échelle, avec des programmes plus complexes, semble difficile. Malgré cela, on note qu'il s'agit de la première preuve automatique de l'algorithme de Peterson sur le modèle TSO avec une taille de tampon arbitraire (là où d'autres outils travaillent avec des tampons de taille fixe).

Chapitre 6

Conclusion et perspectives

A l'issue de ce stage, nous disposons d'un outil capable de traduire un certain nombre de programmes assembleur concurrents sur le modèle TSO vers le langage d'entrée de Cubicle. C'est à notre connaissance le seul outil permettant la vérification de programmes assembleur sur le modèle TSO qui sache faire abstraction de la taille des tampons. De plus, nous avons prouvé l'équivalence entre les sémantique des systèmes de transition générés et les programmes assembleur x86 originaux. Cette preuve est essentielle pour garantir que les résultats trouvés par Cubicle puissent être transposés dans la sémantique du langage d'assemblage. Nous avons pu de cette façon vérifier divers programme types de la littérature du domaine, en particulier l'algorithme de Peterson pour des tampons de taille arbitraire, ce qui montre l'intérêt de cette méthode.

Toutefois, il nous semble que l'on commence à atteindre les limites de cette méthode. Pour envisager le passage à l'échelle, deux approches sont possibles. La première serait de travailler sur les invariants spécifiques aux tampons TSO, pour espérer couper au maximum dans l'espace d'exploration. Nous avons tenté cette approche, en récupérant les invariants générés par Cubicle lors de l'exploration SC et en tentant de les injecter dans TSO, toutefois, cela ne s'est pas révélé efficace. La seconde approche serait de combiner le model checking avec des techniques de test ; approche qui sera explorée en thèse. Nous espérons de cette façon pouvoir envisager la vérification de programmes sur des modèles mémoire plus faibles que x86-TSO, notamment le modèle mémoire C11/C++11[3]. En effet, ce modèle mémoire a été conçu pour prendre en compte le plus grand éventail possible de modèles mémoire, de façon à anticiper les éventuelles évolutions des architectures futures. De ce fait, c'est un modèle mémoire particulièrement relâché, et la vérification de programmes sous ce modèle constitue un véritable défi.

Remerciements

Je tiens tout d'abord à remercier mon directeur de stage Mr Sylvain Conchon, pour son encadrement, ses nombreux conseils et encouragements, et pour le temps qu'il m'a consacré.

Je remercie également Mme Fatiha Zaidi et Mr Alain Mebsout, qui ont contribué à la réussite de ce stage par leurs conseils et leur participation aux réunions de travail.

Je tiens enfin à remercier l'ensemble des membres de l'équipe VALS pour leur accueil et l'ambiance de te travail agréable qu'ils ont su créer.

Bibliographie

[1] The cubicle model checker. http://cubicle.lri.fr.

[2] Relaxed-memory concurrency. http://www.cl.cam.ac.uk/~pes20/weakmemory/.

[3] C11 standard, final committee draft (n1570). http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf, April 2011.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under tso. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 530–536, Berlin, Heidelberg, 2013. Springer-Verlag.

[5] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models : A tutorial. *Computer*, 29(12) :66–76, December 1996.

[6] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence : A static analysis approach to automatic fence insertion. In *CAV*, pages 507–523, 2014. arXiv version available.

[7] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 512–532, Berlin, Heidelberg, 2013. Springer-Verlag.

[8] Jade Alglave and Luc Maranget. Stability in weak memory models. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 50–66, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 7–18, New York, NY, USA, 2010. ACM.

[10] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence : Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 12–21, New York, NY, USA, 2007. ACM.

[11] Sylvain Conchon, Alain Mebsout, and Fatiha Zaidi. Vérification de systèmes paramétrés avec Cubicle. In Damien Pous and Christine Tasson, editors, *JFLA - Journées francophones des langages applicatifs - 2013*, Aussois, France, February 2013. Damien Pous and Christine Tasson.

