Patrons de conception pour l'analyse d'ordonnancement temps réel multiprocesseurs

Stéphane Rubini*, Frank Singhoff*, Alain Plantec*, Hai Nam Tran†, Jalil Boukhobza*, and Pierre Dissaux‡
*Lab-STICC, CNRS, UMR 6285, Univ. Bretagne Occidentale, 29200 Brest, France
†INRIA, Campus de Beaulieu, 35042 Rennes Cedex, Rennes
‡Ellidiss Technologies, 24 quai de la douane, 29200 Brest, France

***Résumé*—Afin de faciliter la vérification *a priori* de systèmes temps réel monoprocesseurs, *Cheddar* dispose d'un service permettant, pour une architecture donnée, de garantir l'applicabilité d'une méthode d'analyse d'ordonnancement. Ce service de vérification exploite un catalogue de patrons de conception.**

Dans cet article, nous proposons d'étendre ces patrons aux architectures multiprocesseurs. Nous proposons un modèle de la plateforme d'exécution, puis, nous l'appliquons afin de modéliser différentes architectures pour lesquelles *Cheddar* propose des méthodes d'analyse de l'ordonnancement. Deux exemples d'analyse illustrent ces propositions.

I. Introduction

Un système embarqué se doit non seulement d'être fonctionnellement correct, mais aussi de respecter des propriétés non fonctionnelles telles que les contraintes de temps. On parle alors de systèmes temps réel [1]. Lorsque le respect de ces contraintes de temps doit absolument être garantie, on parle de systèmes temps réel critiques.

L'implantation de tels systèmes exploite de plus en plus fréquemment plusieurs unités de calcul. Par exemple, un avion ou une voiture, peut mettre en œuvre plusieurs calculateurs inter-connectés [2].

L'analyse d'ordonnancement a pour objectif de vérifier *a priori* le respect des contraintes de temps, et pour ce faire, de définir le partage des ressources de calcul. De nombreux modèles et méthodes d'analyse d'ordonnancement ont été proposés par la communauté académique. Cependant, du fait de la diversité des environnements d'exécution, de l'évolution rapide de leurs caractéristiques physiques et de la multitude des classes de problèmes à traiter, l'analyse d'ordonnancement demeure un problème difficile.

Le projet *Cheddar* [3] vise à faciliter l'application des méthodes d'analyse d'ordonnancement dans les pratiques d'ingénierie des systèmes temps réel critiques. Dans le cadre du projet *Cheddar*, l'outil de meme nom, *Cheddar*, est conçu pour aider les ingénieurs à appliquer ces analyses. Pour cela, il implante plusieurs algorithmes d'ordonnancement, des tests de faisabilité, un moteur de simulation et divers outils permettant d'effectuer de l'exploration d'architectures logicielles.

L'outil *Cheddar* exploite notamment un catalogue de patrons de conception et intègre un service permettant de vérifier la conformité d'un modèle d'architecture à ces patrons [4]. Un patron de conception se constitue d'un ensemble de contraintes sur l'architecture logicielle garantissant que les hypothèses d'applicabilité d'une méthode d'analyse d'ordonnancement soient respectées. L'utilisation d'un patron avec *Cheddar* permet donc au concepteur de s'assurer de l'adéquation d'une méthode d'analyse d'ordonnancement pour l'architecture à vérifier.

Initialement établi pour des environnements d'exécution monoprocesseur, les capacités de *Cheddar* sont actuellement étendues pour la prise en compte des systèmes multiprocesseurs, et en particulier, ceux intégrés sur une puce de silicium unique. L'analyse d'ordonnancement de tels systèmes constitue un défi majeur pour permettre leur usage futur dans des systèmes temps réel critiques.

Contribution : Cet article présente un ensemble de patrons de conception dont l'objectif est de guider la mise en œuvre de l'analyse d'ordonnancement multiprocesseurs. Ils discriminent les systèmes selon le déploiement des tâches sur les unités de calcul disponibles, mais aussi selon les ressources matérielles complémentaires que ces unités de calcul utilisent ou partagent.

Plan : Dans la partie suivante, nous rappelons les principaux patrons de conception monoprocesseurs initialement proposés pour *Cheddar*. Dans la partie 3, nous montrons comment étendre ces patrons aux environnements d'exécution multiprocesseurs. La partie 4 illustre cette proposition par des exemples expérimentés avec *Cheddar*. Enfin, la partie 5 décrit les travaux connexes avant de conclure dans la partie 6.

II. Patrons de conception

Dans cette partie, nous définissons le concept de patron de conception utilisé par l'outil *Cheddar*. Puis, nous donnons quelques exemples de patrons actuellement utilisés par l'outil.

A. Concept de patron de conception adapté à l'analyse d'ordonnancement

Chaque méthode d'analyse d'ordonnancement requiert que le modèle d'architecture à analyser soit conforme à un ensemble d'hypothèses que l'on nomme contraintes d'applicabilité.

Plus le nombre de contraintes d'applicabilité ou le nombre de méthodes d'analyse d'ordonnancement utilisables est grand, plus il est difficile, pour un concepteur, de choisir la méthode d'analyse à appliquer. Le choix devient d'autant plus complexe que chaque méthode d'analyse peut avoir des caractéristiques différentes en terme de précision du résultat calculé ou du passage à l'échelle [5].

Les patrons de conception peuvent aider au choix de la méthode d'analyse. En effet, un patron de conception spécifie les contraintes d'applicabilité d'une méthode d'analyse. Pour le concepteur, le problème revient alors à s'assurer que le modèle d'architecture qu'il souhaite vérifier est bien conforme à un patron de conception. Notons que *Cheddar* [4] automatise la vérification de conformité d'un modèle d'architecture à un patron de conception.

B. Formalisation des patrons de conception pour l'analyse d'ordonnancement

Env1	L'environnement d'exécution comporte une seule unité de calcul.
Env2	La politique d'ordonnancement peut être soit : *EDF*, *LLF* ou à priorités fixes.
Env3	Le niveau de préemptivité de l'ordonnanceur doit être explicite.
Env4	L'ordonnanceur ne doit pas utiliser de quantum.
Env5	Il n'y a pas d'ordonnancement hiérarchique.

TABLE I
CE TABLEAU (EXTRAIT DE [6]) DÉCRIT UN ENVIRONNEMENT D'EXÉCUTION MONOPROCESSEUR. LA COLONNE DE GAUCHE COMPORTE L'IDENTIFIANT DE LA CONTRAINTE. LA COLONNE DE DROITE DÉCRIT LA CONTRAINTE EN LANGUE NATURELLE.

