Réutilisation : objets ou composants ?

Frédéric Mallet* et Fernand Boéri.

Laboratoire Informatique, Signaux et Systèmes (I3S) – UMR 6070 CNRS-UNSA, IUT de Nice-Sophia Antipolis.
2000 rte des Lucioles – Bat. « Euclide B ».
06903 – Sophia Antipolis Cedex
boeri@unice.fr

** Institute for Computing Systems Architecture - University of Edinburgh*
The King's Buildings – JCMB – Mayfield Road
EH9 3JZ – Edinburgh, UK
fmallet@dcs.ed.ac.uk

1. Résumé

Cet article présente une technique originale, orientée objet de modélisation d'architectures matérielles. L'objectif est de modéliser d'une part la structure de données avec un langage objet qui propose des mécanismes de réutilisation élaborés. D'autre part, l'interface physique est réalisée avec un langage de description du matériel qui propose des mécanismes de communication et de synchronisation appropriés.

Architectures de processeurs, modélisation orientée objet, composants.

2. Introduction.

Les architectures de processeurs actuelles intègrent de plus en plus de fonctions élémentaires et deviennent de plus en plus difficile à modéliser. Les ASICs (Application Specific Integrated Circuits) intègrent plusieurs processeurs et co-processeurs. Les architectures parallèles sont constituées de plusieurs dizaines de ces ASICs pour atteindre des puissances de calcul allant jusqu'au tera flops. Hier les architectures traitaient généralement des données de 8, 16 ou 32 bits. Aujourd'hui, les architectures hautes performances manipulent 64 bits ou deviennent VLIW pour augmenter le parallélisme (256 bits). Ces aspects doivent passer au second plan, avant des méthodes qui se concentrent sur les fonctions.

La réutilisation est indispensable, mais que faut-il réutiliser ? Les concepteurs d'ASICs veulent réutiliser les architectures existantes, en

combiner de plus en plus avec des configurations à la demande pour fournir la fonctionnalité demandée au moindre coût. Les développeurs des outils de support souhaitent, pour leur part, pouvoir réutiliser leurs modèles existants au maximum et les enrichir à chaque fois un petit peu pour qu'ils réalisent les nouvelles fonctions.

Les langages objets deviennent alors très intéressants avec en particulier les mécanismes d'héritage et de composition qui permettent d'enrichir relativement facilement les modèles bien conçus.

Les composants pour leur part permettent d'encapsuler par une interface de communication une ou plusieurs fonctionnalités. Les composants communiquent par l'intermédiaire de connecteurs qui ont l'avantage indiscutable sur les objets de faire apparaître sur un pied d'égalité l'émetteur et le récepteur. Il est alors aisé de changer le connecteur et l'implémentation du protocole qu'il supporte sans modifier ni l'émetteur, ni le récepteur.

Les langages de description du matériel (VHDL ou Verilog) sont orientés composants. Ils permettent la description de l'interface de communication ou entité, puis de plusieurs implémentations différentes (composant) et enfin de la configuration. Ils manquent néanmoins de mécanismes d'enrichissement des composants existants. C'est ainsi que des nouveaux langages sont apparus, pour appliquer les propriétés de réutilisation des objets aux composants : OO-VHDL (Swamy, 1995), (Barton & Berge, 1996), (Benzakki & Djaffri, 1997), (Peterson, 1999).

Cet article soutient un autre point de vue qui consiste à combiner les deux approches. D'une part, utiliser des langages orientés objets tels que Java ou C++ pour décrire les fonctions (services) élémentaires réalisées par les architectures matérielles. Ces modèles peuvent alors être enrichis et être adaptés à de nouvelles spécifications. D'autre part, utiliser un langage de description d'architecture (Medvidovic & Taylor, 2000) pour décrire les entités elles-mêmes et les mécanismes de communication et de synchronisation. Cette description matérielle utilise les modèles objets définis précédemment pour réaliser le comportement.

Nous défendons ce point de vue avec l'exemple de la modélisation d'un cache en établissant un modèle objet logiciel utilisé par la suite pour réaliser un modèle de simulation d'une architecture matérielle de type RISC.

