Service de configuration prédictif pour plateforme multicoeur reconfigurable hétérogène

Mehdi Darouich, Stéphane Guyetant, Stéphane Chevobbe

▶ **To cite this version:**

Mehdi Darouich, Stéphane Guyetant, Stéphane Chevobbe. Service de configuration prédictif pour plateforme multicoeur reconfigurable hétérogène. 2007. hal-00202234

HAL Id: hal-00202234
https://hal.science/hal-00202234v1

Preprint submitted on 4 Jan 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

Service de configuration prédictif pour plateforme multicoeur reconfigurable hétérogène

Mehdi Darouich, Stéphane Guyetant, Stéphane Chevobbe

CEA, LIST Laboratoire Calculs Embarqués
Centre de Saclay ; PC 94 ; F-91191 Gif-Sur-Yvette
{mehdi.darouich,stephane.guyetant,stephane.chevobbe}@cea.fr

Résumé
Cet article décrit un service de gestion des reconfigurations par préchargement prédictif utile pour les architectures multicoeur composées de coeurs reconfigurables hétérogènes. Le but est de masquer les latences de reconfiguration dues aux transferts de bitstreams de grande taille, pour ainsi permettre une plus grande dynamicité de reconfiguration. Nous présentons l'implémentation logicielle du service de préchargement et sa validation fonctionnelle. L'architecture du projet européen Morpheus est utilisée comme exemple pour faire cette validation : nous montrons comment, sur des graphes d'applications simplifiés, masquer complètement le surcoût de la reconfiguration.

Mots-clés : architecture hétérogène reconfigurable dynamiquement, OS embarqué, préchargement prédictif

---

1. Introduction

Les architectures reconfigurables dynamiquement constituent une solution potentielle dans des domaines qui nécessitent flexibilité et bonnes performances de calcul, tels que le multimédia, la sécurité, ou les télécommunications. Associant les avantages d'une implémentation matérielle et reconfigurabilité, les coeurs reconfigurables dynamiquement permettent l'accélération de tâches répétitives et parallélisables. La reconfiguration dynamique permet d'étendre les capacités d'accélération d'une application par virtualisation des ressources matérielles [1]. On peut ainsi accélérer plus de tâches que dans le cas de l'utilisation de la reconfiguration statique. Une plateforme hétérogène intégrant un ou plusieurs coeurs reconfigurables dynamiquement et des processeurs généralistes peut traiter une grande diversité de tâches et ainsi répondre aux demandes d'une application donnée.

Un coeur reconfigurable est généralement vu comme une matrice d'éléments de calcul. La fonction de chaque élément et l'interconnexion qui relient les éléments entre eux constituent les différents niveaux de flexibilité. L'ensemble des données définissant l'état de configuration d'un coeur reconfigurable est stocké au sein d'une matrice d'éléments de mémorisation, appelée aussi plan de configuration. La reconfiguration de ce plan s'effectue par le chargement d'un bitstream de configuration. Plus un coeur est flexible, plus le volume de données définissant son état de configuration est important. La granularité de chemin de donnée influe aussi sur la taille du bitstream. Des coeurs reconfigurables à grain fin nécessiteront ainsi des bitstreams volumineux.

Malgré le potentiel qu'offrent les architectures reconfigurables dynamiquement, il existe très peu d'exemples d'application sur le marché. Ceci est en grande partie dû au fait que l'utilisation de la reconfiguration dynamique introduit plusieurs niveaux de complexité supplémentaires qui doivent être pris en compte pour assurer de bonnes performances :

– La reconfiguration d'une architecture induit le chargement de données de configuration, plus ou moins importantes selon les caractéristiques de l'architecture, pendant l'exécution de l'application. Ce temps de chargement introduit un surcoût temporel rarement négligeable.
– La reconfiguration dynamique induit une complexité matérielle supplémentaire par rapport à la reconfiguration statique. L'introduction de logique de contrôle et de gestion de reconfiguration est nécessaire. La logique de gestion augmente avec la complexité des modèles de reconfiguration. Le stockage des bitstreams de configuration nécessite l'utilisation de mémoires sur puce et/ou de mémoires externes, augmentant ainsi la surface nécessaire. Une hiérarchie mémoire dédiée à la configuration peut aussi être envisagée.
– Le manque de modèle de programmation pour ce type d'architecture est un problème critique pour les concepteurs d'applications. La prise en compte de la reconfiguration matérielle complexifie le flot de conception des applications et la mise en place d'outils dédiés devient une nécessité.

Le surcoût temporel induit par le chargement des données de configuration peut dégrader fortement les performances d'un système reconfigurable. Introduire une latence supplémentaire annule les bénéfices de l'utilisation du principe de reconfiguration dynamique qui n'a plus de légitimité. Nous allons nous intéresser dans la suite de cet article à la réduction de ce coût temporel. Elle est assurée par la mise en place d'une gestion de reconfiguration dynamique. Plusieurs services de gestion de reconfiguration permettent d'en gérer les différents aspects. Nous proposons une implémentation logicielle d'un gestionnaire de configuration prédictif qui intègre un service de *prefetching* de configuration visant la réduction du surcoût temporel par préchargement des données de configuration, associé à des services de *caching* de configuration et d'allocation de ressource. Une cosimulation nous a permis d'effectuer une validation fonctionnelle de ce gestionnaire.

