Analyse formelle du protocole ACE : cohérence de caches des systèmes sur puce

Abderahman Kriouile
STMicroelectronics/Inria/LIG
Email: abderahman.kriouile@st.com

Wendelin Serwe
Inria/LIG
Email: wendelin.serwe@inria.fr

***Résumé*—Les architectures des systèmes sur puce (*System-on-Chip*, SoC) d'aujourd'hui intègrent de nombreux composants différents tels que les processeurs, les accélérateurs, les mémoires et les blocs d'entrée/sortie, certains pouvant contenir des caches. Etant donné que l'effort de validation basée sur la simulation, actuellement utilisée dans l'industrie, croît de façon exponentielle avec la complexité des SoCs, nous nous intéressons à des techniques de vérification formelle. Nous utilisons la boîte à outils CADP pour développer et valider un modèle formel d'un SoC générique conforme à la spécification ACE, récemment proposée par ARM dans le but de mettre en œuvre la cohérence de cache au niveau système.**

I. INTRODUCTION

L'intégration de plus en plus de fonctionnalités dans les Smartphones et les décodeurs de télévision acroît la complexité du logiciel embarqué et de l'architecture matérielle. Cette dernière est en général un système sur puce (*System-on-Chip*, SoC) ou réseau sur puce complexe, comportant un nombre important de composants hétérogènes. Un SoC typique comprend aujourd'hui, en plus des processeurs et de la mémoire, des accélérateurs matériels dédiés et des blocs d'entrée/sortie. L'intégration des caches dans certains de ces composants (les processeurs et les accélérateurs matériels) peut améliorer la performance et réduire la consommation d'énergie.

Initialement, la cohérence de caches était gérée au niveau logiciel, afin de profiter de la flexibilité des solutions logicielles. Toutefois, en raison de la complexité croissante des logiciels, une nouvelle tendance [16] consiste à introduire un support matériel pour la cohérence de caches afin d'alléger la charge des processeurs. Dans ce cadre, ARM a proposé ACE [1], qui est devenu un standard industriel pour la cohérence de caches au niveau système dans les SoC hétérogènes. De sa part, STMicroelectronics est en train d'intégrer la cohérence au niveau système dans ses prochains SoCs (en se basant sur ACE).

Compte tenu de la complexité et la difficulté de la validation des protocoles de cohérence de caches, assurer la cohérence au niveau système s'avère l'un des défis majeurs auxquels sont confrontés les architectes des SoCs d'aujourd'hui. Les techniques de validation actuellement utilisées dans le flot de conception industrielle sont basées sur la simulation. Néanmoins, leur croissant exponentiellement avec la complexité des architectures matérielles, nous étudions l'application des techniques de vérification formelles. Ainsi, l'effort humain de modélisation croit linéairement avec la complexité des architectures (chaque composant est modelisé par un processus), et la complexité exponentielle est prise en charge par les outils automatiques de génération et de vérification. En particulier, nous utilisons la boîte à outils CADP [8] et son langage de modélisation LNT [3] pour la modélisation et l'analyse de la cohérence de caches au niveau système dans un SoC hétérogène.

Nous avons commencé par développer un modèle LNT générique d'un SoC, comportant un composant d'interconnexion (*interconnect*) supportant la cohérence de cache conforme au protocole ACE et des composants maîtres et esclaves abstraits (comme les processeurs et la mémoire partagée). Le modèle est paramétrique et peut être instancié avec des configurations différentes (nombre de maîtres, nombre de lignes mémoires, nombre de lignes de cache) et des différents sous-ensembles de transactions ACE. Nous avons utilisé un style orienté contraintes [2] pour modéliser les exigences globales de la spécification ACE, qui doivent être garantis par toute mise en œuvre du protocole ACE. Le modèle LNT a permis aux architectes de STMicroelectronics de simuler de manière interactive un SoC cohérent au niveau système. Nous avons également exprimé plusieurs propriétés de correction dans le langage MCL [11] et nous les avons vérifiées sur le modèle LNT. A partir des contre-exemples générés, nous avons extrait des scénarios intéressants à tester sur toute mise en œuvre d'un interconnect basé sur ACE.

La suite de cet article est organisée comme suit : La section II présente la spécification de ACE. La section III décrit la modélisation LNT d'un SoC conforme à ACE. La section IV traite la validation des propriétés de correction. La section V explore les travaux liés à notre travail. La section VI donne des conclusions et les orientations des travaux à venir.