[12] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards SMT model checking of array-based systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin Heidelberg, 2008.

[13] Thuan Quang Huynh and Abhik Roychoudhury. A memory model sensitive checker for c#. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 476–491, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9) :690–691, September 1979.

[15] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 429–440, New York, NY, USA, 2012. ACM.

[16] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model : X86-tso. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso : A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7) :89–97, July 2010.

Annexe

Annexe 1 : Syntaxe du sous-ensemble du langage d'assemblage x86

⟨*digit*⟩ ::= `0-9`
⟨*alpha*⟩ ::= `a-z A-Z`
⟨*ident*⟩ ::= (⟨*alpha*⟩ | `_`) (⟨*alpha*⟩ | `_` | ⟨*digit*⟩)⋆
⟨*sident*⟩ ::= `.` (⟨*alpha*⟩ | `_` | ⟨*digit*⟩)⋆
⟨*integer*⟩ ::= ⟨*digit*⟩+

⟨*program*⟩ ::= ⟨*section*⟩+
⟨*section*⟩ ::= `section` ⟨*sident*⟩ `NL` ⟨*line*⟩⋆
⟨*line*⟩ ::= ⟨*label*⟩⋆ (⟨*dinstr*⟩ | ⟨*cinstr*⟩) `NL`
⟨*label*⟩ ::= ⟨*ident*⟩ `:` ?
⟨*dinstr*⟩ ::= (`db` | `dd`) ⟨*values*⟩
⟨*values*⟩ ::= ⟨*value*⟩ | ⟨*value*⟩ `,` ⟨*values*⟩+
⟨*value*⟩ ::= ⟨*integer*⟩ | ⟨*string*⟩
⟨*cinstr*⟩ ::= `inc` ⟨*oprm*⟩
| `dec` ⟨*oprm*⟩
| `not` ⟨*oprm*⟩
| `add` ⟨*oprm*⟩ `,` ⟨*oprmi*⟩
| `sub` ⟨*oprm*⟩ `,` ⟨*oprmi*⟩
| `xchg` ⟨*oprm*⟩ `,` ⟨*oprm*⟩
| `xadd` ⟨*oprm*⟩ `,` ⟨*opr*⟩
| `cmp` ⟨*oprm*⟩ `,` ⟨*oprmi*⟩
| `mov` ⟨*oprm*⟩ `,` ⟨*oprmi*⟩
| `jmp` ⟨*ident*⟩
| `jcc` ⟨*ident*⟩
| `nop`
| `mfence`
| `lock` ⟨*cinstr*⟩
| `call` ⟨*ident*⟩
⟨*opr*⟩ ::= ⟨*reg*⟩
⟨*oprm*⟩ ::= ⟨*reg*⟩ | ⟨*mem*⟩
⟨*oprmi*⟩ ::= ⟨*reg*⟩ | ⟨*mem*⟩ | ⟨*imm*⟩
⟨*reg*⟩ ::= `eax` | `ebx` | `ecx` | `edx` | `esi` | `edi`
⟨*mem*⟩ ::= `[` ⟨*ident*⟩ `]`
| `[gs:` ⟨*ident*⟩ `- tdata]`
| `[gs:` ⟨*ident*⟩ `- tbss]`
⟨*imm*⟩ ::= ⟨*integer*⟩ | ⟨*ident*⟩

Remarque :
– Les instructions peuvent être utilisées avec au plus une opérande mémoire