Dans sa thèse [4], Vincent Gaudel formalise le concept de patron de conception utilisé dans *Cheddar*. Chaque patron de conception est défini par le triplet suivant :

1) **Un ensemble de contraintes d'environnement.** Cet ensemble de contraintes caractérise l'environnement d'exécution. Il décrit la partie matérielle du modèle d'architecture ainsi que les logiciels permettant l'exploitation de ces ressources (système d'exploitation, pilotes de périphérique, ...). Dans sa thèse, Gaudel focalise son attention sur un environnement monoprocesseur caractérisé par les contraintes d'applicabilité de la table I.
2) **Un ensemble de contraintes de communication et de synchronisation.**

   Ce deuxième ensemble de contraintes caractérise la partie logicielle du modèle d'architecture à analyser. Ici, nous nous concentrons sur la concurrence, et en particulier sur l'ensemble des tâches constituant l'application ainsi que leur communication et synchronisation.

   Les modèles utilisés pour l'analyse d'ordonnancement temps réel se fondent sur la notion de tâches. Une tâche est caractérisée par un ensemble de paramètres temporels [7]. Par exemple, le modèle de tâche périodique suppose qu'une tâche est invoquée plusieurs fois pour effectuer le même traitement successivement. Chacune de ces invocations est appelée un *job*. Un délai fixe, appelé période, sépare chaque activation de la tâche périodique. Parmi ces paramètres temporels, la plupart est issu de l'analyse et de la conception du système et ne dépend pas de l'environnement d'exécution. Par exemple, l'échéance spécifie l'instant au delà duquel l'exécution d'un *job* risque de perturber le comportement attendu du système. D'autres, et spécifiquement le temps maximum nécessaire pour exécuter un *job* (ou Worst Case Execution Time) d'une tâche, sont liés à l'environnement d'exécution.

   Cinq ensembles de contraintes de communications et de synchronisation ont été définis par [8], [5], puis formalisés par [4]. Ces ensembles sont décrits dans la suite de cette partie.
3) **Un ensemble de méthodes d'analyse.** Chaque patron explicite une association entre des méthodes d'analyse d'ordonnancement et un type d'architecture dont les caractéristiques sont conformes aux contraintes d'applicabilité des méthodes d'analyse. Les méthodes d'analyse peuvent être des tests sur le taux d'occupation du processeur, des simulations sur l'intervalle de faisabilité, des calculs de pire temps de réponse, ...

La figure 1 explicite les relations entre des patrons de conception, des méthodes d'analyse et des architectures à vérifier. On note qu'une architecture donnée peut être conforme à aucun patron ou qu'aucune méthode d'analyse ne soit applicable. On note encore qu'une méthode d'analyse peut être applicable pour une architecture sans qu'aucun patron de conception ne soit utilisable pour garantir sont applicabilité.

Dans la partie suivante, nous rappelons les principaux ensembles de contraintes de communication et de synchronisation permettant à *Cheddar* de réaliser des analyses d'ordonnancement automatiquement.

C. Modèles de communication et de synchronisation entre tâches

Cinq modèles de communications et de synchronisation entre tâches ont été proposés dans [5], [6]. Ces modèles ont été spécifiés avec AADL [9], un langage d'architecture standard utilisé dans le domaine de l'avionique, et avec *Cheddar-ADL* [10] qui est le langage de description d'architecture natif de l'outil *Cheddar*. Chacun de ces modèles exprime un moyen de synchronisation et de communication classique dans les systèmes temps réel et que l'on peut trouver dans les langages, standards ou systèmes d'exploitation utilisés par les acteurs du domaine.

1) **Synchronous data-flow** (LOG_SYNC). Ce premier modèle correspond à une synchronisation/communication classique dans AADL : les communications entre des composants de type *thread* via des *data ports* en mode immédiat. Avec AADL, un composant *thread* modélise un flot de contrôle, i.e. un *thread* est constitué d'une suite d'instructions séquentielles ordonnancée conjointement avec l'ensemble des *threads* déployés sur l'environnement d'exécution. Lorsque deux *threads* AADL communiquent par un *data port*, chaque *thread* lit et écrit les données à des instants spécifiés par le standard AADL. Les instants d'écriture et de lecture des données sont connus et c'est l'environnement

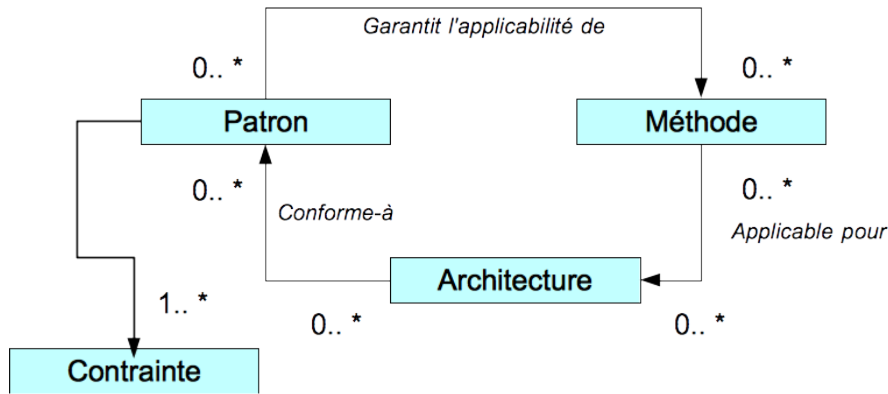

FIGURE 1. Relations entre les patrons de conception, les méthodes d'analyse et les architectures

d'exécution qui gère les problèmes de synchronisation nécessaires pour le partage de ces informations, et ce en garantissant l'absence de perte de données.

2) **Ravenscar** (LOG_RAV). Pour ce modèle, les tâches peuvent échanger des données de façon asynchrone via une mémoire partagée. Les accès à la mémoire partagée sont protégés par un protocole d'évitement d'inversion de priorité, i.e. PCP [11]. Plusieurs contraintes supplémentaires doivent être respectées par les tâches que nous ne détaillerons pas ici. L'ensemble de ces contraintes est issu du standard Ada 2012, et plus précisément de son profil Ravenscar [12].
3) **Blackboard** (LOG_BB). Il s'agit ici d'un modèle lecteurs/rédacteurs. Lectures et écritures s'effectuent de façon asynchrone et seule la dernière valeur écrite peut être lue. Cet ensemble de contraintes modélise un mécanisme de synchronisation/communication défini dans le standard ARINC 653 [13].
4) **Queued buffer** (LOG_QB). Ce modèle exprime une synchronisation de type producteurs/consommateurs où les messages sont produits et consommés selon un ordre FIFO. **Queued buffer** modélise également un mécanisme de synchronisation/communication défini dans le standard ARINC 653.
5) **Unplugged** (LOG_UPG). Ce dernier modèle suppose que les tâches sont indépendantes, i.e. qu'elles ne sont pas synchronisées et qu'elles ne partagent aucune donnée. En d'autres termes, chaque tâche peut être ordonnancé dès son réveil (réveil périodique s'il s'agit d'une tâche périodique) sans autre condition de ressource que la disponibilité de l'unité de calcul.

