ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Méthodologie basée sur des membranes pour la gestion de la reconfiguration dynamique dans les systèmes embarqués parallèles

Pamela WATTEBLED, Jean-Philippe DIGUET

Université Bretagne Sud,
Centre de recherche, BP 92116
56321 Lorient - France
pamela.wattebled-meftali@univ-ubs.fr, jean-philippe.diguet@univ-ubs.fr

---

Résumé
La reconfiguration partielle et dynamique donne une nouvelle dimension pertinente et efficace à la conception des systèmes embarqués parallèles. Toutefois, en raison de la complexité de ces systèmes, assurer la cohérence et la gestion du parallélisme lors de l'exécution reste un défi majeur. Ainsi, des modèles d'architectures et des méthodologies de conception assistée sont nécessaires pour permettre la gestion efficace de la reconfiguration matérielle. Notre approche est inspirée des modèles, à base de composants, bien connus dans le monde du logiciel. Le modèle que l'on propose est basé sur des membranes enveloppant les composants du système. L'objectif est d'améliorer la productivité de conception et d'assurer la cohérence de la gestion des changements de composants virtuels réutilisables (IPs) ainsi que le changement de contexte. Ces membranes sont distribuées et optimisées dans le but de concevoir des systèmes autoadaptatifs.

Mots-clés : Reconfiguration dynamique, FPGA, membrane, sauvegarde de contexte

---

1. Introduction

Les applications évoluent très régulièrement dans de nombreux domaines. Cette évolution combinée au coût important de prototypage (sur ASICs) conduit les ingénieurs vers des solutions matérielles adaptables et flexibles. Ainsi, les FPGAs (Field Programmable Gate Array) apparaissent comme une solution à faible coût, surtout avec l'augmentation significative de portes logiques qu'ils intègrent. Les FPGAs les plus récents prennent en charge la reconfiguration partielle et dynamique, ce qui les rend capables d'intégrer des applications volumineuses. Toutefois, il faut établir une cohérence lors du changement d'IP, ce qui empêche de profiter pleinement de cette fonctionnalité. En effet, le remplacement partiel du bitstream, par un autre, lors de l'exécution est généralement une tâche complexe. Ainsi, le système doit intégrer un mécanisme capable de gérer efficacement la commutation de contexte pour chaque reconfiguration. Le contexte étant toutes les données utiles à l'IP pour reprendre son exécution, dans l'état où il était, en cas d'arrêt de celui-ci pour une reconfiguration. Ainsi, un tel système doit assurer la cohérence, le stockage et la restauration de toutes les informations pertinentes avant de révoquer (respectivement de charger) un bitstream partiel. Ce mécanisme de contrôle doit également être

efficace en minimisant le temps de changement d'IP ou de contexte et enfin le modèle doit être modulaire et flexible, de telle sorte qu'il soit possible de le générer automatiquement. L'objectif de cet article est de présenter l'architecture du support de la reconfiguration dynamique et la méthodologie de sa conception développées dans le projet FAMOUS. Le but étant d'améliorer la productivité (la réutilisation du code) dans la modélisation des systèmes embarqués pour une mise en oeuvre sur des architectures dynamiquement reconfigurables de type FPGAs. Ce papier traite, en particulier, de la conception des composants matériels (HW) pour la gestion du parallélisme et la reconfiguration dynamique. Ils permettent le changement d'IPs en assurant une consistance efficace avec un stockage des données (sauvegarde du contexte) permettant un retour de l'IP dans le même état d'exécution qu'avant son interruption. Cette fonctionnalité est réalisée par un mécanisme matériel distribué pour minimiser les temps de communications au cours de la reconfiguration dynamique. Notre approche est dédiée à des applications de flux de données telles que les réseaux de processus de Kahn (Kahn Process Network ou KPN), soit des tâches sans contraintes fortes de temps réel. Le reste de ce papier est organisé comme suit. La section suivante présente un aperçu du projet FAMOUS, puis sont présentés dans la section 3 quelques travaux importants sur la persistance et le changement de contexte. La section 4 présente l'approche à base de composants ainsi que d'autres concepts nécessaires à la compréhension de ce document. Notre méthode de changement de contexte est détaillée dans la section 5. La section 6 donne quelques résultats expérimentaux et la section 6 conclut ce papier par des perspectives et travaux futurs.

2. Contexte du projet FAMOUS

Ces travaux sont effectués dans le cadre de l'ANR Famous [6], le projet proposé vise à présenter une méthodologie complète qui prend en compte la reconfiguration dynamique matérielle, et propose les mécanismes nécessaires pour exploiter entièrement ces possibilités pendant l'exécution. Famous s'intéresse aux modèles de très haut niveau (UML), méthodes de compilation et d'exécution ainsi que les techniques d'analyse et de vérification. L'objectif est de fournir des outils pour une conception de qualité, améliorants la productivité, tout en garantissant l'optimisation des ressources matérielles utilisées et en réduisant le temps de mise sur le marché. Les travaux présents dans cet article sont un service matériel pour la reconfiguration dynamique (voir FIG.1), ce qui les situe dans la couche la plus en bas du projet Famous.