3. Modélisation d'un cache.

La plupart des systèmes à processeurs intègrent un ou plusieurs caches de données et d'instructions. Cependant, suivant la précision du modèle que l'on veut réaliser, l'interface de communication du composant cache peut

être très différente. Parfois, on peut même souhaiter ne pas visualiser le trafic entrant et sortant du cache mais uniquement le gain de performance en fonction de la taille du cache, de son organisation (direct, complètement associatif, partiellement associatif) ou de la politique de substitution (aléatoire, FIFO, LRU : moins récemment utilise). Le modèle matériel réalisé peut être très différent mais son comportement logiciel ne changera pas.

Nous proposons donc un modèle logiciel du cache et qui peut utilisé pour construire des modèles de composants matériels. Nous montrons comment l'utilisation des paradigmes objets permet une modélisation efficace indépendante des différentes stratégies de cache.

Notre point de vue, consiste à défendre l'idée que dans un premier temps les langages objets doivent jouer leur rôle pour modéliser efficacement la structure de donnée représentée. Dans un deuxième temps, le langage de description du matériel prend le pas pour la description des communications et la synchronisation des services.

3.1 Interposition.

Le premier point important est que le cache doit venir s'interposer entre la mémoire et l'utilisateur sans modifications. Il est donc indispensable que le cache ait la même interface de communication que la mémoire. La notion d'interface du langage Java est ici très appropriée. Si nous définissons l'interface *ProtocoleMemoire* avec les deux services *read* et *write*. A la fois le cache et la mémoire devront implémenter cette interface. Pour être général, le service de lecture (read) prend une adresse représentée sous la forme d'un ensemble de bits dont on ne connaît pas la taille a priori (BitSet) et retourne une donnée (Data). Le service d'écriture (write) a comme paramètre une adresse et une donnée, il retourne une donnée qui représente la victime, c'est à dire la donnée qui est éventuellement remplacée.

3.2 Organisation du cache.

Le cache doit essentiellement éviter les accès à la mémoire principale qui, la plupart du temps, est plus lente et hors du cœur. Pour cela, il s'agit, à chaque accès mémoire de s'assurer que la donnée accédée, identifiée par son adresse, n'est pas présente dans le cache. Si la donnée est présente et valide, elle est retournée. Si la donnée n'est pas présente alors il s'agit d'abord d'effectuer un accès à la mémoire principale, puis ensuite de trouver une place dans le cache pour mémoriser cette nouvelle donnée.

Différentes organisations sont possibles. L'idée de base est de réaliser un cache complètement associatif. La donnée peut être mémorisée n'importe où dans le cache, il faut parcourir tout le cache pour la retrouver. Cette politique

peut être pénalisante car toutes les opérations sur le cache sont alors proportionnelles à la taille du cache.

Une autre organisation possible consiste à réaliser un cache partiellement associatif. Chaque donnée ne peut être mémorisée que dans une partie du cache déterminée en fonction de son adresse. Les opérations sur le cache sont alors proportionnelles à la taille de la partie réservée. En contre partie, les performances du cache sont réduites si la majorité des accès à la mémoire sont concentrés sur une partie du cache en particulier.

Enfin, si l'on souhaite avoir des accès au cache constant à temps constant, il est possible de réduire la plage accessible pour chaque adresse à une seule ligne. Le cache est alors à accès direct, les opérations sont à temps constant mais le système de cache peut s'avérer être très inefficace suivant la répartition des accès à la mémoire.

Un accès au cache consiste en définitive à parcourir un certain nombre de lignes dans le cache, éventuellement une seule pour les accès direct, puis pour chaque ligne parcourue, utiliser le 'tag' inscrit sur la ligne pour identifier la donnée du cache avec l'adresse accédée. Si le tag correspond a l'adresse, alors l'accès est concluant (hit) sinon, la donnée doit être demandée à la mémoire principale (miss).

A chaque accès au cache, la méthode *acces* est appelée pour déterminer si la donnée est présente dans le cache ainsi que le numéro de la ligne à laquelle elle correspond. L'algorithme est alors le suivant :

```
int acces(BitSet adresse) {
   BitSet tag = organisationManager.extraitTag(adresse) ;
   Plage p = organisationManager.extraitPlagePossible(adresse) ;

   for (int i=p.debut ; i<p.fin ; I++)
          if (getLigne(i).correspondA(tag))
                    return i ;
   return MISS ; // MISS est une constante hors des limites du cache (ex : -1).
}
```