Le nom entre parenthèse associé à chaque modèle est défini à fin de référence ultérieure dans cet article.

Dans la partie suivante, nous proposons une mise à jour des patrons de conception *Cheddar*, notamment au niveau de la définition des environnements d'exécution, afin de prendre en compte l'évolution actuelle des supports matériels d'exécution.

III. MODÉLISATION DES ENVIRONNEMENTS D'EXÉCUTION

Avec l'augmentation du nombre de transistors implantés sur les puces silicium, les concepteurs de systèmes numériques disposent maintenant d'environnements d'exécution intégrés complexes, offrant de très hautes performances de calcul parallèle, et ce à moindre coût financier. Depuis le début des années 2000, l'augmentation de la puissance de calcul passe principalement par l'augmentation du parallélisme de traitement. Le recours de plus en plus fréquent aux processeurs multi-cœurs ou à cœurs multiples, ou aux MPSoC (Multi-Processor System-on-Chip) hétérogènes illustre cette évolution.

Dans la suite de l'article, nous appellerons, sans distinction, *Multi-Core Processor* ou MCP ces différentes classes d'environnement d'exécution. Leur usage dans les systèmes embarqués va dans le sens de l'optimisation des paramètres SWaP (*Size, Weigth and Power*), et ainsi répond à une contrainte majeure de ce domaine applicatif.

Par exemple, l'industrie avionique intègre l'augmentation des puissances de calcul en exécutant sur un même calculateur mono-processeur plusieurs applications selon l'approche IMA (IMA pour *Integrated Modular Avionic*) [14]. Ce regroupement a été rendu possible au prix d'une ségrégation forte des applications par l'intermédiaire d'une pré-allocation temporelle et spatiale des ressources (approche dite *Time and Space Partionning* [15]). Les acteurs du domaine cherchent maintenant à exploiter les circuits multi-cœurs pour la mise en œuvre de leurs systèmes critiques [16], ou à criticité-mixte [17].

L'évaluation du WCET est l'un des premiers défis soulevés par l'analyse des systèmes supportés par les MCP. Celle-ci a donné lieu à de multiples travaux et outils [18]. Cependant, de nombreuses approches considèrent un environnement d'exécution où les tâches sont isolées, c-à-d que le WCET obtenu ne dépend pas des autres activités au sein du système en absence d'interaction fonctionnelle explicitement définie avec ces dernières.

En pratique, cette hypothèse n'est pas vérifiée pour tous les environnements d'exécution, et notamment dans les MCP. Toutes les méthodes d'analyse, et notamment celles qui se

basent sur un WCET constant indépendant des ressources matérielles partagées entre les unités de calcul, ne peuvent donc pas être utilisées dans tous les cas.

Dans la suite de cette partie, nous proposons un modèle de l'environnement d'exécution qui permet d'identifier les interférences liées aux conflits d'accès à certaines ressources matérielles. Nous qualifierons ces interférences d'*implicites*, dans le sens où elles ne sont pas exprimées dans le modèle de l'architecture logicielle du système, c-à-d par les dépendances entre les tâches de l'application.

A. Modèle d'interférence d'un environnement d'exécution

Un environnement d'exécution peut être modélisé par un ensemble d'entités dont la sémantique est décrite par des attributs afin d'expliciter les interférences existantes au sein d'une architecture matérielle qui peuvent conduire à une variabilité du WCET.

Ici, un environnement d'exécution est un ensemble d'entités de type **Processing Element** et **Shared resource** ainsi que leur relation conduisant à une potentielle interférence :

Définition 1 (Processing Element, PE): Un *Processing Element*, ou PE, est un composant matériel permettant l'exécution d'un flot de contrôle.

Un PE peut être un cœur dans une architecture multi-cœurs ou un MPSoC, un processeur mono-cœur, un *thread* physique (c'est à dire un contexte d'exécution matériel dans le processeur), ou un opérateur spécialisé (une tâche matérielle) [19].

Définition 2 (Shared resource, R): Une *Shared resource*, ou R, est un composant matériel requis par un ou plusieurs PE pour l'exécution d'un ou plusieurs flots de contrôle et conduisant à une potentielle interférence entre ces flots de contrôle.

Une instance de R peut modéliser un cache mémoire, un bus, un *Network-On-Chip* (NoC), ...

Lorsque l'utilisation d'une ressource matérielle partagée entre plusieurs PE n'implique aucune interférence, alors la relation entre PE et la ressource partagée peut être abstraite du modèle d'environnement d'exécution. Ainsi, à titre d'exemple, si un cache partagé entre plusieurs PE est partitionné afin d'éviter tout conflit lors de son utilisation, il n'est pas nécessaire d'expliciter la relation entre cette ressource matérielle et les PE correspondants.

La figure 2 présente le modèle décrivant les interférences entre entités d'un environnement d'exécution. Chaque relation *use* modélise une potentielle interférence entre deux entités.

La figure 3 présente quatre exemples typiques de modèles d'environnement d'exécution. La figure 3 (a) comporte deux PE (PE1 et PE2) indépendants : ces unités de calcul, qui n'exploitent aucune ressource matérielle, ou qui exploitent des ressources matérielles ne conduisant pas à des interférences, ne conduisent pas à une variabilité du WCET. Il s'agit du modèle généralement considéré en ordonnancement multiprocesseurs avec des processeurs identiques [20].

La figure 3 (b) comporte cette fois une ressource matérielle partagée entre les deux PE. Un exemple possible pour ce modèle est le partage d'un bus mémoire requis par PE1 et PE2.