II. ACE : COHÉRENCE DE CACHE AU NIVEAU SYSTÈME

Ces composants hétérogènes d'un SoC (tels que les processeurs génériques ou spécialisés, les mémoires, les interconnects, les IPs dédiés ou les composants d'entrée/so rtie) accèdent habituellement à une *mémoire partagée*, composée de plusieurs *lignes mémoires*. Pour améliorer les performances d'accès aux données, certains composants peuvent utiliser un *cache*, contenant des copies locales des lignes mémoires. Un SoC est dit *cohérent* si les opérations d'écriture sur la même ligne mémoire par deux composants sont observables

dans le même ordre par tous les composants du système. On peut distinguer les lignes mémoire *partageables* et *non partageables*.

Les composants d'un SoC peuvent être regroupés en composants *maîtres* (comme les processeurs) et composants *esclaves* (comme les mémoires). Les composants communiquent par l'intermédiaire d'un composant d'interconnexion, appelé *interconnect*. Dans le cas d'un système cohérent, l'interconnect est également appelé CCI. Chaque composant communique avec l'interconnect à travers un port de communication. Les opérations effectuées sur les ports sont appelées *transactions*.

A. Le protocole ACE

La spécification ACE, définie par ARM, est conçue pour maintenir la cohérence lors du partage de données entre les caches d'un SoC, pour permettre l'interaction entre des composants hétérogènes, et pour assurer la réutilisation maximale des données en cache. La spécification ACE définit le protocole d'interface matérielle (entre les composants et l'interconnect), le comportement attendu des composants, et les responsabilités de l'interconnect.

B. Les états ACE

Le protocole ACE distingue cinq états d'une ligne de cache.

- ACE_I : la ligne de cache n'est pas valide.
- ACE_UD : la donnée de la ligne de cache est unique et le maître est responsable de son écriture à la mémoire.
- ACE_UC : la donnée de la ligne de cache est unique et le maître n'est pas responsable de son écriture à la mémoire.
- ACE_SD : la donnée de la ligne de cache peut être partagée et le maître est responsable de son écriture à la mémoire.
- ACE_SC : la donnée de la ligne de cache peut être partagée et le maître n'est pas responsable de son écriture à la mémoire.

C. Les ports et canaux ACE

La spécification ACE distingue trois types de ports : un port ACE utilisé pour des composants ayant un cache, un port ACE-Lite utilisé pour les composants sans cache et un port AXI utilisé pour les composants n'utilisant pas la cohérence.

Chaque port est constitué de plusieurs canaux. Le protocole ACE prévoit trois types de canaux: *canaux de lecture*, *canaux d'écriture* et *canaux de fouine* (*snoop*). Les canaux de lecture (AR, R) et d'écriture (AW, W, B) sont utilisés pour lire et écrire des données. Les canaux de fouine (AC, CR, CD) sont utilisés pour les demandes de fouine émises par l'interconnect à des maîtres avec un cache.

D. Les transactions ACE

Le protocole ACE définit plusieurs types de transactions. Les *transactions cohérentes* sont utilisées pour accéder à des lignes mémoires partageables, qui pourraient être présentes dans les caches d'autres composants. Les *transactions sans fouine* sont utilisées pour accéder à des lignes de mémoire non partageables, qui ne doivent pas être présentes dans les caches d'autres composants. Les *transactions de mise à jour de la mémoire* sont utilisées pour mettre à jour la mémoire partagée. Les *transactions de maintenance de caches* sont utilisées par les maîtres afin d'accéder et modifier les caches d'autres maîtres. Les *transactions de fouine* sont initiées par l'interconnect à l'issue de transactions cohérentes ou de transactions de maintenance de caches. Les *transactions ACE-Lite* sont un sous-ensemble des transactions ACE qui ne nécessitent pas d'être copiées dans un cache.

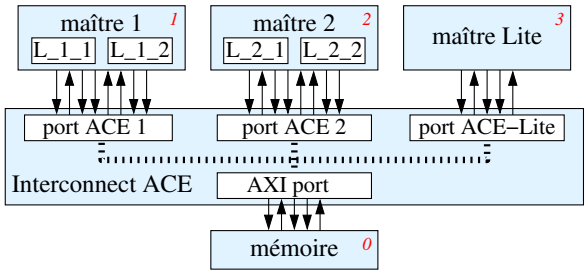

Fig. 1. Architecture du modèle

III. Modélisation formelle d'un SoC ACE

Nous avons utilisé le langage de spécification LNT [3], pour modéliser formellement un SoC conforme à ACE, composé d'un interconnect, de maîtres et d'esclaves. Notre modèle formel (environ 3200 lignes de code LNT) représente le comportement du système en mettant l'accent sur les interactions entre les composants. Il est paramétrique et peut être instancié avec des configurations différentes (nombre de maîtres, nombre de lignes de cache, nombre de lignes mémoires, transactions initiables par les maîtres, etc.). Chaque opération sur un canal est modélisée par un rendez-vous LNT.