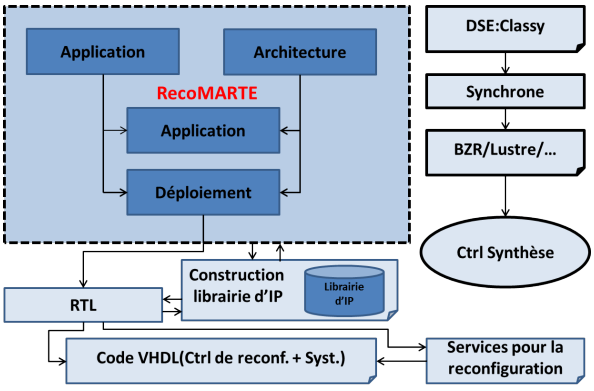

FIGURE 1 – Flot FAMOUS

3. Etat de l'art

De nombreux travaux ont été effectués pour répondre à la difficulté du contrôle et de la gestion du contexte dans les systèmes dynamiquement reconfigurables. Ces travaux peuvent être classés en trois grandes catégories, en fonction des choix d'implémentation :

- Gestion et contrôle effectués avec logiciel : De nombreuses recherches ont été menées pour la reconfiguration dans le monde du logiciel. Dans [9] est proposé un style architectural pour les systèmes temps réel, dans lequel la reconfiguration dynamique est mise en oeuvre par une tâche de contrôle synchrone en réponse aux changements d'état. Pour gérer les dépendances d'exécution entre composants, [11] ont développé un contrôle de reconfiguration logiciel qui analyse à l'exécution les différentes interactions et détermine les interactions qui devraient être autorisées pour continuer. Dans [5], un modèle logiciel hiérarchique auto-adaptatif est présenté, pour les systèmes robot, permettant la tolérance aux pannes. [1] propose une plateforme logicielle orientée services pour traiter certaines options de conception pour la reconfiguration dynamique. [4] présente un framework logiciel orienté service pour aider à la conception des systèmes reconfigurable dynamiquement.[2] propose un paradigme de programmation donnant des lignes directrices spécifiques aux ingénieurs afin d'intégrer des composants logiciels en utilisant des objets basés sur les ports.
- Système d'exploitation : L'objectif des SE est de soutenir le développement et l'exécution de différentes applications sur des architectures données. Ainsi, dans le cadre de reconfiguration dynamique sur FPGA, le système d'exploitation peut gérer les changements de contextes et contrôler les tâches matérielles/logicieles. Ces fonctionnalités rendent ces systèmes d'exploitation beaucoup plus complexes que les SE classiques. [2] présente les questions fondamentales qui peuvent avoir une conséquence sur la conception d'un système d'exploitation pour FPGA reconfigurable dynamiquement. L'auteur propose que les applications soient conçues en noyaux amovibles. [10] et [19] discutent de services système d'exploitation de haut niveau comme support pour la reconfiguration dynamique. Dans [21] et [8] les systèmes d'exploitation sont présentés comme des couches intermédiaires entre le matériel et le logiciel. Le support permettant la reconfiguration dynamique est caché aux concepteurs d'IPs. Ce qui enlève une grande partie de la complexité liée à la reconfiguration dynamique aux concepteurs. Ces approches conduisent à ralentir le système et ne permettent pas de profiter pleinement de l'énorme quantité de portes logique disponible dans les FPGAs.
- Réponse matérielle pour la gestion de la reconfiguration dynamique : Les premiers travaux effectués sur le sujet de la gestion de sauvegarde de contextes sont dédiés aux FPGAs non reconfigurable dynamiquement, parce qu'ils ont été utilisés principalement à des fins d'émulation et de prototypage. Plus récemment, tout commence à changer, de nouveaux travaux ciblant les FPGAs reconfigurables dynamiquement ont été proposés. Par exemple [20] propose des architectures capables de stocker un nombre de contextes fixe pour configurer les IPs. Avec cette approche, le changement de contexte est contrôlé de façon matérielle, il en résulte une réduction de temps, mais cette approche est dépendante de la plateforme, n'est pas flexible et non réutilisable. [14] propose une architecture reconfigurable utilisant une mémoire externe pour stocker les flux binaires et les contextes, mais l'approche demande des transferts de données prenant beaucoup de temps dû au fait que tout est stocké dans une mémoire en dehors de la puce. Dans [7] et [18] les auteurs proposent un mécanisme de gestion de changement de contexte basé sur des registres. Cette approche repose sur l'expérience du concepteur et donc sujette aux erreurs. Elle n'est pas générique et donc dépendante de l'application. Plus récemment, [3] a proposé une enveloppe pour la gestion des sauvegardes de contextes dans le cas de reconfiguration dynamique. Il contient un simple tampon qui envoie

le contexte courant en mémoire globale en cas de reconfiguration dynamique. Le principal inconvénient de cette approche est la dynamicité de l'enveloppe. Cela rend les bistreams plus complexes et peut occasionner une surcharge importante lors de la reconfiguration. De plus l'enveloppe n'est pas générique et dépend de l'IP.