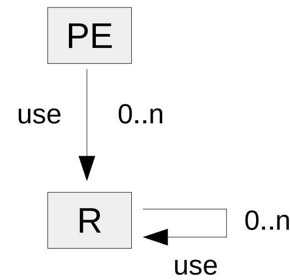

FIGURE 2. Modèle d'un environnement d'exécution

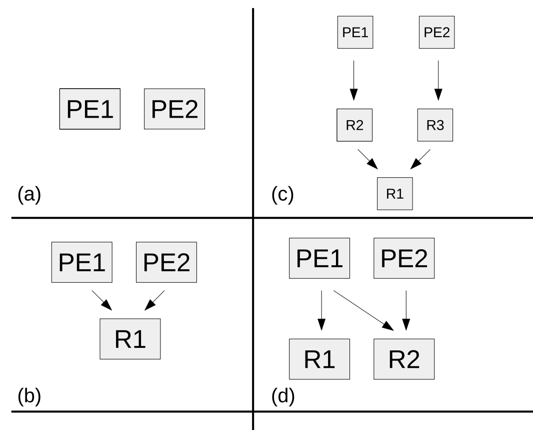

FIGURE 3. Exemples de modèles d'interférences d'un environnement d'exécution

La figure 3 (c) expose un troisième exemple où plusieurs ressources matérielles partagées et privées sont utilisées par PE1 et PE2. R1 modélise ici un cache L2 partagé entre l'ensemble des PE alors que R2 et R3 modélisent respectivement les caches L1 de PE1 et PE2.

Enfin, la figure 3 (d) présente également deux PE dont certaines ressources sont partagées et d'autres privées. Ainsi la ressource matérielle R1 peut modéliser un cache L1 pour PE1, alors que R2 modélise un bus de communication partagé par PE1 et PE2.

À partir du modèle décrit dans la section précédente, nous pouvons classer les différents PE, en fonction des *interférences implicites* qu'ils peuvent subir lors de l'exécution des tâches. Le tableau II présente ces différentes classes de PE.

Chaque entité PE ou R est décrite par un ensemble d'attributs que nous présentons dans la section suivante.

B. Attributs des entités de l'environnement d'exécution

Les attributs (préfixe A) précisent les propriétés intrinsèques d'une entité de l'environnement d'exécution, c'est-à-dire d'une ressource matérielle R ou d'un PE. Ils caractérisent l'entité associée indépendamment de son contexte d'utilisation dans le système.

On peut distinguer les attributs qui renseignent sur la catégorie d'une entité matérielle ; une entité peut assurer une

Nom	Signification
HA_independent(PE)	Vrai si le PE n'utilise pas de ressource matérielle dont le partage pourrait produire des interférences implicites entres les tâches.
HA_isolation(PE)	Vrai si le PE n'utilise pas de ressource matérielle dont le partage pourrait produire une interférence implicite impossible à prédire/éviter au niveau du *run-time* du système.
HA_bounded(PE)	Vrai si le PE n'utilise pas de ressource matérielle dont le partage peut produire une interférence implicite qui ne peut être *a priori* bornée en temps.
HA_dependent(PE)	Vrai si le PE utilise des ressources matérielles dont le partage peut produire une interférence implicite non bornée en temps.

TABLE II
MODÈLES D'INTERFÉRENCES ENTRE PE

fonction de calcul, de communication ou de mémorisation. L'attribut $A_type(E)$ définit ainsi la catégorie de l'entité E :

$A_type := (PE, memory, communication)$

D'autres attributs précisent la mise en œuvre et les caractéristiques structurelles/temporelles de l'entité. Les tableaux III, IV et V donnent une liste d'attributs classée en fonction du type d'entité auquel ils s'appliquent. Cette liste n'est évidemment pas exhaustive et peut être étendue pour une méthode d'analyse donnée.

Les attributs interviennent à 2 niveaux dans la définition d'un patron de conception. Ils participent pour certains à la sélection d'une méthode d'analyse spécifique, et pour d'autres au paramétrage de la méthode sélectionnée. Par exemple, $A_mem_cache_associativity$ permet de vérifier si une méthode supposant des caches à correspondance directe est adaptée, ou dans le cas de méthodes prenant en compte tout type de cache, est utilisé pour calculer le placement des données dans ce dernier.

C. Attributs d'accès

Les attributs d'accès (préfixe AM) définissent quand et comment les PE accèdent aux ressources matérielles R. L'architecture de l'environnement d'exécution permet d'identifier une potentielle interférence liée au partage d'une ressource matérielle. Cependant, la ressource matérielle peut être une entité dont les éléments le constituant peuvent être utilisés indépendamment les uns des autres. D'autre part, les intervalles de temps où les PE accèdent à la ressource peuvent être disjoints par construction du système. Dans ces deux cas, les accès à la ressource matérielle ne sont pas susceptibles de créer des interférences. Si la conception du système n'interdit pas un accès concurrent, le patron de conception doit exprimer comment le conflit d'accès est traité.

Les trois attributs ci-dessous représentent respectivement les intervalles de temps pendant lesquels un PE est autorisé à accéder à une ressource matérielle, la partition (au sens d'ARINC 653) de la ressource matérielle allouée au PE, et en cas de conflit d'accès, la politique d'arbitrage utilisée :

Nom de l'attribut	Signification
AM_time(PE, R)	ensemble des instants où le PE est autorisé à accéder à la ressource matérielle R
AM_space(PE, R)	sous-partie ou partition de R à laquelle PE est autorisé à accéder
AM_arbitration(PE,. . .,PE,R)	politique d'arbitrage en cas d'accès concurrent d'un ensemble de PE à R.

Par exemple, un ensemble de 2 unités de calcul PE1 et PE2 accédant à un bus selon une trame TDM se caractérisera par la relation suivante :

$$AM_time(PE1, Bus) \cap AM_time(PE2, Bus) = \emptyset$$

Notez que les attributs d'accès ne fournissent pas les mêmes informations que les attributs décrivant l'ordonnancement des tâches logicielles sur les PE. Une tâche peut être active sur un PE alors que le PE est bloqué en attente d'une ressource matérielle. Inversement, un PE peut disposer d'un accès exclusif à une ressource matérielle alors qu'aucune tâche logicielle ne nécessite d'accès à cette ressource.

D. Attribut de déploiement

Les attributs de déploiement (préfixés DM) indiquent comment les ressources matérielles sont attribuées aux entités logicielles. Comme dans AADL [9], ils définissent soit une affectation effective, soit une autorisation d'affectation.

Nom de l'attribut	Signification
DM_PE_actual(T)	PE où la tâche T est effectivement exécutée
DM_PE_allowed(T)	ensemble des PE autorisés à exécuter la tâche T

Même dans le cas où une tâche est autorisée à s'exécuter sur plusieurs PE, un *job* de cette tâche ne sera traité que par un seul PE à un instant donné, c'est à dire que le code de la tâche n'est pas parallélisé.

A priori, une tâche s'exécutant sur un PE peut utiliser l'ensemble des ressources matérielles accessibles par ce PE dans son environnement. Les attributs de déploiement d'une tâche permettent de définir cet ensemble explicitement, et donc de le restreindre si nécessaire.