Annexe 2 : code Cub86-SC généré pour l'algorithme de Peterson

```
type loc = IDLE | L_wait_1_2 | L_wait_1_1 | L__start | L_sc_2_end_1 | L_wait_2 |
L_thread_2 | L_wait_2_2 | L_wait_1 | L_sc_2 | L_thread_2_1 | L_wait_2_3 |
L__start_4 | L_thread_1 | L_sc_1 | L_sc_1_end_1 | L_wait_1_3 | L_thread_1_1 |
L__start_2 | L__start_3 | L_sc_2_end | L_sc_1_end | L__start_1 | L_wait_2_1 |
END

var GLOB_turn : int
var GLOB_want1 : int
var GLOB_want2 : int
var PC_0x1 : loc
var RES_0x1 : int
var TMP_0x1 : int
var ARG_0x1 : int
var PC_1x1 : loc
var RES_1x1 : int
var TMP_1x1 : int
var ARG_1x1 : int
var PC_2x1 : loc
var RES_2x1 : int
var TMP_2x1 : int
var ARG_2x1 : int

init (p) { GLOB_turn = 0 && GLOB_want1 = 0 && GLOB_want2 = 0 && PC_0x1 = IDLE &&
           PC_1x1 = IDLE && PC_2x1 = IDLE }

unsafe (p) { PC_1x1 = L_sc_1 && PC_2x1 = L_sc_2 }

transition t0x1_IDLE_L__start ()
requires { PC_0x1 = IDLE }
{ PC_0x1 := L__start }

transition t0x1_L__start_L__start_1_pthread_create ()
requires { PC_0x1 = L__start }
{ PC_0x1 := L__start_1; PC_1x1 := L_thread_1 }

transition t0x1_L__start_1_L__start_2_pthread_create ()
requires { PC_0x1 = L__start_1 }
{ PC_0x1 := L__start_2; PC_2x1 := L_thread_2 }

transition t0x1_L__start_2_L__start_3_pthread_join ()
requires { PC_0x1 = L__start_2 && PC_1x1 = END }
{ PC_0x1 := L__start_3 }

transition t0x1_L__start_3_L__start_4_pthread_join ()
requires { PC_0x1 = L__start_3 && PC_2x1 = END }
{ PC_0x1 := L__start_4 }

transition t0x1_L__start_4_END ()
```

```
requires { PC_0x1 = L__start_4 }
{ PC_0x1 := END }

transition t1x1_L_thread_1_L_thread_1_1_mov ()
requires { PC_1x1 = L_thread_1 }
{ PC_1x1 := L_thread_1_1; GLOB_want1 := 1 }

transition t1x1_L_thread_1_1_L_wait_1_mov ()
requires { PC_1x1 = L_thread_1_1 }
{ PC_1x1 := L_wait_1; GLOB_turn := 1 }

transition t1x1_L_wait_1_L_wait_1_1_cmp ()
requires { PC_1x1 = L_wait_1 }
{ PC_1x1 := L_wait_1_1; RES_1x1 := GLOB_want2 - 1 }

transition t1x1_L_wait_1_1_L_sc_1_jmp_true ()