Exemple : La Figure 1 illustre le modèle d'un SoC composé d'un interconnect, deux maîtres ACE, un maître ACE-Lite et une mémoire partagée. Chaque maître ACE contient deux lignes de cache. La mémoire partagée contient trois lignes de mémoire. Il a été observé [6], qu'une telle configuration permet de découvrir les principales erreurs qui peuvent survenir dans un protocole de cohérence de caches.

Chaque canal ACE est modélisé par une porte LNT. Les portes LNT sont typées, précisant le nombre et les types des paramètres, c'est à dire, les valeurs échangées sur un rendez-vous.

A. Composants maîtres et esclaves

La mémoire partagée (esclave ACE) est modélisée par un processus LNT.

Comme les lignes de cache d'un maître ACE sont essentiellement indépendantes les unes des autres, c'est à dire, les transactions sur les différentes lignes de cache peuvent librement s'entrelacer,[1] nous choisissons de modéliser chaque maître par une composition parallèle de lignes de cache.

[1]La seule contrainte est de ne pas stocker la même ligne mémoire dans plus qu'une ligne de cache du même maître.

trans. permise			contraintes	taille du STE		propriétés				
M1	M2	Lite	globales	états	transitions	φ_1	φ_2	φ_3	φ_4	φ_5
$\mathcal{A}$	$\emptyset$	$\mathcal{A}$	yes	7,518,552	21,227,610	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_1	$\emptyset$	S_1	yes	3,685,311	10,649,422	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_1	$\emptyset$	S_1	no	3,127,707	9,121,134	$\surd$	$\surd$	$\times$	$\times$	$\times$
S_2	S_2	$\emptyset$	yes	3,545,801	11,122,536	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_2	S_2	$\emptyset$	no	2,819,505	9,095,620	$\surd$	$\surd$	$\times$	$\times$	$\surd$
S_3	$\emptyset$	S'_3	yes	1,834,195	5,170,829	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
S_3	$\emptyset$	S'_3	no	1,437,412	4,547,398	$\surd$	$\surd$	$\surd$	$\surd$	$\times$

Les sous-ensembles des transactions autorisées sont définis ainsi :

- $\mathcal{A}$: Toutes les transactions ACE (resp. ACE-Lite)
- S_1 : {*MakeUnique*, *ReadOnce*, *ReadUnique*, *WriteBack*}
- S_2 : {*MakeInvalid*, *MakeUnique*, *ReadShared*, *ReadUnique*, *WriteBack*}
- S_3 : {*MakeUnique*, *WriteBack*}, S'_3 = {*ReadOnce*}

Tableau I
GÉNÉRATION DES STE ET VÉRIFICATION

Le modèle LNT d'un maître ACE-Lite est obtenu à partir du modèle d'un maître ACE en supprimant la manipulation des demandes de fouines.

B. Interconnect cohérent (CCI)

Pour permettre tous les entrelacements entre transactions sur les ports du CCI, nous modélisons le CCI par une composition parallèle d'autant de processus qu'il y a de ports.

Exemple : Le CCI de la figure 1 contient quatre processus ports : deux ports ACE, chacun connecté à un maître ACE, un port ACE-Lite connecté à un maître ACE-Lite, et un port AXI connecté à la mémoire partagée.

C. Exigences relatives à l'ordre globale des transactions

La spécification ACE comprend certaines exigences globales pour la mise en œuvre de tout interconnect conforme à ACE, qui concernent principalement la cohérence globale du SoC. Par exemple la spécification ACE exige que deux écritures reçues par l'interconnect dans un certain ordre soient écrites dans la mémoire dans le même ordre. Suivant un style de spécification orienté contraintes [2], notre modèle LNT intègre ces exigences globales moyennant des processus dédiés, composés en parallèle avec le reste du modèle. Par conséquent, ces processus surveillent le système et ont une vue globale de toutes les transactions.