4. Approche à base de composants dans le monde logiciel

Dans le monde du logiciel, les applications auto-adaptatives modifient leur comportement de façon dynamique et autonome en utilisant l'introspection, la recomposition, l'ajout et l'élimination des composants, afin de s'adapter aux changements de leur environnement qui peuvent survenir en cours d'exécution. Le lecteur peut se référer par exemple à [15]. Un composant est vu comme une boîte noire qui peut être modifiée et remplacée par une autre boîte noire ayant les mêmes interfaces (comme dans le monde matériel). Les concepteurs de logiciels sont passés de façon naturelle d'une programmation d'applications à base de tâches (séquentiel) à un modèle de conception à base d'objets [12]. Le passage au monde des modèles basés sur les composants est également une solution évidente pour l'avenir. En fait, ces modèles permettent un très haut niveau d'abstraction en permettant de modifier la granularité du composant. Un composant est divisé en trois parties. La première est la partie métier qui est le contenu de la figure 2, la seconde est la partie technique contenant divers services utilisés par le composant, cette partie est la membrane. Et la troisième partie contient des interfaces permettant la communication avec d'autres composants de l'architecture. Voir Fig. 2. L'approche à base de composants assure une séparation nette entre la technique (contrôle) et la partie métier (la fonction du composant). Ceci constitue l'avantage majeur de ces modèles. La sauvegarde du contexte fait partie de la partie technique et donc dans la méthodologie présente dans ce papier elle est intégrée dans la membrane du composant.

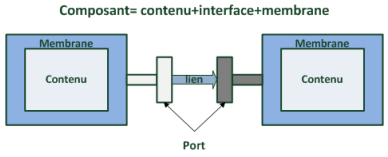

FIGURE 2 – Modèle à base de composants

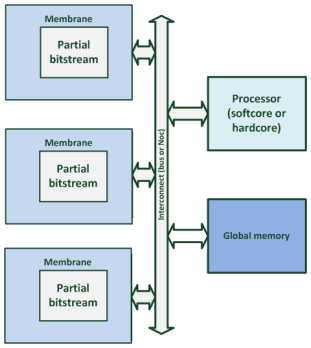

FIGURE 3 – Architecture reconfigurable à base de composants

5. Méthodologie pour la persistance et le changement de contexte

La contribution décrite est un support distribué pour la reconfiguration dynamique partielle matérielle. Ce support permet le changement et le stockage de contexte, ainsi que la reconfiguration d'un système en cours d'exécution. Le modèle proposé est flexible (peut être réutilisé

pour différentes applications), modulaire et générique. Il est inspiré des modèles à base de composants bien connus dans le monde du logiciel. Le support est implémenté comme une membrane matérielle se connectant à l'IP qui est dynamiquement reconfigurable (voir Fig. 3).

5.1. La membrane

Le support de reconfiguration dynamique proposé consiste en des membranes génériques enveloppants les bitstreams partiels. Chaque membrane est attachée à un bitstream partiel à un moment donné. Comme un bitstream partiel est susceptible d'être remplacé par un autre, les sauvegardes de contextes des tâches concernées par la reconfiguration sont effectuées. Ces opérations de sauvegarde peuvent permettre à des tâches d'être restaurées. La granularité de la sauvegarde est choisie par le concepteur d'IP. Il s'agit d'un point important du modèle proposé dans cet article pour la reconfiguration dynamique, voir la section 5.10. Ainsi, la membrane sauvegarde des données, gère le service de changement de contexte ainsi que la reconfiguration dynamique d'IPs. Elle dispose d'une mémoire cache contenant des données de contextes correspondant à différents bitstreams partiels comme illustré dans la figure 4. Les données des contextes ne sont pas examinées ou interprétées par la membrane, elles ne sont vues que comme une succession de bits bruts. Ceci donne un aspect générique aux contextes et donc un contexte peut être utilisé par différents bitstreams partiels. Une telle propriété peut offrir des avantages intéressants dans le cas où par exemple, il est intéressant de changer une tâche «A» par une tâche «B» ayant la même fonctionnalité, mais consommant moins d'énergie. Dans ce cas, le contexte stocké de la première tâche peut être restauré et utilisé par la seconde. Cela permet à «B» de poursuivre l'itération où «A» a été arrêtée. Le second point qui rend les membranes génériques est qu'elles sont attachées aux IPs par une suite de bits non spécialisés ce qui rend les membranes utilisables dans de nombreuses applications sans modifications de celle-ci. Dans la suite les différents éléments composant la membrane sont présentés en détail Fig.4.

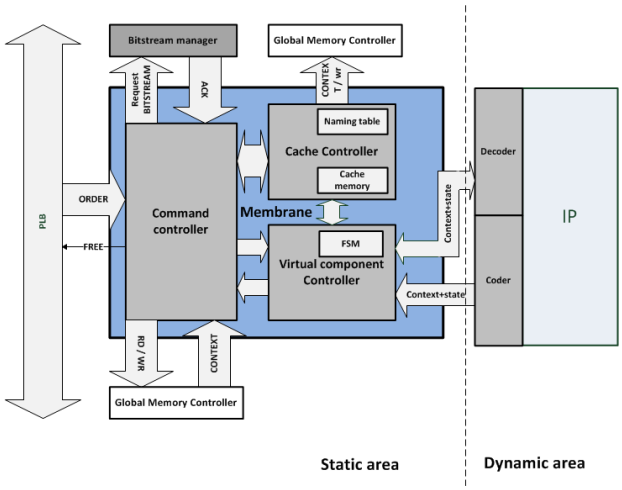

FIGURE 4 – Architecture de la membrane

5.2. Le composant virtuel

Alors que dans le monde logiciel, une membrane est détenue entièrement par un seul composant, notre membrane est statique et peut être attachée à différents composants. Dans le monde