Nom de l'attribut	Signification
DM_R_actual(T)	ensemble des ressources matérielles que la tâche T utilise effectivement
DM_R_allowed(T)	ensemble des ressources matérielles dont l'accès est autorisé à la tâche T

Le déploiement des tâches sur les entités matérielles partagées est précisé par une politique d'ordonnancement qui régit les intervalles de temps pendant lesquelles l'entité est affectée à la tâche. Les attributs de déploiement $DM_PE_scheduling$ et $DM_R_scheduling$ définissent les

Nom de l'attribut	Type	Signification
A_men_type(R)	(memory, DCache, ICache, IDCache, hierarchy)	sous-type d'une entité (banc mémoire, cache de données, cache d'instruction, ...)
A_mem_cache_associativity(R)	entier ≥ 1	associativité du cache
A_mem_cache_replacement_policy(R)	(LRU, LRR, random)	politique de remplacement pour les caches associatifs
A_mem_cache_miss_time(R)	ns	temps de chargement d'une ligne cache (BRT ou *Block Reload Time*) en cas d'échec
A_mem_cache_size(R)	octets	taille totale du cache ou d'une partition du cache
A_mem_cache_line_size(R)	octets	taille d'une ligne du cache
A_mem_cache_level(R)	entier	niveau du cache par rapport au processeur
A_mem_cache_coherency(R)	(copy_back, write_through)	stratégie d'écriture, protocole de cohérence
A_mem_memory_access_time(R)	ns	temps d'accès à un mot enregistré en mémoire

TABLE III
EXEMPLES D'ATTRIBUTS APPLIQUÉS AUX RESSOURCES DE LA CATÉGORIE "MÉMORISATION"

Nom de l'attribut	Type	Signification
A_PE_type(PE)	(core, processor, thread, dedicated)	sous-type (cœur de processeur ou de MPSoC, *thread* physique, processeur physique, opérateur spécialisé)
A_PE_isa(PE)	string	jeu d'instructions supporté
A_PE_speed(PE)	{operation/s, operation/s}	vitesse, ou intervalle de vitesse dans le cas d'un contôle DVFS (Dynamic Voltage and Frequency Scaling.), éventuellement relative aux PE supportant la même ISA.

TABLE IV
EXEMPLES D'ATTRIBUTS APPLIQUÉS AUX RESSOURCES DE LA CATÉGORIE "UNITÉ DE CALCUL"

Nom de l'attribut	Type	Signification
A_comm_type(R)	(bus, NoC, star, p2p)	sous-type communication (bus, *Network-on-Chip*, étoile, point-à-point, ...)
A_comm_throughput(R)	mots/s	débit maximum de transfert sur un bus
A_comm_latency(R)	ns	latence maximum d'un transfert si le bus est disponible

TABLE V
EXEMPLES D'ATTRIBUTS APPLIQUÉS AUX RESSOURCES DE LA CATÉGORIE "COMMUNICATION"

paramètres de la politique d'ordonnancement.

Nom de l'attribut	Signification
DM_PE_scheduling (PE, T)	ensemble des paramètres qui définissent la politique d'ordonnancement d'une tâche T sur un PE
DM_R_scheduling (R,T)	ensemble des paramètres qui définissent la politique d'ordonnancement d'une tâche T lors de l'accès à une ressource R

L'accès effectif à une ressource partagée R à un instant t est conditionné par l'ordonnancement sur un PE de la tâche y demandant l'accès, et en second lieu, par l'autorisation donnée à ce PE d'utiliser la dite ressource. Autrement dit, nous sommes en présence d'un ordonnancement hiérarchique, le plus souvent géré à des échelles de temps très dissemblables. Par exemple, dans le cas d'un processeur double cœurs avec une mémoire partagée, l'instruction de lecture mémoire d'une tâche va s'exécuter sans attente si la tâche est ordonnancée sur un cœur et si l'arbitrage du bus mémoire l'autorise à accéder à la mémoire.

E. Mise à jour des patrons de conception de l'outil Cheddar

Dans les parties précédentes, nous avons proposé un modèle de l'environnement d'exécution afin d'expliciter les interférences implicites au sein de ces environnements. Nous décrivons maintenant comment les patrons de conception de l'outil *Cheddar* sont adaptés afin de pouvoir les appliquer à des environnements d'exécution MCP. Dans ce contexte, un patron de conception est donc à présent constitué :

1) **De contraintes de synchronisations et de communication entre les tâches**. Pour la suite, nous considérons les ensembles *Synchronous data-flow*, *Ravenscar*, *Blackboard*, *Queued buffer* et *Unplugged* (section II-C).
2) **D'un modèle d'architecture matérielle, complété éventuellement d'attributs d'accès**. Ces modèles permettent d'exhiber les potentielles interférences entre les entités matérielles (sections III-A et III-C).
3) **D'attributs qui caractérisent les fonctions intrinsèques des entités matérielles**, ressources R ou PE, de l'environnement d'exécution (section III-B).
4) **D'attributs de déploiement** qui indiquent comment les entités logicielles (i.e. les tâches) sont déployées sur les PE et les ressources R (section III-D).
5) Et finalement **d'un ensemble de méthodes d'analyse d'ordonnancement** qui peuvent être appliquées aux

systèmes conformes aux contraintes des points (1), (2), (3) et (4).

Dans la partie suivante, nous utilisons donc ces éléments de modélisation pour définir les environnements d'exécution sous forme de patrons de conception.

IV. Patrons pour l'analyse d'ordonnancement multiprocesseurs

Dans cette section, nous listons les méthodes d'analyses applicables dans le contexte d'un MCP et qui sont disponibles dans l'outil *Cheddar*. Les conditions qui autorisent leur utilisation sont précisées par un patron de conception exprimé selon les attributs que nous avons définis précédemment.

A. *Fonctions d'analyse implantées dans* Cheddar

Les fonctions d'analyse destinées aux systèmes temps réel multiprocesseurs ou répartis de *Cheddar* peuvent être classées en trois catégories.

La première catégorie est fondée sur la simulation. Il s'agit ici de produite une simulation de l'ordonnancement du jeu de tâches, si possible sur l'intervalle de faisabilité si celui-ci existe [21], puis, de calculer divers critères de performances (pire temps de réponse, de temps de blocage, absence de *deadlock*, d'inversion de priorité, nombre de préemption, de commutation de contexte, ...). Les simulations possibles à ce jour portent sur des ordonnancements multiprocesseurs globaux ou partitionnés, avec des algorithmes classiques (priorités fixes, EDF, LLF, ...) ou des algorithmes spécifiques tels que EDZL ou Proportionate-Fair. Les simulations peuvent être configurées via des paramètres sur les ordonnanceurs (niveau de préemption, quantum) ou sur les entités de l'architecture à vérifier (*jitter*, *offset* et plus généralement modèle de tâche, protocole d'accès aux ressources partagées, ...).