L'extraction du tag dans l'adresse et le calcul de la plage du cache dans laquelle la donnée peut être mémorisée est déléguée à l'objet responsable de l'organisation (*organisationManager*) qui implémente l'interface suivante :

```
public interface OrganisationManager {
  public BitSet extraitTag(BitSet adresse) ;
  public Plage extraitPlage(BitSet addresse) ;
}
```

A partir de là, différentes organisations sont implémentées en définissant une classe par organisation :

1. Pour l'accès direct, l'adresse est décomposée en deux parties. La partie basse est utilisée pour déterminer la ligne du cache dans laquelle la donnée peut être mémorisée (adresse % nombre_de_lignes). Le reste (adresse / nombre_de_lignes) est utilisé comme identifiant (tag) :

```
public AccesDirect implements OrganisationManager {
  public BitSet extraitTag(BitSet adresse) {
          return adresse.div(nombre_de_ligne) ;
  }
  public Plage extraitPlage(BitSet adresse) {
           int nLigne = adresse.modulo(nombre_de_ligne).intValue() ;
          return new Plage(nLigne, nLigne) ; // une seule ligne.
  }
}
```

2. Pour le cache complètement associatif. Il faut parcourir tout le cache, toute l'adresse est alors utilisée comme identifiant :

```
public ComplementAssociatif implements OrganisationManager {
  public BitSet extraitTag(BitSet adresse) {
          return adresse ;
  }
  public Plage extraitPlage(BitSet adresse) {
          return new Plage(0, nombre_de_ligne-1) ; // tout le cache
  }
}
```

3. Pour le cache partiellement associatif. La partie basse de l'adresse sert à déterminer la partie dans laquelle la donnée peut être mémorisée, le reste sert d'identifiant.

```
public ComplementAssociatif implements OrganisationManager {
  public BitSet extraitTag(BitSet adresse) {
          return adresse.div(nombre_parties) ;
  }
  public Plage extraitPlage(BitSet adresse) {
          int lPartie=adresse.modulo(nombre_parties).intValue() * taille_parties ;
          return new Plage(lPartie, lPartie+taille_parties-1) ;
  }
}
```

4. Conclusion.

Le modèle de cache proposé est décrit de façon purement logicielle. Le choix de la politique de remplacement dans le cache peut être très facilement modélisée de la même manière par délégation.

Le rôle du langage de description du matériel (HDL) est alors de proposer les mécanismes pour créer l'entité matérielle ainsi que les modes de communication. Aucun HDL ne permet de réaliser le modèle logiciel proposé, mais de plus en plus d'approches permettent une approche mixte avec du C++ ou du Java.

(Mallet, 2000) propose une méthode semi-automatique pour générer un composant à partir d'une classe en associant les méthodes construites à des ports physiques et en choisissant le protocole physique qui déclenchera l'exécution de la méthode. Par exemple, un front montant sur un port *write* du cache, déclenche l'appel de la méthode *acces*. Si la donnée est présente, elle est retournée, les connecteurs mis en place par le langage de description du matériel permettent de modéliser les délais de propagation. Sinon, un port de sortie est excité pour déclencher un accès à la mémoire principale.

5. Références bibliographiques.

(Barton & Berge, 1996) « A proposed Design Objectives Document for Object-Oriented VHDL. ». The RASSP Digest - Vol. 3, septembre 1996. http://rassp.aticorp.org/newsletter/html/96sep/news_18.html

(Benzakki & Djaffri, 1997) « Object-Oriented Extensions to CHDL : The LaMI Proposal ». IFIP 1997. Chapman & Hall. p. 334-347.

(Mallet, 2000) « Modélisation et Evaluation de Performances d'architectures matérielles numériques. ». Thèse de doctorat, Université de Nice-Sophia Antipolis, décembre 2000.

(Medvidovic & Taylor, 2000) « A classification and comparison Framework for Software Architecture Description Languages. », IEEE Transactions on Software Engineering, Vol.26, No. 1, janvier 2000.

(Peterson, 1999) « Proposed Language Requirements for Object-Oriented Extensions to VHDL. ». G.D. Peterson, Proc. of Forum on Design Languages, FDL'99, France, sept. 99.

(Swamy, 1995) « Object-Oriented VHDL Provides New Modeling and Reuse Techniques for RASSP. ». Vista RASSP Program Manager. The RASSP Digest - Vol. 2, No. 1, 1st. Qtr. 1995. http://rassp.aticorp.org/newsletter/html/95q1/news_6.html