du logiciel, une membrane contient tous les services du composant et le contenu du composant contient la partie métier. Nous gardons cette propriété dans notre travail. Dans notre modèle, nous avons inclus le principe de composant virtuel, qui est composé d'une membrane et un contenu. Chaque membrane peut supporter différents types d'IP et chaque IP peut être fixé à n'importe quelle membrane. Cependant, une membrane donnée ne peut être fixée à plus d'un IP en même temps. Il en résulte le principe de composant virtuel. La membrane est responsable du contrôle de la reconfiguration dynamique et de la sauvegarde du contexte. Son contenu, qui est notre IP (bitstream), est une boîte noire et contient la partie métier du composant. Le contenu est implémenté sur la partie dynamique du FPGA et non sur la partie statique. Ce qui diffère du monde logiciel où la membrane appartient à un seul composant et ne peut pas s'en détacher. Chaque composant doit avoir des points de sauvegardes, servant à enregistrer le contexte voir section 5.10. Une propriété importante de notre modèle est que l'IP peut reprendre, après une reconfiguration dynamique, son exécution dans le dernier état de sauvegarde. L'autre avantage est qu'un contexte sauvegardé peut être utilisé par différents IPs. Ceci permet de remplacer un IP consommant beaucoup d'énergie par un autre consommant moins en permettant au nouvel IP de commencer son exécution da le même état où l'ancien avait été arrêté.

5.3. Commandes entrants dans la membrane

La communication avec la membrane se fait avec deux types d'ordres. Le premier type consiste à envoyer directement un ordre de nouvelle configuration. La membrane est alors informée qu'une tâche donnée, avec un contexte spécifique, doit être exécutée. Dans ce cas, la membrane vérifie si cette tâche est déjà en cours d'exécution et si elle a le bon contexte. Si le bitstream courant attaché à la membrane n'est pas le bon, la membrane adresse une demande de bitstream au gestionnaire de bitstream. Puis si le contexte demandé est manquant, une demande en mémoire globale est effectuée. Le second type d'ordres possibles dans le modèle est d'envoyer directement un ordre à la membrane. Ce mode peut être utilisé lorsque l'application en cours d'exécution a des connaissances sur les contextes présents dans la membrane ainsi sur le bitstream qui est en cours d'exécution. Les différents ordres possibles peuvent être : changement de contexte courant, changement de l'IP courant, changer IP et contexte, arrêter l'IP.

5.4. Mémoire cache

Le modèle de la membrane intègre une petite mémoire cache permettant le stockage de contextes. Lorsque cette mémoire est pleine, les contextes sont déplacés vers la mémoire globale. Ce mécanisme améliore les performances de manière significative, en terme de temps de transfert de données lors des reconfigurations (dépendante de l'application). Le cache est associatif avec une stratégie de remplacement LRU (Least Recently Used) et contient deux lignes. La longueur d'une ligne de mémoire dépend du besoin de l'application. Dans le cas où les contextes ont des tailles importantes (par exemple en traitement de l'image), seule l'adresse mémoire (mémoire globale) du contexte est sauvegardée dans le cache. Ceci permet d'éviter des mémoires locales volumineuses dans les membranes. La taille de la membrane dépend de la taille des contextes.

5.5. FSM du composant Virtuel

Le modèle proposé contient une machine à états finis (FSM,) avec 9 états, détaillé à la figure 5. Cette FSM est intégrée dans la membrane. Les transitions d'un état à l'autre sont assurées par différents événements. Comme le montre la figure 5, nous pouvons distinguer trois déclencheurs. Le premier déclencheur est le contrôleur de configuration de l'architecture extérieur à la membrane qui envoie des ordres. Le second déclencheur est l'IP (par exemple si l'exécution courante est terminée). Le troisième déclencheur est la membrane elle-même quand elle reçoit

un ordre, ou qu'elle termine un ordre ou certaines des étapes de l'ordre. La figure 5 montre les différents messages envoyés par les déclencheurs. Par exemple, pour passer de l'état 'init' à l'état de 'run / backup' un événement est déclenché par l'IP en envoyant le message "backup".

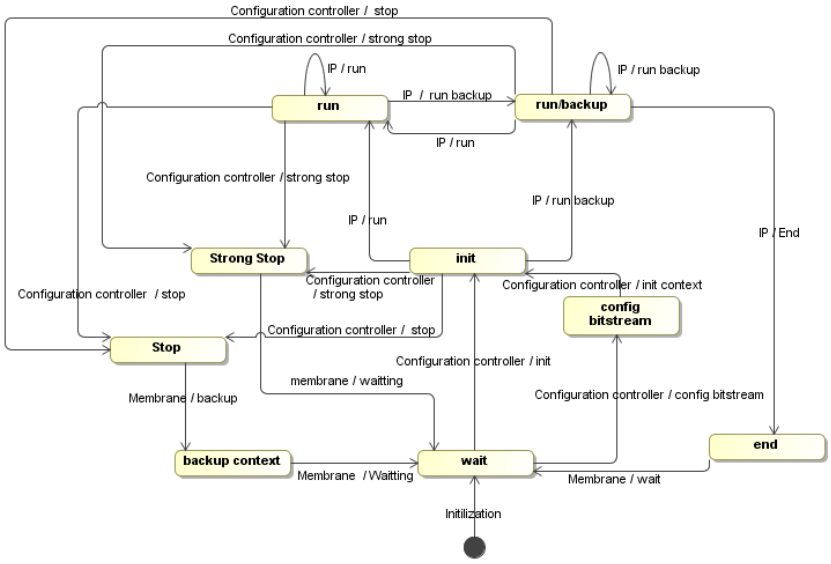

FIGURE 5 – Diagramme UML de la FSM du composant virtuel

5.6. Contrôleurs

Les contrôleurs sont mis de façon à englober les composants qu'il contrôle. Ainsi, le contrôleur de mémoire contient la mémoire cache et la table de nommage.