Un état de l'art des services de gestion de reconfiguration est exposé dans la partie suivante. La troisième partie présente le support d'exécution choisi pour cette étude. Le service de gestion de reconfiguration que nous proposons est présenté en quatrième partie. Les résultats de la validation fonctionnelle sont explicités dans une dernière partie.

2. Travaux similaires

Il existe de nombreux exemples de services de gestion de reconfiguration dans la littérature. La figure 1 regroupe les services les plus couramment rencontrés. Ces services peuvent être classés selon trois catégories. Les services appartenant à la catégorie *optimisation temporelle* visent à réduire le temps nécessaire à la reconfiguration par réduction ou masquage du temps de chargement des données de configuration. Les services d'*optimisation spatiale* ont pour objectif de maximiser le nombre de configurations présentes sur la ressource matérielle reconfigurable. Ainsi, plus de tâches peuvent être exécutées simultanément sur un plan de configuration donné. Les services d'*optimisation fonctionnelle* permettent d'augmenter la flexibilité des architectures.

Le comportement dynamique d'un service est assuré par l'échange de données avec l'ordonnanceur. Celui-ci met à la disposition du service des informations concernant l'état d'exécution de l'application en cours. Le service peut ainsi adapter son traitement à l'évolution de l'application et ainsi répondre au mieux à ses demandes. L'association de plusieurs de ces services permet d'obtenir de meilleures performances. Ces services peuvent collaborer de plusieurs manières, comme l'illustre la figure 1. Les services de *prefetching* de configuration et de *caching* de configuration, par exemple, mènent des actions complémentaires pour réduire le temps de reconfiguration. Les services de partitionnement, placement et routage sont souvent associés au sein d'une chaîne de traitement pour permettre l'optimisation spatiale de la reconfiguration.

Il n'est cependant pas envisageable d'intégrer l'ensemble de ces services au sein d'un même gestionnaire de reconfiguration. En effet, chaque service, du fait du traitement qu'il nécessite, introduit une complexité calculatoire supplémentaire. Nous nous intéressons par la suite à trois services essentiels : le *prefetching* de configuration, le *caching* de configuration et l'allocation.

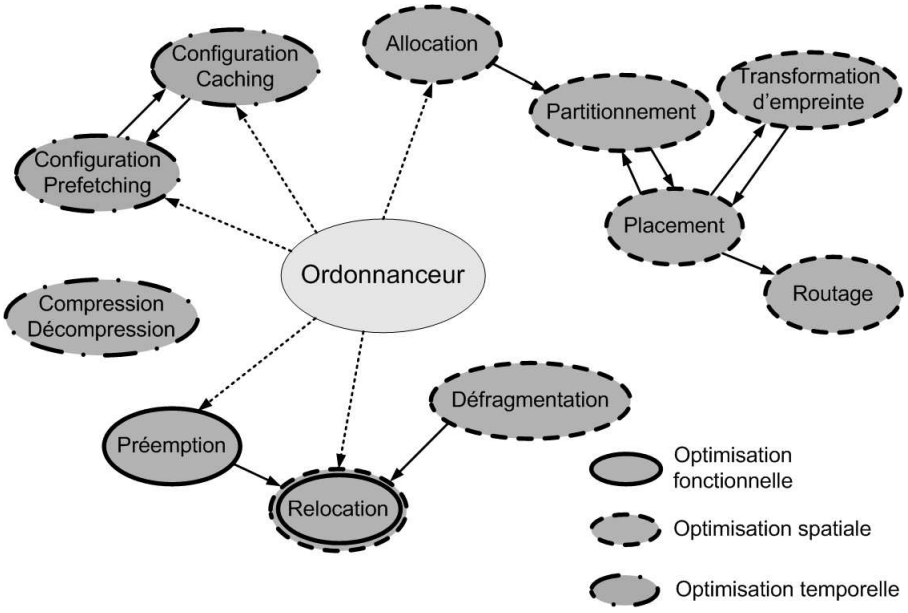


FIG. 1 – Services de gestion de reconfiguration existants, classés selon les trois types d'optimisations qu'ils permettent. Certains peuvent collaborer directement avec l'ordonnanceur.

2.1. *Prefetching* de configuration

Le service de *prefetching* de configuration repose sur le même principe que le *prefetch* dans les processeurs généralistes (GPP). Les données de configuration sont chargées sur la ressource matérielle avant que celle-ci en ait besoin. On peut ainsi masquer tout ou partie du temps de reconfiguration. La figure 2 illustre ce principe avec trois coeurs reconfigurables (IP). Les blocs *Ln* correspondent au chargement d'une tâche, les blocs *Ex_n* correspondent à son exécution. Dans le cas où aucun *prefetching* n'est effectué (figure 2.a), la reconfiguration entraîne un surcoût temporel. Lorsque le service de *prefetching* est appliqué (figure 2.b) le temps de chargement est réduit, voire masqué dans le cas idéal.