requires { PC_1x1 = L_wait_1_1 && RES_1x1 <> 0 }
{ PC_1x1 := L_sc_1 }

transition t1x1_L_wait_1_1_L_wait_1_2_jmp_false ()
requires { PC_1x1 = L_wait_1_1 && RES_1x1 = 0 }
{ PC_1x1 := L_wait_1_2 }

transition t1x1_L_wait_1_2_L_wait_1_3_cmp ()
requires { PC_1x1 = L_wait_1_2 }
{ PC_1x1 := L_wait_1_3; RES_1x1 := GLOB_turn - 1 }

transition t1x1_L_wait_1_3_L_wait_1_jmp_true ()
requires { PC_1x1 = L_wait_1_3 && RES_1x1 = 0 }
{ PC_1x1 := L_wait_1 }

transition t1x1_L_wait_1_3_L_sc_1_jmp_false ()
requires { PC_1x1 = L_wait_1_3 && RES_1x1 <> 0 }
{ PC_1x1 := L_sc_1 }

transition t1x1_L_sc_1_end_L_sc_1_end_1_mov ()
requires { PC_1x1 = L_sc_1_end }
{ PC_1x1 := L_sc_1_end_1; GLOB_want1 := 0 }

transition t1x1_L_sc_1_end_1_END ()
requires { PC_1x1 = L_sc_1_end_1 }
{ PC_1x1 := END }

transition t2x1_L_thread_2_L_thread_2_1_mov ()
requires { PC_2x1 = L_thread_2 }
{ PC_2x1 := L_thread_2_1; GLOB_want2 := 1 }

transition t2x1_L_thread_2_1_L_wait_2_mov ()
requires { PC_2x1 = L_thread_2_1 }
```

```
{ PC_2x1 := L_wait_2; GLOB_turn := 0 }

transition t2x1_L_wait_2_L_wait_2_1_cmp ()
requires { PC_2x1 = L_wait_2 }
{ PC_2x1 := L_wait_2_1; RES_2x1 := GLOB_want1 - 1 }

transition t2x1_L_wait_2_1_L_sc_2_jmp_true ()
requires { PC_2x1 = L_wait_2_1 && RES_2x1 <> 0 }
{ PC_2x1 := L_sc_2 }

transition t2x1_L_wait_2_1_L_wait_2_2_jmp_false ()
requires { PC_2x1 = L_wait_2_1 && RES_2x1 = 0 }
{ PC_2x1 := L_wait_2_2 }

transition t2x1_L_wait_2_2_L_wait_2_3_cmp ()
requires { PC_2x1 = L_wait_2_2 }
{ PC_2x1 := L_wait_2_3; RES_2x1 := GLOB_turn - 0 }

transition t2x1_L_wait_2_3_L_wait_2_jmp_true ()
requires { PC_2x1 = L_wait_2_3 && RES_2x1 = 0 }
{ PC_2x1 := L_wait_2 }

transition t2x1_L_wait_2_3_L_sc_2_jmp_false ()
requires { PC_2x1 = L_wait_2_3 && RES_2x1 <> 0 }
{ PC_2x1 := L_sc_2 }

transition t2x1_L_sc_2_end_L_sc_2_end_1_mov ()
requires { PC_2x1 = L_sc_2_end }
{ PC_2x1 := L_sc_2_end_1; GLOB_want2 := 0 }

transition t2x1_L_sc_2_end_1_END ()
requires { PC_2x1 = L_sc_2_end_1 }
{ PC_2x1 := END }
```