5.7. Controleur de commande

Après avoir reçu une commande de la part du contrôleur de configuration externe développée dans le projet FAMOUS [17], le contrôleur de commande demande au contrôleur de mémoire cache de vérifier si le contexte concerné est présent dans le cache. En cas de défaut de cache, la demande du contexte sera envoyée à la mémoire globale. Le contrôleur de commande va également vérifier que l'IP attaché à la membrane est celui demandé. Si ce n'est pas le cas, une demande de transfert de bitstream est effectuée au gestionnaire de bitstream. Quand les vérifications sont effectuées, le contrôleur de commande envoie un ordre d'initialisation au contrôleur du composant virtuel.

5.8. Contrôleur du composant virtuel

Le contrôleur du composant virtuel contient la FSM de ce dernier ainsi que son état courant dans un registre. Ce registre est mis à jour par le contrôleur selon les différents ordres reçus du contrôleur de commande, ou si aucun ordre n'est arrivé une vérification sur un éventuel ordre en provenance de l'IP (sauvegarde ou d'arrêt de l'IP). Dans ce dernier cas le contrôleur de commandes met le registre d'état en mode 'WAIT'.
Les commandes que peut recevoir le contrôleur dans le dispositif sont les suivantes :

– INIT : Initialisation du contexte. Le message contient le code de message, le nom du contexte et le contexte lui-même. Le contrôleur met à jour le registre d'état.
– STOP : arrêter l'IP. Le message ne contient que le message code. Le contrôleur définit le registre d'état dans le mode 'WAIT'.
– RUNBACKUP : IP en cours d'exécution avec des sauvegardes à effectuer. Le message contient le code de message et le contexte à sauvegarder. Le registre d'état est mis à jour par le contrôleur à l'état 'BACKUP'.
– READ : Lecture d'un contexte dans la mémoire cache. Le message contient le code de message et le nom de contexte.
– READ / DISABLE : Lecture d'un contexte dans la mémoire cache et désactive la ligne de cache. Le message contient le code de message et le nom de contexte. Le contrôleur de FSM demande au contrôleur de cache les données et aussi demande la désactivation du contexte.
– DISABLE : Désactive un contexte présent dans le cache. Le message contient le code message et le nom du contexte.

5.9. Contrôleur de la mémoire cache

Le contrôleur commande de la mémoire cache contient la table de nommage et la mémoire cache de la membrane. Il gère 3 ordres : écrire, lire et désactiver une ligne de cache. Deux formes d'écriture sont possibles. Soit les données ne sont pas dans le cache, donc est effectué un chargement à partir de la mémoire globale, ou les données sont déjà présentes dans la mémoire cache et donc l'écriture consiste à faire une mise à jour. Ainsi, afin d'écrire, dans le cache, il faut fournir le nom du contexte, le contexte lui-même et mettre le bit d'écriture à 1. Pour lire un contexte, il suffit de fournir le nom du contexte à lire et mettre le bit lecture à 1.

5.10. Flux de conception pour le concepteur matériel

L'effort du concepteur d'IP est considérablement réduit avec notre méthodologie, en particulier grâce à l'aspect générique de la machine à états finis. Toutefois, le concepteur a toujours de deux tâches en charge. La première est une modification légère de l'IP existant de sorte qu'il sera en mesure d'envoyer son état courant et son contexte (voir 5.10.1). Ces envois sont effectués par des points de sauvegarde [13], ces points sont dépendants de l'application et choisis par le concepteur. La deuxième tâche consiste à assurer la communication entre l'IP et la membrane. Il s'agit de la conception des fonctions codeurs/décodeurspou standardiser les entrées/sorties(I/O) de la membrane(voir 5.10.2).

5.10.1. L'état de l'IP

Certaines transitions entre les états de la FSM du composant virtuel sont dues à des événements en provenance de l'IP (voir Fig.5). De tels événements doivent être intégrés dans la FSM de l'IP, cette intégration est laissée à la charge du concepteur. Nous distinguons trois événements :
– RUN : Lorsque la membrane reçoit l'événement 'RUN' à partir de l'IP, la FSM du composant virtuel met son état à 'RUN'.
– RUNBACKUP : Quand la membrane reçoit l'événement 'RUN / BACKUP' de l'IP, la FSM se déplace vers l'état "RUN / BACKUP". Ds sauvagardessont effectuées
– END : Cet événement avertit la membrane que l'exécution de la tâche courante est terminée, de sorte que la FSM du composant virtuel se met dans l'état 'WAIT'.

5.10.2. Standarsisation des entrées/sorties(I/O)

Cette tâche est due à l'aspect générique de la membrane qui voit un contexte seulement comme une succession de bits. Ainsi, lorsque le contexte est envoyé à l'IP, il doit être divisé en groupes de bits pour former des données réelles.

D'autre part, les sorties de l'IP doivent être regroupés afin de stocker le contexte comme une succession de bits. Le concepteur doit alors implémenter un codeur et un décodeur autour de l'IP. Ceci ne prend pas plus de 5 minutes.