Un exemple de service de *prefetching* de configuration a été développé dans [2]. Il cible des plateformes multiprocesseur hétérogènes intégrées dans un FPGA. L'application à exécuter est découpée en tâches qui sont elles-mêmes subdivisées en sous-tâches. Le service a pour but d'ordonnancer le chargement des sous-tâches de manière à masquer le surcoût temporel. Il utilise pour cela des données statiques et des données dynamiques. La phase de génération des données statiques s'effectue à la conception de l'application. Un ensemble de sous-tâches dites critiques est extrait : elles correspondent aux sous-tâches qui devront être chargées en priorité à l'exécution pour assurer le masquage du temps de chargement. La phase dynamique consiste à déterminer les sous-tâches critiques à charger en fonction de l'état de l'exécution de l'application. Cette information est fournie par l'ordonnanceur. La génération d'une partie des données hors-ligne permet de réduire le volume de calcul à effectuer lors de l'exécution tout en conservant un comportement dynamique. Cette approche est intéressante dans le cas de services devant s'adapter à l'état d'exécution de l'application et mettant en jeu de nombreux calculs. Un autre exemple de service de *prefetching* utilisant à la fois des données statiques et dynamiques est présenté dans [3], mais en ne se concentrant aussi que sur les techniques propres aux circuits FPGA.

2.2. *Caching* de configuration

Le service de *caching* de configuration est basé sur le même fonctionnement que le *caching* des GPPs [4]. Il a pour objectif de déterminer les données de configuration qui doivent rester dans le plan de configuration. On diminue ainsi le surcoût de chargement des bitstreams sur le coeur reconfigurable et le nombre de mémoires sur la puce. Cependant, l'approche de *caching* utilisée sur les GPPs ne convient

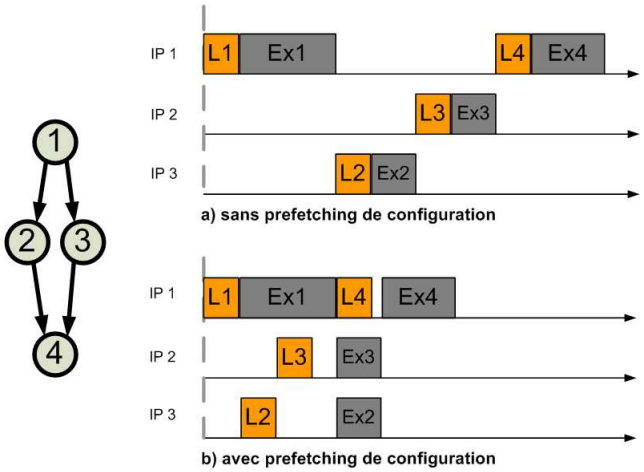


FIG. 2 – Principe de *prefetching* de configuration, pour quatre tâches qui s'exécutent sur trois coeurs différents, notés IP 1 à 3. Dans (b) le *prefetch* permet un temps d'exécution total moins long pour l'application.

pas au cas des architectures reconfigurables à cause de la non-uniformité de la taille des bitstreams de configuration. Dans une approche GPP, on gardera les données les plus souvent accédées. Dans une approche architecture reconfigurable, la taille du bitstream et le temps de chargement entrent en compte : il sera parfois plus judicieux de conserver un bitstream volumineux plutôt qu'un bitstream plus petit et plus fréquemment utilisé.

Dans [4], plusieurs algorithmes de *caching* de configuration sont développés, ciblant plusieurs modèles de FPGAs. On peut notamment citer un algorithme de *caching* pour FPGA *multi-context* et des algorithmes pour différents types de FPGAs partiellement reconfigurables. Ces algorithmes ont été implémentés en C++ pour être testés. Il ressort notamment de ces tests que, dans le cas de plans de configuration de petite taille, le modèle *multi-context* présente un surcoût de reconfiguration 20% à 40% moins important que pour le modèle partiellement reconfigurable.

2.3. Allocation

Le service d'allocation de ressources a pour rôle d'attribuer des ressources à une tâche. Il maintient à jour une liste des ressources libres, qu'il consulte au moment de l'allocation. Dans le cas où la fragmentation du plan de configuration est pénalisante, le service d'allocation peut être amené à déterminer la contigüité des blocs allouables. L'article [5] associe le service d'allocation à un service de partitionnement, un service de placement et un service de routage. Ces quatre services sont présentés comme les services fondamentaux que doit intégrer un OS pour une architecture reconfigurable. Les algorithmes associés à ces opérations pouvant être très complexes, un compromis est fait entre leur efficacité et leur temps d'exécution.

2.4. Synthèse

Au terme de cette étude, il nous est apparu que le *prefetching* de configuration est un des services essentiels à la réduction du coût temporel de la reconfiguration. Pour une meilleure efficacité, notre service utilise des informations générées hors-ligne après analyse des dépendances de l'application, et des données provenant de l'ordonnanceur au cours de l'exécution. Les bitstreams de configuration sont amenés au coeur reconfigurable et nous ne considérons pas l'aspect intrinsèque de sa reconfiguration. Cette approche au niveau système nous permet d'être générique et d'adresser des systèmes possèdant différents coeurs reconfigurables. Ce service de *prefetching* de configuration est intégré au sein d'un gestionnaire de configuration prédictif appelé Predictive Configuration Manager (PCM). Un service de *caching* de configuration et un service d'allocation de ressources complètent l'action du service de *prefetching* de configuration.