La seconde catégorie de méthodes d'analyse est basée sur l'utilisation de tests de faisabilité. Dans le cadre d'architectures multiprocesseurs, il s'agit principalement de méthodes de calcul de pire temps de réponse sur des transactions arbres ou linéaires [22], [23], [24], [25], [26], ou des tests sur le taux d'occupation des processeurs [20]. Certains outils permettent également de calculer diverses données à agréger aux pires temps de réponse (par exemple le délai de préemption lié aux caches, i.e. CRPD [27] ou le temps de blocage sur les ressources partagées).

Enfin, la dernière catégorie de méthodes regroupe des outils utilisables lors de l'exploration d'architectures. Dans un contexte multiprocesseurs, ce peut être, par exemple, des méthodes de partitionnement, des méthodes d'affectation de divers paramètres des tâches pour la prise en compte des ressources matérielles (ex : affectation des priorités selon les CRPD), ou de la répartition et des contraintes de précédences impliquées par les communications.

B. *Structuration des patrons de conception*

La définition d'un patron de conception implique la mise en place de contraintes applicables sur différents niveaux des couches de système. Afin de simplifier leur formalisation, les patrons sont exprimés par plusieurs ensembles de contraintes, éventuellement ré-utilisables.

La figure 4 présente les ensembles de contraintes qui seront utilisés dans la suite de cet article. Le préfixe de leur nom rappelle à quel niveau ils interviennent :

- LOG : contraintes sur les synchronisations et communications entre les tâches,
- DEP : contraintes sur le déploiement,
- EXE : contraintes sur les interférences entre entités matérielles et sur leurs accès aux ressources,
- FEA : contraintes sur les caractéristiques des entités matérielles.

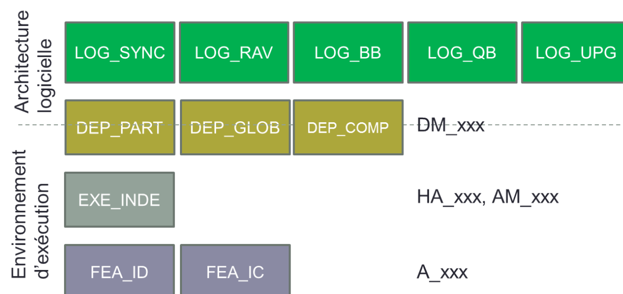

FIGURE 4. Structuration des patrons de conception par couche du système.

Ces ensembles de contraintes sont définis dans la suite de la section. Pour ces définitions, T désigne l'ensemble des tâches du système à analyser, et P l'ensemble des PE disponibles dans l'environnement d'exécution.

a) Tâches compatibles (DEP_COMP) : Cet ensemble contraint le déploiement des tâches en imposant qu'elles soient affectées à des ensembles disjoints de PE. Chaque tâche est déployée vers un seul ensemble et la migration n'est pas possible entre ces ensembles.

$$\begin{array}{l} \forall t, s \in T, \\ \|DM_PE_allowed(t)\| \geq 1 \wedge \\ DM_PE_allowed(t) \subset P \wedge \\ (\\ \quad DM_PE_allowed(t) = DM_PE_allowed(s) \vee \\ \quad DM_PE_allowed(t) \cap DM_PE_allowed(s) = \emptyset \\ ) \end{array}$$

b) Partitionnement des tâches par PE (DEP_PART): L'ensemble des tâches est partitionné sur des processeurs du système, et l'ordonnancement des tâches sur chaque processeur doit être géré par un même algorithme.

$$\begin{array}{l} \forall t \in T, \\ \exists p \in P \setminus DM_PE_actual(t) = p \wedge \\ DM_PE_scheduling(p, t) \ exists \wedge \\ DM_PE_scheduling(p, t) = DM_PE_scheduling(p, s) \end{array}$$

c) Partitionnement des tâches par ensemble de PE (DEP_GLOB) : Les ensembles de PE définis par le déploiement doivent être les mêmes, ou dans le cas contraire, disjoints. Toutes les tâches du système à analyser doivent être déployées sur au

moins un PE. Au sein d'un ensemble de PE, un protocole d'ordonnancement unique doit être utilisé.

$$\forall t, s \in T,$$
$$\|DM_PE_allowed(t)\| \geq 1 \wedge$$
$$DM_PE_allowed(t) \subset P \wedge$$
$$($$
$$DM_PE_allowed(t) = DM_PE_allowed(s) \vee$$
$$DM_PE_allowed(t) \cap DM_PE_allowed(s) = \emptyset$$
$$)$$
$$\wedge$$
$$\forall t, s \in T \backslash DM_PE_allowed(t) = DM_PE_allowed(s),$$
$$\forall p, q \in DM_PE_allowed(t),$$
$$DM_PE_scheduling(p, t)\ exists \wedge$$
$$DM_PE_scheduling(p, t) = DM_PE_scheduling(q, s)$$

d) PE multiples indépendants (EXE_INDE) :

Un environnement d'exécution conforme à cet ensemble garantit que le comportement temporel et fonctionnel des PE qui exécutent une tâche est indépendant de l'état des autres PE, en absence d'interférence explicitement modélisée dans le système.

$$\forall t \in T,$$
$$\forall p \in DM_PE_allowed(t) \cup DM_PE_actual(t),$$
$$HA_independent(p)$$

e) PE identiques (FEA_ID) :
Les PE sont uniquement caractérisés du point de vue de leurs performances en terme de vitesse d'exécution, et du type de code qu'ils sont en mesure d'exécuter. La vitesse d'exécution peut être éventuellement définie relativement à celle des autres PE.

$$\forall p, q \in \bigcup_{t \in T} DM_PE_allowed(t) \cup DM_PE_actual(t),$$
$$($$
$$A_PE_speed(p) = A_PE_speed(q) \vee$$
$$\neg(A_PE_speed(p)\ exits) \vee \neg(A_PE_speed(p)\ exits)$$
$$) \wedge ($$
$$A_PE_isa(p) = A_PE_isa(q) \vee$$
$$\neg(A_PE_isa(p)\ exits) \vee \neg(A_PE_isa(p)\ exits)$$
$$)$$

Si l'attribut de vitesse ou de type n'existe pas pour un PE, on considère par défaut qu'il est identique à celui des autres PE.