6. Expérimentations

Afin de démontrer l'efficacité et la facilité d'utilisation de notre méthodologie, nous avons choisi une application simple qui ne montre pas un effort important de conception d'IP, mais permet de décrire le comportement, l'intégration et les performances de la membrane. Ainsi, l'application choisie consiste en un calcul de factoriel à partir d'un IP matériel. L'entrée (n) et sortie (n !) de cette IP sont des entiers 32 bitsAinsi, l'IP possède trois états possibles :
– Initialisation : recevoir l'entier 'n' comme entrée.
– Fin d'une itération : à la fin de l'itération 'i', le produit des entiers de 'n' à 'i' a été calculé.
– Fin de la tâche : l'IP a terminé le calcul du factorielle.
Ainsi, le contexte est constitué de deux données pour cette tâche, l'entrée 'n' qui est l'itération courante et l'entier résultant de la fin de chaque itération. Ces valeurs permettent le calcul de factoriel, à partir du résultat de l'itération où le système a été arrêté. Dans ce cas, il est nécessaire de stocker un contexte composé de deux entiers de 32 bits. Il nous faut donc 64 bits pour le stocker le contexte complet.

6.1. Intégration du support pour la reconfiguration dynamique

Afin d'intégrer notre support de reconfiguration dynamique, l'IP factoriel a été modifiée en intégrant trois éléments : une petite FSM, un codeur et un décodeur de contexte comme illustré dans la figure 6). Le décodeur prend les 64 bits du contexte, les sépare en deux, les 32 premiers bits pour l'entier à calculer et les 32 autres bits pour les calculs intermédiaires, le codeur applique la transformation inverse. Une membrane générique a été également ajoutée au système afin de contrôler et de gérer le stockage des contextes et les changements de ceux-ci.

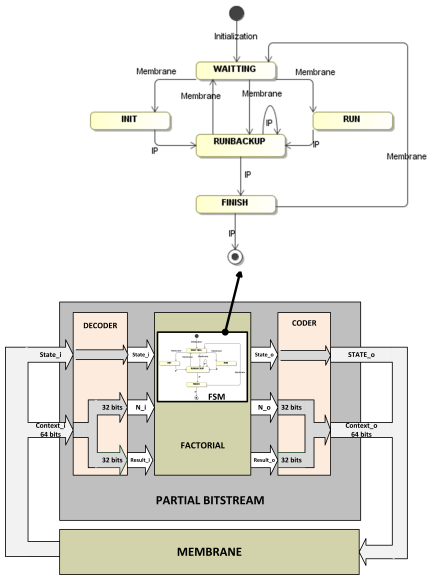

FIGURE 6 – Architecture à base de membrane pour une application factorielle reconfigurable

6.2. Scenarii de simulation

L'objectif de notre simulation est de montrer le bien-fondé de l'approche proposée, en terme de fonctionnalité. Plusieurs scénarios ont été réalisés, mais nous avons choisi de présenter un cas relativement simple afin de rester dans cette section claire et facilement compréhensible. Ainsi, le scénario consiste, dans un premier temps, à calculer la factorielle de 9. Comme le montre la figure 7, la membrane, à l'état INIT, envoie un message de deux parties, à l'IP. 'INIT' pour informer l'IP qu'il reçoit une nouveau contexte, et le contexte lui-même contenant 9 et 1(résultat intermédiare mis à 1 au départ), 9 représente le nombre pour lequel on veut calculer la factorielle. C'est deux nombre sont concaténé avant de rentrer dans le décodeur. Ainsi, l'IP reçoit 'INIT', 9 et 1 que le décodeur de contexte a séparé. Il effectue la première itération, se met dans l'état 'RUN / BACKUP' et envoie son état de sortie ainsi que le résultat intermédiaire qui est 8 et 9 (il reste 8 itérations, et le résultat intermédiaire est 1*9=9). Le codeur concatène codeur les deux nombre et envoie le résultat ainsi que l'état de l'IP à la membrane comme le montre le message 4 fig 7. Après avoir reçu ce message, la membrane sauvegarde le contexte et modifie son propre état pour le mettre à 'RUN /BACKUP '. L'IP poursuit son exécution et le décodeur continu à envoyer les résultats intermédiaires par les ports de sortie de l'IP. Après la deuxième itération, le système a besoin de faire un calcul de plus haute priorité et demande à l'IP de calculer factoriel de 3. Ainsi, la membrane arrête l'exécution en cours en envoyant le message "WAITING" à l'IP (message 7). Ensuite, l'IP se met dans l'état ''WAIT' et arrête son exécution en cours. De même que pour le message 1, le message 9 initialise un nouveau contexte d'exécution et l'IP commence son exécution afin de calculer factorielle 3. Après deux itérations de l'exécution est terminée, les résultats et l'état ''Finish' sont envoyés par le codeur à la membrane (message 14). Après avoir reçu le message 'FINISH', la FSM du composant virtuel passe à l'état 'WAIT'. Le calcul de la factorielle 9 peut alors être repris. Dans le message 15, nous reprenons le calcul avec 72 comme résultat intermédiaire et 7 comme le nombre d'itérations restantes à calculer. L'exécution se poursuit normalement.