3. Cas d'étude

La plateforme developpée au sein du projet MORPHEUS[1] [6] a été prise comme cas d'étude. Cette plateforme, dont la structure est représentée figure 3, possède trois coeurs reconfigurables de grains de calcul hétérogènes : XPP-3[7] – gros grain partiellement reconfigurable, DREAM[8] – grain moyen partiellement reconfigurable, FLEXEOS[9][10] – grain fin à contexte unique. L'échange de données s'effectue soit à travers un *Network on Chip* (NoC) à forte bande passante, soit à travers le bus de donnée. Un Système d'exploitation (OS) implémenté sur l'ARM9 permet d'effectuer le contrôle haut niveau de l'application, le processeur pouvant également servir à du traitement applicatif. Un bus de configuration AHB et quatre niveaux de hiérarchie mémoire sont prévus pour le stockage des données de configuration. Le premier niveau de hiérarchie est constitué des *mémoires IP* qui sont les plans de configuration des coeurs reconfigurables. Au niveau *mémoire cache*, une mémoire double port est dédiée à chaque coeur. Le troisième niveau est constitué par la mémoire locale de configuration. Elle est placée sur le bus de configuration et est partagée par les trois coeurs reconfigurables. Les données de configuration sont initialement stockées dans une *mémoire externe* placée sur le bus de configuration et partagée par les trois coeurs.

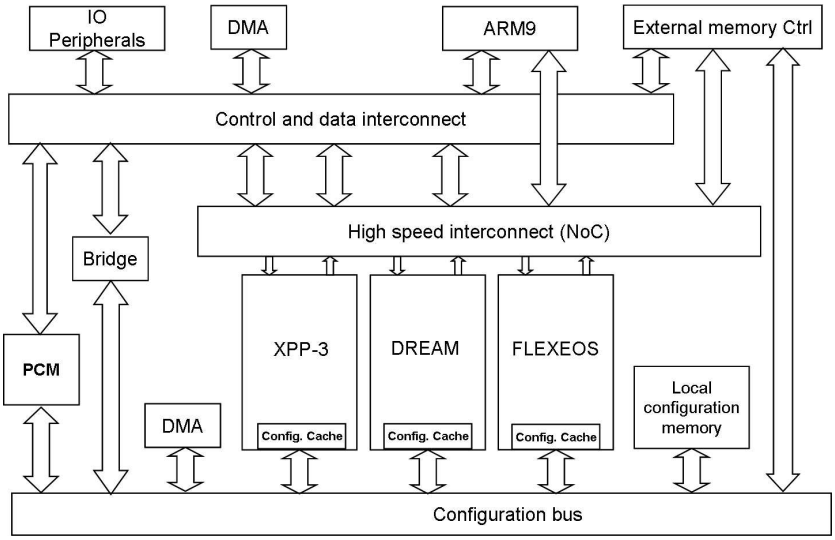


FIG. 3 – La plateforme MORPHEUS est composée de trois coeurs reconfigurables XPP-3, DREAM et FLEXEOS interconnectés par un NoC. Les configurations sont gérées par le composant matériel dédié PCM.

Le PCM gère les transferts mémoire des données de configuration. Dans le cadre du projet MORPHEUS, l'implémentation du PCM sous la forme d'un composant matériel dédié a été considérée comme la solution pour répondre au compromis entre efficacité et temps de calcul. Les ressources de l'ARM9 sont disponibles pour les autres traitements, et notamment les traitements applicatifs, et l'accélération matérielle permet une amélioration des performances du PCM. Dans un premier temps, une implémentation logicielle du PCM a été mise en place afin d'effectuer sa validation fonctionnelle. Implémenté sous la forme d'un pilote de périphérique intégré à l'OS embarqué eCos [11] s'exécutant sur l'ARM9, le PCM reçoit en entrée les instructions de l'ordonnanceur gérant l'application et programme le DMA à travers le bus de configuration.

[1] Projet européen IST 027343
"Multi-purpose dynamically Reconfigurable Platform for intensive Heterogenous processing"

L'un des rôles de la cosimulation décrite dans cet article a été le dimensionnement relatif – à surface de silicium fixée – des mémoires caches et de la mémoire locale. A mesure que l'on descend dans la hiérarchie mémoire, les transferts sont optimisés. Les mémoires caches permettent un chargement rapide mais manquent de flexibilité car elles sont dédiées à un coeur. La mémoire locale est, au contraire, partagée par tous les coeurs. Cependant, les accès à cette mémoire sont plus lents car ils sont limités par la bande passante du bus de configuration et ils ne peuvent être effectués en parallèle.

4. Gestionnaire de configuration prédictif

Le PCM a pour rôle de gérer de manière optimale la hiérarchie mémoire de configuration. Son objectif principal est d'alimenter les mémoires de configuration des coeurs reconfigurables afin de charger les tâches nécessaires à l'application dont l'exécution est gérée au sein de l'ARM9 par un OS. Le PCM utilise des données statiques et des données dynamiques pour effectuer son traitement (voir figure 4).