Annexe 3 : code Cub86-TSO généré pour l'algorithme de Peterson

```
type loc = IDLE | L_wait_1_2 | L_wait_1_1 | L__start | L_sc_2_end_1 | L_wait_2 |
L_thread_2 | L_wait_2_2 | L_wait_1 | L_sc_2 | L_thread_2_1 | L_wait_2_3 |
L__start_4 | L_thread_1 | L_sc_1 | L_sc_1_end_1 | L_wait_1_3 | L_thread_1_1 |
L_wait_2b | L__start_2 | L__start_3 | L_sc_2_end | L_sc_1_end | L__start_1 |
L_wait_1b | L_wait_1_2b | L_wait_2_1 | L_wait_2_2b | END

type tvar = None | Vturn | Vwant1 | Vwant2

var GLOB_turn : int
var GLOB_want1 : int
var GLOB_want2 : int
var PC_0x1 : loc
var RES_0x1 : int
var TMP_0x1 : int
var ARG_0x1 : int
var PC_1x1 : loc
var RES_1x1 : int
var TMP_1x1 : int
var ARG_1x1 : int
var PC_2x1 : loc
var RES_2x1 : int
var TMP_2x1 : int
var ARG_2x1 : int
var LOCK : bool
array BVAR_0x1[proc] : tvar
array BVAL_0x1[proc] : int
var BLEN_0x1 : int
array BVAR_1x1[proc] : tvar
array BVAL_1x1[proc] : int
var BLEN_1x1 : int
array BVAR_2x1[proc] : tvar
array BVAL_2x1[proc] : int
var BLEN_2x1 : int

init (p b) { GLOB_turn = 0 && GLOB_want1 = 0 && GLOB_want2 = 0 && PC_0x1 = IDLE &&
 PC_1x1 = IDLE && PC_2x1 = IDLE && LOCK = False && BVAR_0x1[b] = None &&
 BLEN_0x1 = 0 && BVAR_1x1[b] = None && BLEN_1x1 = 0 && BVAR_2x1[b] = None &&
 BLEN_2x1 = 0 }

unsafe (p) { PC_1x1 = L_sc_1 && PC_2x1 = L_sc_2 }

transition t0x1_IDLE_L__start ()
requires { PC_0x1 = IDLE }
{ PC_0x1 := L__start }

transition t0x1_L__start_L__start_1_pthread_create ()
requires { PC_0x1 = L__start }
{ PC_0x1 := L__start_1; PC_1x1 := L_thread_1 }
```

```
transition t0x1_L__start_1_L__start_2_pthread_create ()
requires { PC_0x1 = L__start_1 }
{ PC_0x1 := L__start_2; PC_2x1 := L_thread_2 }

transition t0x1_L__start_2_L__start_3_pthread_join ()
requires { PC_0x1 = L__start_2 && PC_1x1 = END }
{ PC_0x1 := L__start_3 }

transition t0x1_L__start_3_L__start_4_pthread_join ()
requires { PC_0x1 = L__start_3 && PC_2x1 = END }
{ PC_0x1 := L__start_4 }

transition t0x1_L__start_4_END ()
requires { PC_0x1 = L__start_4 }
{ PC_0x1 := END }

transition t1x1_L_thread_1_L_thread_1_1_mov (bd)
requires { PC_1x1 = L_thread_1 && BVAR_1x1[bd] = None &&
           forall_other bx. (bx < bd || BVAR_1x1[bx] = None) }
{ PC_1x1 := L_thread_1_1; BVAL_1x1[bd] := 1; BVAR_1x1[bd] := Vwant1;
  BLEN_1x1 := BLEN_1x1 + 1 }

transition t1x1_L_thread_1_1_L_wait_1_mov (bd)
requires { PC_1x1 = L_thread_1_1 && BVAR_1x1[bd] = None &&
           forall_other bx. (bx < bd || BVAR_1x1[bx] = None) }
{ PC_1x1 := L_wait_1; BVAL_1x1[bd] := 1; BVAR_1x1[bd] := Vturn;
  BLEN_1x1 := BLEN_1x1 + 1 }

transition t1x1_L_wait_1_L_wait_1b_pre_early (bs)
requires { PC_1x1 = L_wait_1 && LOCK = False && BVAR_1x1[bs] = Vwant2 &&