6.3. Résultat de synthèse et analyse

6.3.1. Résultat pour l'exemple factorielle

Nos simulations et la synthèse ont été réalisées sur un Virtex 6 FPGA intégré dans une carte ML605. Comme le montre le tableau 1, la membrane utilise très peu de ressources disponibles. Si la mémoire cache est exclu, la membrane nécessite 437 Luts et 26 blocs RAM /FIFO. Ces ressources sont indépendantes de la complexité de l'IP. Le seul surcoût possible et du à la taille de la mémoire cache, puisque la taille des lignes est un paramètre dépendant de la longueur souhaitée d'un contexte, ce qui rend la taille de la mémoire cache variable. Dans l'exemple décrit dans la session, deux lignes la mémoire cache est composée de deux lignes de 64 bits chacune. Ainsi, on observe que le cout de surcharge matérielle due l'implémentation de la membrane n'est pas vraiment significative sur un FPGA Virtex 6. Cela nous permet d'imaginer des architectures reconfigurables hautement parallèles à l'aide de notre approche basée sur des membranes. Comme le montre le tableau 2, une membrane traite quatre commandes possibles, ces commandes proviennent du contrôleur de reconfiguration du système (Microblaze par exemple), ainsi notre membrane prend 3 cycles pour exécuter un ordre STOP, 14 cycles pour l'ordre 'IP RUN + CONTEXTE' en avec un défaut de cache, on demande l'exécution de l'IP avec un contexte, (fréquence d'horloge de 100 Mhz).

6.3.2. Taille de la membrane

La membrane est générique est sa taille dépend de la taille du contexte. Comme nous pouvons le voir dans le tableau 3, la taille de la membrane dépend d'une équation linéaire. Un nombre de

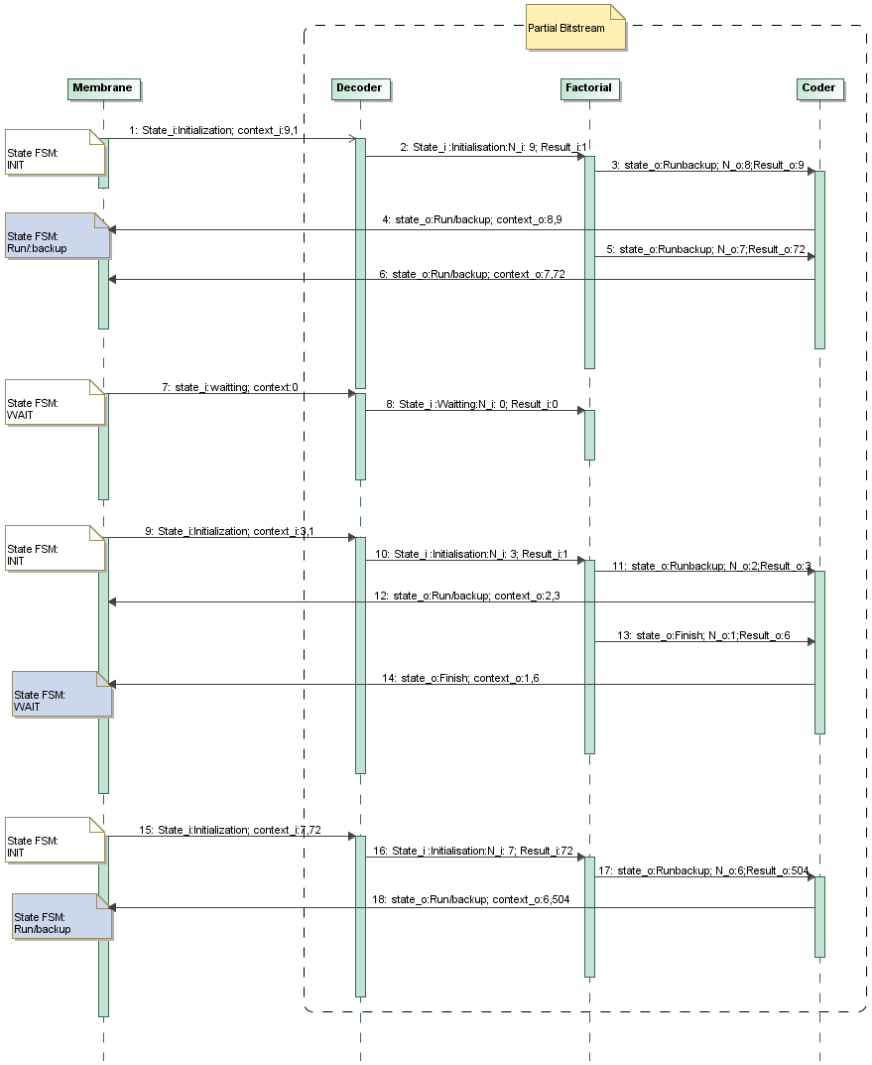

FIGURE 7 – Diagrame UML d'une simulation de calculs factoriels

	Membrane with cache memory		Membrane without cache memory	
Number of slice register	794	0,63%	794	0,63%
Number of slice Luts	552	0,36%	272	0,17 %
Number of of block RAM/FIFO	26	6%	26	6%

TABLE 1 – Résultat de synthèse/Virtex6 xc6vlx240t

LUTs minimum est fixé selon le FPGA. Ceci est principalement du principalement à la machine d'état que contient la membrane. Par exemple pour le virtex 6 xc6vlx240t 272 Luts requis est minimum. Le nombre de Luts qu'utilise le virtex 5 est Supérieur au virtex 6 parceque les Luts du Virtex 5 sont des plus petits éléments.