Des graphes de dépendances de configuration sont générés à la compilation de l'application à partir des Control-Data Flow Graph (CDFG) (flot de compilation Molen[12]) et d'informations spécifiques fournies par le concepteur de l'application. L'utilisation du PCM est facilitée par la mise à disposition d'un ensemble de paramètres ajustables au regard des spécificités de l'application. Plusieurs implémentations différentes correspondant à autant de bitstreams différents peuvent être spécifiées pour chaque tâche, ainsi qu'un niveau de priorité entre chacune indiquant laquelle sera utilisée de préférence.

L'OS envoie au PCM, via l'ordonnanceur, des commandes le renseignant sur l'évolution de l'exécution de l'application et des données spécifiques aux choix d'ordonnancement retenus comme les priorités de thread. Les trois instructions envoyées par l'ordonnanceur au PCM sont :
– SET : la tâche doit être chargée immédiatement sur le coeur considéré
– EXEC : l'exécution de la tâche est lancée. C'est pendant ce temps que les préchargements peuvent être effectués
– RELEASE : marque la fin de l'exécution de la tâche, la ressource occupée est alors libérée

Dans cet ordonnancement statique de chaque thread, les commandes SET sont positionnées le plus tard possible pour éviter de polluer les mémoires de configuration avec des bitstreams inutiles. Cette commande sert essentiellement à rattraper une mauvaise prédiction du PCM. Au pire cas d'exécution, les chargements des bitstreams s'effectueront donc sur ces commandes SET.

Le PCM intègre trois services de gestion de reconfiguration : un service de *prefetching* de configuration qui constitue le service principal, et des services de *caching* de configuration et d'allocation de ressources qui en complètent l'action. Ces trois services sont décrits ci-dessous.

4.1. *Prefetching* de configuration

Le service de *prefetching* de configuration détermine les préchargements à effectuer pour répondre aux demandes de l'application en terme de ressources matérielles. Cette opération s'effectue en plusieurs étapes.

Tout d'abord, à partir de l'état donné de l'exécution et de la connaissance des dépendances de contrôle, l'ensemble des tâches éligibles pour un préchargement est déterminé. Chaque bitstream de configuration est constitué d'un ensemble de blocs mémoire. L'ordre de préchargement des tâches et la proportion de blocs à rapatrier pour chaque tâche sont déterminés selon plusieurs paramètres comme le niveau de priorité de la tâche, le mode de reconfiguration du coeur visé et l'état d'occupation des différents niveaux de la hiérarchie mémoire. L'objectif étant de minimiser le temps de chargement des tâches sur les coeurs reconfigurables, les données de configuration doivent être rapprochées le plus possible des mémoires de configuration, en fonction de leur probabilité d'exécution. Le service de *prefetching* de configuration détermine la mémoire cible et le nombre de blocs à rapatrier pour répondre à ce critère.

A l'issue de cette opération, le service de *prefetching* de configuration parcourt la liste des tâches classées

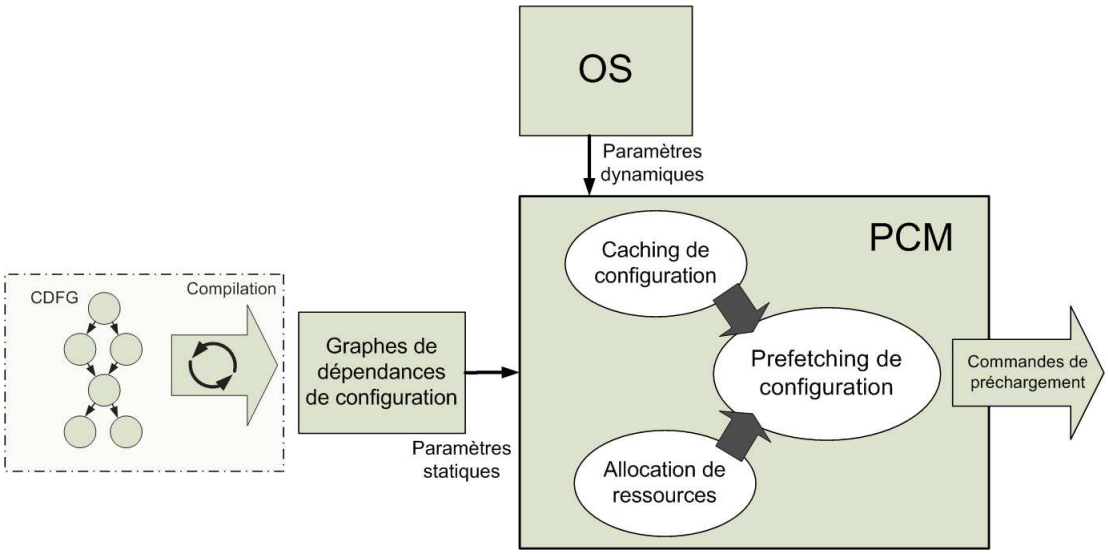


FIG. 4 – Principe de fonctionnement du PCM : Les graphes de dépendances de configuration sont issus d'une analyse statistique de l'application. D'autre part, les appels systèmes sont envoyés par l'OS au PCM au cours de l'exécution de l'application. Le PCM se base sur ces deux types d'information pour le calcul du nombre de blocs à rapatrier et le niveau de hiérarchie mémoire ciblé.

par ordre de priorité et, pour chacune, programme des transferts DMA pour précharger les blocs du bitstream de configuration de l'emplacement mémoire source à l'emplacement mémoire destination précédemment déterminés. Ces préchargements sont effectués lorsque le bus de configuration est libre. Les tâches devant être exécutées dans l'immédiat étant chargées en priorité, les préchargements sont effectués pendant les phases d'exécution de tâche sur les coeurs reconfigurables.