           forall_other bx. (bx < bs || BVAR_1x1[bx] <> Vwant2) }
{ PC_1x1 := L_wait_1b; TMP_1x1 := BVAL_1x1[bs] }

transition t1x1_L_wait_1_L_wait_1b_pre_direct (bs)
requires { PC_1x1 = L_wait_1 && LOCK = False &&
           forall_other bx. (BVAR_1x1[bx] <> Vwant2) }
{ PC_1x1 := L_wait_1b; TMP_1x1 := GLOB_want2 }

transition t1x1_L_wait_1b_L_wait_1_1_cmp ()
requires { PC_1x1 = L_wait_1b }
{ PC_1x1 := L_wait_1_1; RES_1x1 := TMP_1x1 - 1 }

transition t1x1_L_wait_1_1_L_sc_1_jmp_true ()
requires { PC_1x1 = L_wait_1_1 && RES_1x1 <> 0 }
{ PC_1x1 := L_sc_1 }

transition t1x1_L_wait_1_1_L_wait_1_2_jmp_false ()
requires { PC_1x1 = L_wait_1_1 && RES_1x1 = 0 }
```

```
{ PC_1x1 := L_wait_1_2 }

transition t1x1_L_wait_1_2_L_wait_1_2b_pre_early (bs)
requires { PC_1x1 = L_wait_1_2 && LOCK = False && BVAR_1x1[bs] = Vturn &&
           forall_other bx. (bx < bs || BVAR_1x1[bx] <> Vturn) }
{ PC_1x1 := L_wait_1_2b; TMP_1x1 := BVAL_1x1[bs] }

transition t1x1_L_wait_1_2_L_wait_1_2b_pre_direct (bs)
requires { PC_1x1 = L_wait_1_2 && LOCK = False &&
           forall_other bx. (BVAR_1x1[bx] <> Vturn) }
{ PC_1x1 := L_wait_1_2b; TMP_1x1 := GLOB_turn }

transition t1x1_L_wait_1_2b_L_wait_1_3_cmp ()
requires { PC_1x1 = L_wait_1_2b }
{ PC_1x1 := L_wait_1_3; RES_1x1 := TMP_1x1 - 1 }

transition t1x1_L_wait_1_3_L_wait_1_jmp_true ()
requires { PC_1x1 = L_wait_1_3 && RES_1x1 = 0 }
{ PC_1x1 := L_wait_1 }

transition t1x1_L_wait_1_3_L_sc_1_jmp_false ()
requires { PC_1x1 = L_wait_1_3 && RES_1x1 <> 0 }
{ PC_1x1 := L_sc_1 }

transition t1x1_L_sc_1_end_L_sc_1_end_1_mov (bd)
requires { PC_1x1 = L_sc_1_end && BVAR_1x1[bd] = None &&
           forall_other bx. (bx < bd || BVAR_1x1[bx] = None) }
{ PC_1x1 := L_sc_1_end_1; BVAL_1x1[bd] := 0; BVAR_1x1[bd] := Vwant1;
  BLEN_1x1 := BLEN_1x1 + 1 }

transition t1x1_L_sc_1_end_1_END ()
requires { PC_1x1 = L_sc_1_end_1 }
{ PC_1x1 := END }

transition t2x1_L_thread_2_L_thread_2_1_mov (bd)
requires { PC_2x1 = L_thread_2 && BVAR_2x1[bd] = None &&
           forall_other bx. (bx < bd || BVAR_2x1[bx] = None) }
{ PC_2x1 := L_thread_2_1; BVAL_2x1[bd] := 1; BVAR_2x1[bd] := Vwant2;
  BLEN_2x1 := BLEN_2x1 + 1 }

transition t2x1_L_thread_2_1_L_wait_2_mov (bd)
requires { PC_2x1 = L_thread_2_1 && BVAR_2x1[bd] = None &&
           forall_other bx. (bx < bd || BVAR_2x1[bx] = None) }
{ PC_2x1 := L_wait_2; BVAL_2x1[bd] := 0; BVAR_2x1[bd] := Vturn;
  BLEN_2x1 := BLEN_2x1 + 1 }

transition t2x1_L_wait_2_L_wait_2b_pre_early (bs)
requires { PC_2x1 = L_wait_2 && LOCK = False && BVAR_2x1[bs] = Vwant1 &&
           forall_other bx. (bx < bs || BVAR_2x1[bx] <> Vwant1) }
```

```
{ PC_2x1 := L_wait_2b; TMP_2x1 := BVAL_2x1[bs] }