Si la vitesse et le type ne sont définis pour aucun PE, alors ceux-ci sont considérés comme des entités abstraites fournissant des "unités" pré-définies de puissance de calcul, à mettre en relation avec le paramétrage du modèle de tâches à analyser.

f) PE disposant d'un cache d'instructions privé (FEA_IC) :
Les PE contiennent un cache d'instructions privé de niveau 1, à correspondance directe.

$$\forall p \in \bigcup_{t \in T} DM_PE_allowed(t) \cup DM_PE_actual(t),$$
$$\|USE(p)\| = 1 \wedge$$
$$\forall r \in USE(p),$$
$$A_mem_type(r) = Icache \wedge$$
$$A_mem_associativity(r) = 1 \wedge$$
$$A_mem_miss_time(r) \leq constante$$

C. Patrons de conception et analyses applicables

Par la suite, à partir des ensembles de contraintes ci-dessus, nous spécifions 4 patrons de conception nommés respectivement DP1, DP2, DP3 et DP4.

a) DP1: Un environnement d'exécution conforme à ce patron autorise un groupe de tâches à s'exécuter sur un groupe de PE identiques et indépendants. Les groupes de tâches et de PE sont disjoints.

$$DP1 := (LOG_UPG \vee LOG_SYNC) \wedge DEP_COMP(T, PE) \wedge EXE_INDE(T, PE) \wedge FEA_ID(T, PE)$$

Le respect de ce patron permet d'utiliser les méthodes de partitionnement de jeux de tâches disponibles dans *Cheddar*. Plusieurs heuristiques de partitionnement d'un ensemble de tâches indépendantes sont implantées : *Best Fit*, *First Fit*, *Next Fit*, *Small Task* et *General Task* [28], [29].

b) DP2: Un environnement d'exécution conforme à ce patron définit un PE unique pour l'exécution de chacune des tâches du système, et une politique d'ordonnancement temporelle pour chaque PE. Les PE sont indépendants et identiques.

$$DP2 := (LOG_UPG \vee LOG_RAV \vee LOG_SYNC) \wedge DEP_PART(T, PE) \wedge EXE_INDE(T, PE) \wedge FEA_ID(T, PE)$$

Ce patron définit un système multiprocesseurs partionné. Selon la conformité de l'architecture logicielle, ce patron permet d'appliquer de nombreuses méthodes d'analyse implantées dans *Cheddar*.

A titre d'exemple, si l'architecture à vérifier est conforme à LOG_RAV ou LOG_SYNC ou LOG_UPG, nous pouvons y appliquer un calcul de pire temps de réponse [30].

c) DP3: Un environnement d'exécution conforme à ce patron autorise un groupe de tâches à s'exécuter sur un groupe de PE identiques et indépendants. Les groupes de tâches et de PE sont disjoints. Tous les PE d'un groupe sont gérés par la même politique d'ordonnancement.

$$DP3 := (LOG_UPG \vee LOG_RAV \vee LOG_SYNC) \wedge DEP_GLOB(T, PE) \wedge EXE_INDE(T, PE) \wedge FEA_ID(T, PE)$$

Ce patron spécifit un système multiprocesseurs avec un ordonnancement global. Divers algorithmes sont disponibles

dans *Cheddar*, qui contrôlent dynamiquement l'ordonnancement des tâches du système sur un ensemble de PE. Certains sont des adaptations des algorithmes classiques utilisés en environnement monoprocesseur (RM, DM, EDF, ...), et d'autres ont été spécifiquement développés pour le contexte multiprocesseurs (EDZL, P-Fair, LLREF, ...) [31].

Par ailleurs, *Cheddar* évalue un test de faisabilité dans le contexte d'un ordonnancement global par la méthode Proportionate-FAIR sur m processeurs identiques.

d) DP4: Un environnement d'exécution conforme à ce patron définit un PE unique pour l'exécution de chacune des tâches du système, et une politique d'ordonnancement temporelle pour chaque PE. Les PE sont indépendants, identiques, et contiennent un cache d'instructions privé de niveau 1 à correspondance directe.

$$\begin{aligned} DP4 \; := \; & (LOG_UPG \vee LOG_SYNC) \wedge \\ & DEP_PART(T, PE) \wedge \\ & EXE_INDE(T, PE) \wedge \\ & FEA_ID(T, PE) \wedge FEA_IC(T, PE) \end{aligned}$$

La conformité à ce patron donne accès aux méthodes qui, dans *Cheddar*, prennent en compte le CRPD (*Cache Related Preemption Delay*), c-à-d la simulation avec CRPD [32], le calcul des interférences entre tâches dues au cache d'instructions [33], et l'extension de l'algorithme optimal proposé par Audsley [34] pour l'affectation des priorités.

D. Exemples d'analyse

Dans cette partie, nous illustrons les patrons avec deux exemples d'analyse d'ordonnancement multiprocesseurs en nous focalisant sur la modélisation de l'environnement d'exécution.

1) Exemple 1, ordonnancement global : Nous présentons ici l'analyse d'une architecture logicielle composée de 4 tâches périodiques, $t0$, $t1$, $t2$, et $t3$, ordonnancées sur un environnement d'exécution comprenant deux processeurs $C0$ et $C1$. Les deux processeurs sont identiques et indépendants. L'architecture logicielle est conforme à `LOG_UPG`.

Les attributs ci-dessous caractérisent la partie matérielle de l'environnement d'exécution.

$$\begin{aligned} & HA_independent(C0) \\ & HA_independent(C1) \\ & A_PE_isa(C0) = A_PE_isa(C1) = "PowerPC" \\ & A_PE_speed(C0) = A_PE_speed(C1) = 100.10^6 \end{aligned}$$

Le concepteur de cette architecture choisit un ordonnancement global de type PFair (*Proportionnate-Fair*). Les règles de déploiement ci-dessous montrent que les tâches logicielles sont autorisées à s'exécuter sur tous les processeurs et définissent le protocole de partage des ressources de calcul :

$$\begin{aligned} & DM_PE_allowed(t0) = \{C0, C1\} \\ & DM_PE_allowed(t1) = \{C0, C1\} \\ & DM_PE_allowed(t2) = \{C0, C1\} \\ & DM_PE_allowed(t3) = \{C0, C1\} \\ & Let\ sched = \{PFair, preemptive, time_unit_migration\} \\ & DM_PE_scheduling(C0, t0) = sched \\ & DM_PE_scheduling(C0, t1) = sched \\ & DM_PE_scheduling(C0, t2) = sched \\ & DM_PE_scheduling(C0, t3) = sched \\ & DM_PE_scheduling(C1, t0) = sched \\ & DM_PE_scheduling(C1, t1) = sched \\ & DM_PE_scheduling(C1, t2) = sched \\ & DM_PE_scheduling(C1, t3) = sched \end{aligned}$$

Le modèle de l'environnement d'exécution est conforme au patron DP3, il est donc possible d'utiliser *Cheddar* pour simuler l'exécution des tâches à partir de ces hypothèses (voir la figure 5).