4.2. *Caching* de configuration

Le service de *caching* de configuration gère l'espace mémoire et détermine les blocs de configuration à conserver aux différents niveaux de la hiérarchie mémoire. Un âge est attribué à chaque tâche présente dans la hiérarchie mémoire. Cet âge évolue selon plusieurs critères et notamment les décisions prises par l'ordonnanceur et la taille des données de configuration. Le service de *caching* de configuration vérifie périodiquement l'âge des tâches. Lorsque l'une d'elle a dépassé un certain âge limite, il libère l'espace qu'elle occupait en mémoire.

4.3. Allocation de ressources

Plusieurs versions d'une même tâche peuvent exister. Ces versions diffèrent soit par le coeur ciblé, soit par l'implémentation pour un même coeur. Le service d'allocation détermine l'implémentation la plus adaptée selon les ressources disponibles et les besoins de l'application. On peut par exemple choisir de charger une tâche dans une implémentation nécessitant peu de ressources mais moins rapide dans le cas d'une application nécessitant l'exécution de beaucoup de tâches en parallèle.

5. Expérimentations

Le comportement du module logiciel du PCM a été testé et validé en exécutant plusieurs pseudo-applications. Chacune est définie par un ordonnancement statique exécuté au sein de l'OS ainsi que des graphes décrivant les dépendances de contrôle, de configuration et de données. La figure 5 présente un exemple de graphes d'une pseudo-application découpée en trois sous-graphes indépendants. Il est à noter que, dû à une limitation du simulateur, les graphes implémentés sont de petite taille et n'excèdent pas la vingtaine de noeuds. Cependant les graphes choisis ont permis de tester un ensemble varié de

motifs de dépendances suffisant pour valider le PCM.

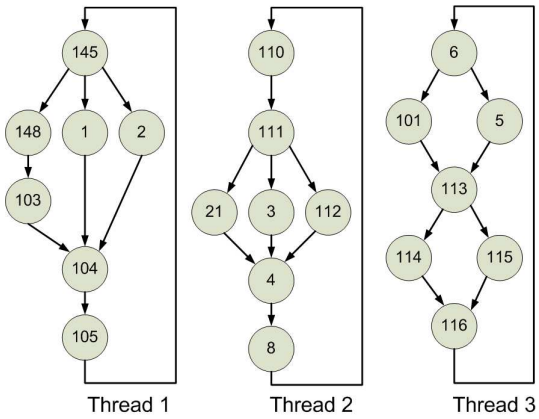


FIG. 5 – Graphes de dépendance de configuration pour une application à trois threads d'exécution

5.1. Critère d'évaluation

Le surcoût temporel du chargement d'une configuration est masqué lorsque celle-ci est présente en mémoire cache au moment où elle est nécessaire − c'est à dire lors de l'exécution de l'instruction SET lui correspondant. Le critère d'évaluation du module PCM se base donc sur la mesure du nombre de configurations présentes en mémoire cache lorsque le SET lui correspondant est exécuté. Ce critère est appelé << taux de match >> . Il est facilement calculé en faisant le rapport entre le nombre total de SET correspondant à un bitstream en mémoire cache sur le nombre total de SET au cours d'une exécution.

5.2. Environnement de simulation

Une plateforme de simulation a été mise en place pour permettre la validation fonctionnelle de l'implémentation logicielle du PCM. Cette plateforme est basée sur un modèle systemC de la plateforme MORPHEUS et une << cible synthétique >> (voir figure 6). La cible synthétique est une solution d'émulation sur PC muni d'une distribution Linux permettant d'exécuter l'application à tester en tant que processus Linux. L'utilisation d'une cible synthétique nous permet d'effectuer une première estimation des performances du PCM avec des temps de simulation et de prise en main relativement courts. Ce type de cosimulation a été utilisé dans [13], la communication entre les deux simulations étant effectuée par l'intermédiaire de *sockets*. Notre approche se base sur l'utilisation de segments de mémoire partagés pour la communication, plus simples à mettre en place.

Le temps de chargement d'un bloc mémoire effectué par le DMA doit être au moins égal à 1 cycle d'exécution de la plateforme de cosimulation − cycle correspondant à un tick d'exécution de la cible synthétique. Nous avons arbitrairement fixé ce temps de chargement à 1 cycle. Après la réception d'une instruction EXEC, un délai est établi pour émuler l'exécution de la tâche correspondante. Ce délai est réglable et correspond à un nombre de cycle d'exécution de la plateforme de cosimulation. C'est pendant ce temps d'attente que le PCM peut effectuer les préchargements de blocs mémoire par transfert DMA. Ce délai fixe le nombre de blocs de configuration qui peuvent être préchargés pendant l'exécution d'une tâche. D'autres paramètres tels que la taille des mémoires caches et des bitstreams sont modifiables. La taille des mémoires caches fait l'objet d'une discussion dans la partie résultat. La taille des bitstreams de configuration a été fixée à 10 blocs de 2Koctets. Ceci correspond à un cas moyen dans la plateforme MORPHEUS.