transition t2x1_L_wait_2_L_wait_2b_pre_direct (bs)
requires { PC_2x1 = L_wait_2 && LOCK = False &&
           forall_other bx. (BVAR_2x1[bx] <> Vwant1) }
{ PC_2x1 := L_wait_2b; TMP_2x1 := GLOB_want1 }

transition t2x1_L_wait_2b_L_wait_2_1_cmp ()
requires { PC_2x1 = L_wait_2b }
{ PC_2x1 := L_wait_2_1; RES_2x1 := TMP_2x1 - 1 }

transition t2x1_L_wait_2_1_L_sc_2_jmp_true ()
requires { PC_2x1 = L_wait_2_1 && RES_2x1 <> 0 }
{ PC_2x1 := L_sc_2 }

transition t2x1_L_wait_2_1_L_wait_2_2_jmp_false ()
requires { PC_2x1 = L_wait_2_1 && RES_2x1 = 0 }
{ PC_2x1 := L_wait_2_2 }

transition t2x1_L_wait_2_2_L_wait_2_2b_pre_early (bs)
requires { PC_2x1 = L_wait_2_2 && LOCK = False && BVAR_2x1[bs] = Vturn &&
           forall_other bx. (bx < bs || BVAR_2x1[bx] <> Vturn) }
{ PC_2x1 := L_wait_2_2b; TMP_2x1 := BVAL_2x1[bs] }

transition t2x1_L_wait_2_2_L_wait_2_2b_pre_direct (bs)
requires { PC_2x1 = L_wait_2_2 && LOCK = False &&
           forall_other bx. (BVAR_2x1[bx] <> Vturn) }
{ PC_2x1 := L_wait_2_2b; TMP_2x1 := GLOB_turn }

transition t2x1_L_wait_2_2b_L_wait_2_3_cmp ()
requires { PC_2x1 = L_wait_2_2b }
{ PC_2x1 := L_wait_2_3; RES_2x1 := TMP_2x1 - 0 }

transition t2x1_L_wait_2_3_L_wait_2_jmp_true ()
requires { PC_2x1 = L_wait_2_3 && RES_2x1 = 0 }
{ PC_2x1 := L_wait_2 }

transition t2x1_L_wait_2_3_L_sc_2_jmp_false ()
requires { PC_2x1 = L_wait_2_3 && RES_2x1 <> 0 }
{ PC_2x1 := L_sc_2 }

transition t2x1_L_sc_2_end_L_sc_2_end_1_mov (bd)
requires { PC_2x1 = L_sc_2_end && BVAR_2x1[bd] = None &&
           forall_other bx. (bx < bd || BVAR_2x1[bx] = None) }
{ PC_2x1 := L_sc_2_end_1; BVAL_2x1[bd] := 0; BVAR_2x1[bd] := Vwant2;
  BLEN_2x1 := BLEN_2x1 + 1 }

transition t2x1_L_sc_2_end_1_END ()
requires { PC_2x1 = L_sc_2_end_1 }
```

```
{ PC_2x1 := END }

transition mt2x1_store_want2 (b)
requires { LOCK = False && BVAR_2x1[b] = Vwant2 &&
           forall_other bx. (b <= bx || BVAR_2x1[bx] = None) }
{ BVAR_2x1[b] := None; GLOB_want2 := BVAL_2x1[b]; BLEN_2x1 := BLEN_2x1 - 1 }

transition mt1x1_store_want1 (b)
requires { LOCK = False && BVAR_1x1[b] = Vwant1 &&
           forall_other bx. (b <= bx || BVAR_1x1[bx] = None) }
{ BVAR_1x1[b] := None; GLOB_want1 := BVAL_1x1[b]; BLEN_1x1 := BLEN_1x1 - 1 }

transition mt1x1_store_turn (b)
requires { LOCK = False && BVAR_1x1[b] = Vturn &&
           forall_other bx. (b <= bx || BVAR_1x1[bx] = None) }
{ BVAR_1x1[b] := None; GLOB_turn := BVAL_1x1[b]; BLEN_1x1 := BLEN_1x1 - 1 }

transition mt2x1_store_turn (b)
requires { LOCK = False && BVAR_2x1[b] = Vturn &&
           forall_other bx. (b <= bx || BVAR_2x1[bx] = None) }
{ BVAR_2x1[b] := None; GLOB_turn := BVAL_2x1[b]; BLEN_2x1 := BLEN_2x1 - 1 }
```