D. Génération du système de transitions étiquetées

Nous considérons plusieurs configurations de SoCs, chacun composé d'une mémoire partagée, un maître ACE-Lite, et deux maîtres ACE, avec deux lignes de cache pour chacun. Mettant l'accent sur la cohérence, les premières lignes de cache de chaque maître ACE exécutent des transactions relatives à la même ligne mémoire.

Nous avons choisi des sous-ensembles de transaction susceptibles de créer des problèmes pour les propriétés à vérifier. Pour chaque configuration considérée, le tableau I donne la taille du *système de transitions étiquetées* (STE) correspondant. La première colonne (resp., deuxième et troisième) donne l'ensemble des opérations que le maître 1 (resp., le maître 2 et le maître ACE-Lite) est autorisé à initier. La quatrième colonne indique si le modèle inclut les processus représentant les exigences globales; nous générons les STEs pour les modèles sans les processus des exigences globales afin d'étudier leur impact sur les propriétés du système.

IV. VALIDATION

Nous avons vérifié plusieurs propriétés au niveau système sur notre modèle présenté dans l'exemple. Nous exprimons ces propriétés dans le langage MCL (*Model Checking Language*) [11]. L'outil EVALUATOR 4.0 de CADP [8] peut vérifier les propriétés MCL à la volée.

A. L'exécution complète des transactions

Pour vérifier que chaque transaction finit inévitablement, nous utilisons deux formules de vivacité.[2]

```
[ true * . { AR ?op:String ?c:Nat ?m:Nat ... } ]
    inev ( { R !op !c !m } )
```

La formule φ_1 ci-dessus exige que chaque lecture sur le canal `AR` donnera lieu necessairement à une réponse correspondante sur le canal `R`. MCL permet de capturer la transaction ACE en cours (`op`), l'indexe du master (`c`) et la ligne mémoire en question (`m`). La formule φ_2 pour les écritures est définie de manière similaire.

B. La cohérence de caches

Pour vérifier la cohérence des états ACE des caches du système, on renomme les transitions en utilisant une porte unique `G` en gardant seulement les paramètres utiles. Ceci permet de simplifier la formule et réduire la complexité de la vérification.

La cohérence de caches est exprimée par deux formules de sureté. La première (φ_3, présenté ci-dessous), exige que si une ligne de cache `l1` est dans l'état ACE_UD alors toute autre ligne de cache `l2` contenant la même ligne mémoire `m` doit être dans l'état ACE_I:

```
[ true * .
   {G ?l1:Nat ?m:Nat ?s:String
    where s="ACE_UD"} .
   ( not ({G !l1 !m ?s:String
     where <>"ACE_UD"}) ) * .
   {G ?l2:Nat !m ?s:String
    where (l2<>l1) and (s<>"ACE_I")}
] false
```

Une deuxième formule φ_4 (similaire à φ_3), exige que si une ligne de cache est dans l'état ACE_SD alors toutes les autres lignes de cache contenant la même ligne mémoire doivent être dans l'état ACE_SC ou l'état ACE_I.

C. L'intégrité des données

Afin de vérifier l'intégrité des données du système, nous utilisons la propriété de sureté φ_5, qui impose un ordre correct des écritures dans la mémoire partagée:

[2]Ces formules utilisent la macro `inev (L)`, qui exprime que l'action `L` finira par se produire. Cette macro peut être définie comme suit:

```
macro
  inev(L) = mu X . ( < true > true and [ not L ] X )
end_macro.
```

```
[  true * .
  { W !"WRITEBACK" ?c:Nat ?m:Nat ?d:Nat } .
  ( not { W !"WRITEBACK" !"0" !m !d !c } ) * .
  { W !"WRITEBACK" !"0" !m !d !c } .
  (
    ( not { AC ... !c ... !m } ) and
    ( not { W ... !"0" !m ... } )
  ) * .
  { W ... !"0" !m ?h:Nat ... where h<>d }
] false
```

Après l'initialisation d'une écriture en mémoire d'une donnée `d` par un maître `c`, la propriété interdit l'écriture d'une donnée `h` différente de `d` dans la ligne mémoire `m` sans avoir reçu auparavant une demande de fouine concernant la ligne `m`.