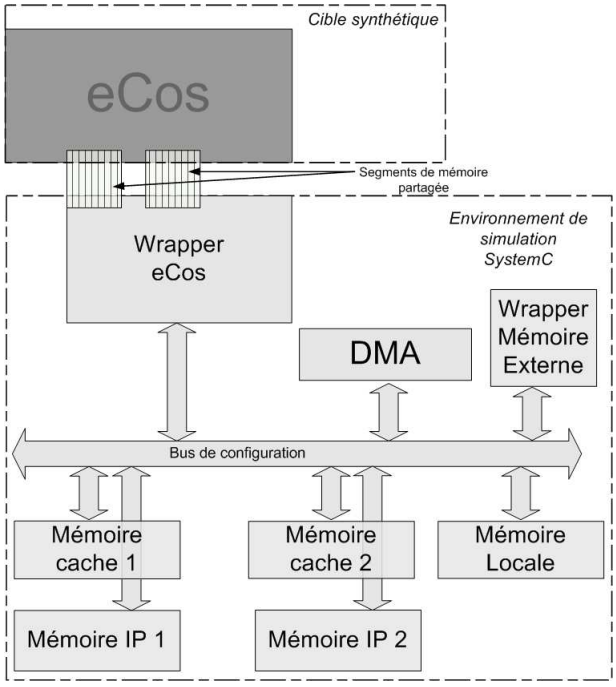


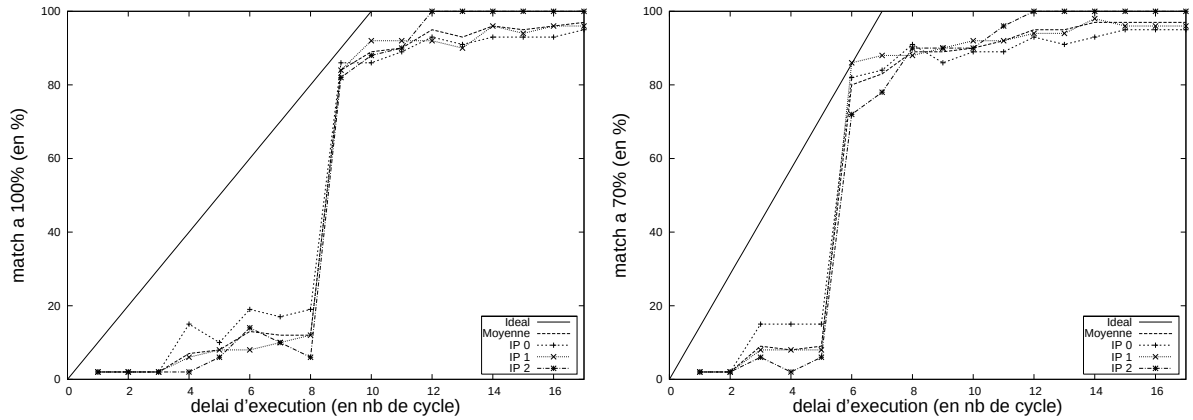
FIG. 6 – Plateforme de cosimulation associant une cible synthétique et une simulation systemC

5.3. Résultats

La figure 7 présente les résultats de la simulation de l'application figure 5, pour des mémoires caches de taille 20 blocs de 2Koctets chacune. On fait varier le temps d'exécution pour observer son impact sur le taux de match. Les deux figures 7(a) et 7(b) présentent le taux de tâches faisant match respectivement à 100% et à 70%. Ces deux graphes ont une forme similaire, avec la présence de deux palliers, l'un situé en dessous de 20% et l'autre situé au dessus de 80%. La seule différence notable est la position du saut entre les deux palliers. Ce saut correspond au délai permettant au module PCM d'atteindre de bons résultats. Dans le cas du match à 70%, le délai est plus court et près de 80% des tâches font match à 70% au bout de six cycles. Il est à noter que le taux de tâches faisant match à 70% est un bon indicateur. Une tolérance de 30% – soit trois blocs ici – à charger de la mémoire externe peut être admise. Les mémoires caches étant des mémoires *dualport*, une partie des blocs encore en mémoire externe peut être rapatriée de celle-ci à la mémoire cache à travers le bus de configuration pendant que les blocs déjà présents en mémoire cache sont transférés vers le plan de configuration.

Les courbes nommées << Idéal >> dans la figure 7 représentent le meilleur taux de match atteignable pour un délai donné. Elle est définie par le ratio entre le temps d'exécution d'une tâche et le temps nécessaire au chargement d'un bitstream de configuration. La comparaison entre les courbes de simulation du module PCM et les courbes idéales nous permet d'évaluer les résultats obtenus. Dans la figure 7(b) nous pouvons voir que, pour la politique choisie, la courbe moyenne se rapproche de la courbe idéale à partir de 12 cycles. Cette politique n'étant pas optimisée pour l'application donnée, d'autres paramètres pourront être trouvés pour se rapprocher de la courbe idéale.