	cache hit	cache miss	cache miss and cache full
Run context	8	11 GMA	12 GMA
Run IP	4 LB	4 LB	4 LB
Run IP + context	10 LB	13 GMA,LB	14 GMA,LB
Stop	3	3	3

TABLE 2 – Cycles par ordre (GMA :Global Memory Access cost, LB :Load Bitstream cost)

Context size	Virtex 5	Virtex 6
2	329 Luts	281 Luts
8	379 Luts	307 Luts
16	440 Luts	343 Luts
64	785 Luts	552 Luts
Linear equation	Y=5,5X + 314	Y=1,09X+272

TABLE 3 – Nombres de Luts de la membrane selon la taille du contexte

7. Conclusion et perspectives

Le principal avantage des FPGAs est la capacité d'exécuter des parties matérielles plus volumineuses avec moins de portes et de veiller à la souplesse d'une solution logicielle tout en conservant l'avantage vitesse d'exécution du matériel. Cette capacité est assurée principalement par les caractéristiques de reconfiguration partielle et dynamique. Dans cet article, on présente une nouvelle approche matérielle pour le contrôle et la gestion de contexte dans les systèmes parallèle et dynamiquement reconfigurable. La méthodologie proposée est distribuée, modulaire et réutilisable avec des membranes flexibles. C'est un modèle inspiré de l'approche par composant bien connu dans le développement de logiciels. Nos membranes permettent de partager, stocker et modifier différents contextes de l'IP, ces sauvegarde se font automatiquement, sans utiliser des services complexes de système d'exploitation. De plus, nos membranes n'utilisent que très peu de superficies par rapport à la place disponible sur les FPGAs récents. Nos travaux en cours traitent désormais de la conception d'un gestionnaire global. Il contiendra une table de nommage globale capable de localiser n'importe quel contexte (dans des caches de membranes ou dans la mémoire globale). Un tel gestionnaire effectue le contrôle de reconfiguration dynamique avec un Microblaze et s'occupe de la communication entre les membranes. En coopération avec des partenaires du projet FAMOUS, les prochaines étapes sont la génération automatique du code des membranes au moyen d'une méthodologie MDE[16] et la conception d'un démonstrateur avec l'implémentation d'un système de vidéo complexe, s'appuyant sur une architecture hautement parallèle.

Bibliographie

1. B. MacDonald (B. H.) et Warren (I.). – Design for dynamic reconfiguration for robot software. *ICARA, Palmerston North, New Zealand*, 2004.

2. Brebner (G.). – A virtual hardware operating system for the xilinx xc6200. *the 6th IWFPLA*, 1996.
3. C.-H. Huang (K.-J. S. e. a.). – Dynamically swappable hardware design in partially reconfigurable systems. *ISCAS*, 2007.
4. D. Stewart (R. V.) et Khosla (P.). – Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transaction on Software Engineering*, 1997.
5. et al (J. C.). – Containment units : a hierarchically composable architecture for adaptive systems. *ACM SIGSOFT Software Engineering Notes*, 2002.
6. Famous. – http ://www.lifl.fr/ meftali/famous.
7. Ferreira (J. C.) et Silva (M. M.). – Runtime reconfiguration support for fpgas with embedded cpus : The hardware layer. *IPDPS*, 2005.
8. FosFor. – http ://users.polytech.unice.fr/ fmuller/fosfor/.
9. Gafni (V.). – Arobots : a real-time systems architectural style. *ESEEC, Toulouse*, 1999.
10. H. El Gindy (M. M. e. a.). – Task rearrangement on partially reconfigurable fpgas with restricted buffer. *fPLA Workshop*, 2000.
11. J. Almeida, M. Wegdam (M. S.) et Nieuwenhuis (L.). – Transparent dynamic reconfigurable for corba. *DOA, Rome, Italy*, 2001.
12. J. Dondo, F. Rincon (e. a.). – Dynamic reconfiguration management based on a distributed object model. *FPL*, 2007.
13. J-Y. Mignolet, V. Nollet (P. C. e. a.). – Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. *Parallel Architectures, Algorithms, and Networks*, 2008.
14. Kim (D.). – An implementation of fuzzy logic controller on the reconfigurable fpga system. *IEEE Transactions on Industrial Electronics*, 2000.
15. M. Simonot (V.). – A declarative formal approach to dynamic reconfiguration. *IWOCE'09*, 2009.
16. Quadri (I.), Boulet (P.), Meftali (S.) et Dekeyser (J.). – Using an mde approach for modeling of interconnection networks. *Parallel Architectures, Algorithms, and Networks, 2008. I-SPAN 2008. International Symposium on*, 2008.
17. S. Guillet, F. Lamotte (N. G. E. R. J.-P. D.) et Gogniat (G.). – Modeling and synthesis of a dynamic and partial reconfiguration controller. *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway*, 2012.
18. Torresen (J.) et Vinger (K.). – High performance computing by context switching reconfigurable logic. *ESMMS*, 2002.
19. Wigley (G.) et Kearney (D.). – The development of an operating system for reconfigurable computing. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.
20. Yamashina (M.) et Motomura (M.). – Reconfigurable computing :its concept and a practical embodiment using newly developed dynamically reconfigurable logic (drl) lsi. *5th ASP-DAC*, 2000.
21. Y.Eustache et J-Ph.Diguet. – Specification and os-based implementation of self-adaptive, hardware software embedded systems. *CODES-ISSS, Atlanta*, 2008.