D. Résultats de Vérification du Modèles

Les résultats de la vérification de ces propriétés sur les STEs de la section III-D sont présentés dans les colonnes sept à onze du tableau I. Tout STE comprenant les exigences globales satisfait ($\surd$) les cinq propriétés. Les STE sans exigences globales peuvent ne pas satisfaire les propriétés de cohérences et d'intégrité des données, dans quel cas, EVALUATOR 4.0 génère des séquences de contre-exemples minimaux. Ceci est intéressant pour STMicroelectronics, car ces séquences correspondent à des scénarios à tester sur toute mise en œuvre industrielle de ACE, facilitant ainsi le test de la mise en œuvre (complexe) des contraintes de cohérence dans un interconnect.

V. TRAVAUX VOISINS

Les techniques de vérification formelle, notamment de vérification symbolique des modèles ont été souvent appliquées à la vérification des conceptions des protocoles matérielles de cohérence de caches [13], [10], en utilisant différents langages de modélisation, logiques temporelles, et vérificateurs de modèle. La plupart des travaux [4], [5], [7], [9], [12], [14], [15] concerne des topologies plus complexes que la topologie entièrement connectée que nous avons choisie, mais la différence principale avec notre travail est que nous étudions un SoC hétérogène, en nous concentrant sur un interconnect générique qui inclut le comportement de *toutes* implémentations correctes — plutôt que de vérifier un protocole de cohérence particulier pour un système homogène.

VI. CONCLUSION

Nous avons développé un modèle formel LNT générique d'un SoC prenant en compte la cohérence de caches au niveau système conformément à la spécification ACE [1]. Le style de spécification orienté contraintes s'est avéré utile dans la modélisation des exigences globales. Nous avons exprimé les propriétés du système avec des formules de logique temporelle et nous les avons vérifiés automatiquement. Ainsi, les techniques de vérification formelle peuvent être utilisées pour l'analyse des systèmes sur puce cohérents hétérogènes au niveau industriel.

Ce travail peut être poursuivi selon plusieurs directions. D'une part, le modèle d'interconnect générique peut être utilisé pour analyser l'impact d'un interconnect cohérent dans un modèle de SoC concret. D'autre part, le modèle formel peut être utilisé pour guider les tests et la validation d'un SoC concret, en jouant le rôle d'un modèle de référence pour la co-simulation ou par l'extraction automatique des scénarios de tests intéressants. STMicroelectronics a exprimé son intérêt pour les deux perspectives, vu leurs impacts dans le développement des produits futurs.

RÉFÉRENCES

[1] ARM. *AMBA AXI and ACE Protocol Specification*, Feb. 2013. version ARM IHI 0022E, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022e`.

[2] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems*, 14(1):25–59, Jan. 1988.

[3] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LOTOS NT to LOTOS translator (version 5.8). INRIA/VASY, 155 pages, Mar. 2013.

[4] G. Chehaibar. Integrating formal verification with Murϕ of distributed cache coherence protocols in FAME multiprocessor system design. In *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE 2004 (Madrid, Spain)*, *LNCS* 3235, pp. 243–258. Springer, Sept. 2004.

[5] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, Mar. 1995.

[6] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors ICCD'92*, pp. 522–525. IEEE, Oct. 1992.

[7] A. T. Eiríksson and K. L. McMillan. Using Formal Verification/Analysis Methods on the Critical Path in System Design: A Case Study. In *Proceedings of the 7th International Conference on Computer Aided Verification CAV (Liège, Belgium)*, *LNCS* 939, pp. 367–380. Springer, July 1995.

[8] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2011: A toolbox for the construction and analysis of distributed processes. *Software Tools for Technology Transfer*, 15(2):89–107, Apr. 2013.

[9] H. K. Kapoor, P. Kanakala, M. Verma, and S. Das. Design and formal verification of a hierarchical cache coherence protocol for NoC based multiprocessors. *The Journal of Supercomputing*, 2013.

[10] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, Apr. 1999.

[11] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland)*, *LNCS* 5014, pp. 148–164. Springer, May 2008.

[12] K. L. McMillan and Schwalbe. Formal Verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, pp. 242–251, 1991.

[13] F. Pong and M. Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Computing Surveys*, 29(1):82–126, Mar. 1997.

[14] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study. In *Proceedings of the 1st International Conference on Parallel Processing EURO-PAR'95 (Stockholm, Sweden)*, *LNCS* 966, pp. 287–300. Springer, Aug. 1995.

[15] U. Stern and D. L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Correct Hardware Design and Verification Methods*, *LNCS* 987, pp. 21–34. Springer, 1995.

[16] C. Thompson. *Verifying Cache Coherency Protocols with Verification IP*. Synopsis, Oct. 2012.