La figure 8 présente les résultats obtenus pour la simulation de l'application figure 5, pour des tailles de mémoire cache de 10, 20, 30 et 40 blocs. Il apparaît que, pour un délai d'exécution compris entre 7 et 15 cycles, des mémoires caches de taille 20 blocs permettent d'obtenir les mêmes résultats que des mémoires de taille supérieure. A partir de 16 cycles, les mémoires de 30 et 40 blocs permettent d'atteindre un taux de match à 70% de près de 100%. On peut noter cependant qu'augmenter la taille des



(a) Taux de tâches disponibles à 100%

(b) Taux de tâches disponibles à 70%

FIG. 7 – Nombre de tâches dont les blocs sont présents à 100% et 70% en mémoire cache. La mémoire cache de chaque coeur reconfigurable (IP0, IP1 et IP2) peut contenir 20 blocs de 2Koctets. Chaque bitstream de configuration est composé de 10 blocs de 2Koctets.

mémoires caches au delà de 30 blocs n'améliore pas les résultats. Ces résultats permettent une première estimation, pour une application donnée, de la taille des mémoires caches nécessaires pour obtenir un bon taux de match en un délai limité.

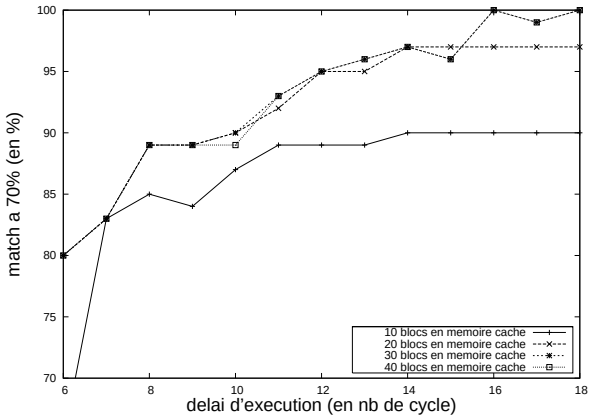


FIG. 8 – Evolution du taux de match à 70% en fonction de la taille des mémoires caches

6. Conclusion

Le surcoût temporel de la reconfiguration dynamique peut dégrader fortement les performances d'un système. La mise en place de services de gestion de reconfiguration permet de pallier à ce problème. Nous avons mis en place un gestionnaire de reconfiguration prédictif − PCM − se basant sur l'utilisation de données générées à la compilation de l'application et de données générées pendant l'exécution. Cette approche permet de limiter le nombre de calculs à effectuer pendant l'exécution du service. Le

PCM intègre des services de *prefetching* de configuration, de *caching* de configuration et d'allocation de ressources. L'étude présentée dans cet article est une preuve de concept du module PCM. Une validation fonctionnelle du PCM a été effectuée sous forme d'un service d'OS temps réel en utilisant des graphes de dépendances de configuration simplifiés. Pour les applications réelles qui seront exécutées sur la plateforme MORPHEUS, le calcul du service de *prefetch* se fera dans un composant matériel dédié, libérant ainsi la charge du processeur ARM pour du calcul applicatif.

Bibliographie

1. Andre DeHon and John Wawrzynek. Reconfigurable computing : what, why, and implications for design automation. In *DAC '99 : Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 610–615, New York, NY, USA, 1999. ACM Press.
2. Javier Resano, Daniel Mozos, and Francky Catthoor. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In *DATE '05 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 106–111, Washington, DC, USA, 2005. IEEE Computer Society.
3. Zhiyuan Li and Scott Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *FPGA '02 : Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 187–195, New York, NY, USA, 2002. ACM Press.
4. Zhiyuan Li, Katherine Compton, and Scott Hauck. Configuration caching management techniques for reconfigurable computing. In *FCCM '00 : Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 22, Washington, DC, USA, 2000. IEEE Computer Society.
5. Grant Wigley and David Kearney. The development of an operating system for reconfigurable computing. In *FCCM '01 : Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–250, Washington, DC, USA, 2001. IEEE Computer Society.
6. MORPHEUS. www.morpheus-ist.org/.
7. PACT. The XPP III white paper.
8. Andrea Lodi, Mario Toma, and Fabio Campi. A pipelined configurable gate array for embedded processors. In *FPGA '03 : Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2003. ACM Press.
9. M. Borgatti, F. Lertora, B. Foret, and L. Cali. A reconfigurable system featuring dynamically extensible embedded microprocessor, fpga, and customizable I/O,. *IEEE Journal of Solid-State Circuits*, vol.38, no.3 :pages 521–529, 2003.
10. M2000. http ://www.m2000.com.
11. eCos. http ://ecos.sourceware.org/.
12. E. Moscu Panainte, K.L.M. Bertels, and S. Vassiliadis. The molen compiler for reconfigurable processors. *ACM Transactions in Embedded Computing Systems (TECS)*, February 2007.
13. Franco Fummi, Stefano Martini, Giovanni Perbellini, and Massimo Poncino. Native ISS-SystemC integration for the co-simulation of multi-processor SoC. In *DATE '04 : Proceedings of the conference on Design, automation and test in Europe*, page 10564, Washington, DC, USA, 2004. IEEE Computer Society.