2) Exemple 2, effet des caches d'instructions sur l'ordonnancement : Pour ce second exemple, on cherche à analyser un système temps réel implanté sur un environnement d'exécution dual-cœurs ($C0$ et $C1$). Les instructions sont enregistrées dans une mémoire partagée, et chaque processeur dispose d'un cache d'instructions privé de niveau 1. La figure 6 schématise l'environnement d'exécution considéré. Les *Scratchpad Memories (SPM)* sont utilisées pour enregistrer les données et le contexte d'exécution des tâches. La mémoire partagée stocke uniquement les instructions. La bande passante du bus mémoire est allouée équitablement aux deux cœurs par une trame TDM composée de 2 slots d'égale durée.

La liste des attributs qui caractérisent cette architecture, et qui permettra de fixer certains paramètres de l'analyse d'ordonnancement, est donnée ci-dessous :

$$\begin{aligned} & A_type(C0) = A_type(C1) = PE \\ & A_type(IC0) = A_type(IC1) = memory \\ & A_type(MB) = comm \\ & A_PE_isa(C0) = A_PE_isa(C1) = "SPARC_V8" \\ & A_PE_speed(C0) = A_PE_speed(C1) = 100.10^6 \\ & A_mem_type(IC0) = A_mem_type(IC1) = ICache \\ & A_mem_level(IC0) = A_mem_level(IC1) = 1 \\ & A_mem_cache_size(IC0) = 1.2^{10} \\ & A_mem_cache_size(IC1) = 1.2^{10} \\ & A_mem_cache_line_size(IC0) = 16 \\ & A_mem_cache_line_size(IC1) = 16 \\ & A_mem_miss_time(IC0) = 500 + 500 // voir\ plus\ bas \\ & A_mem_miss_time(IC1) = 500 + 500 \\ & A_comm_type(MB) = bus \end{aligned}$$

Sont définies ensuite les interférences liées au partage de ressources matérielles au sein de l'environnement d'exécution. Les cœurs utilisent la même mémoire pour stocker leurs instructions et y accèdent par un même bus (MB). L'accès effectif à la mémoire étant conditionné par l'accès au bus, le modèle représente les règles d'accès à ce dernier uniquement, c'est-à-dire sa trame TDM. Le temps de blocage additionnel

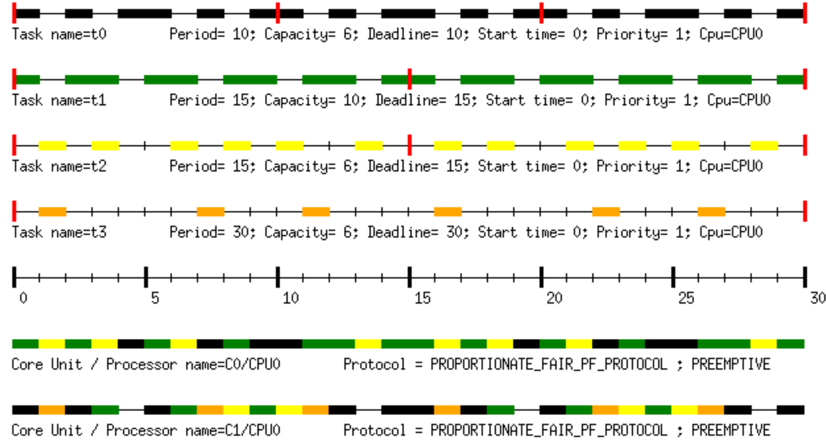

FIGURE 5. Ordonnancement global PFair, sur 2 processeurs identiques.

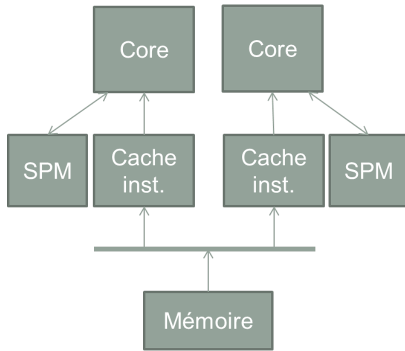

FIGURE 6. Environnement d'exécution.

que peut subir le cache pour traiter un échec dépend de la durée du slot affecté à l'autre processeur. Comme ce temps est statique et connu, il est possible d'en déduire que la durée de l'interférence est bornée si le nombre d'échecs cache l'est aussi (modèle d'interférence `HA_isolation`).

Comme l'interférence implicite s'exprime au moment des échecs cache, il est possible de l'intégrer dans le temps de traitement de l'échec, et d'utiliser le modèle d'interférence `HA_independent`. En d'autres termes, pour cette architecture, nous pouvons abstraire les interférences liées au bus mémoire. Notons que ce temps actualisé doit être aussi pris en compte lors du calcul du WCET des tâches.

$$HA_independent(C0)$$
$$HA_independent(C1)$$
$$AM_time(C0, MB) = \{[0 + 1000.k, 500 + 1000.k[\mid k \geq 0\}$$
$$AM_time(C1, MB) = \{[500 + 1000.k, 1000.(k+1)[\mid k \geq 0\}$$

Pour cet exemple, l'architecture logicielle est composée de 4 tâches, nommées $t0$, $t1$, $t2$ et $t3$. Leur code ainsi que leur position en mémoire sont connus. L'architecture logicielle est conforme à `LOG_UPG`. Le concepteur a choisi d'affecter les tâches $t0$ et $t1$ au premier processeur, et les tâches $t2$ et $t3$ au second, comme le montre le modèle de déploiement suivant :

$$DM_PE_actual(t0) = C0$$
$$DM_PE_actual(t1) = C0$$
$$DM_PE_actual(t2) = C1$$
$$DM_PE_actual(t3) = C1$$
$$DM_PE_scheduling(C0, t0) = \{FP, preemptive\}$$
$$DM_PE_scheduling(C0, t1) = \{FP, preemptive\}$$
$$DM_PE_scheduling(C1, t2) = \{FP, preemptive\}$$
$$DM_PE_scheduling(C1, t3) = \{FP, preemptive\}$$

L'ensemble des attributs de cet environnement d'exécution est conforme au patron de conception DP4, et l'analyse d'ordonnancement peut être effectuée par une simulation avec CRPD. La figure 7 montre le résultat de simulation produit par *Cheddar*.

V. TRAVAUX CONNEXES

L'un des domaines applicatifs où l'analyse d'ordonnancement est nécessaire, est celui des systèmes de transports, dont les systèmes avioniques font partie. Actuellement, il n'existe pas de procédure de certification établie qui s'applique à l'utilisation de MCP dans les avions civiles. Cependant, les autorités de certification mènent un travail exploratoire sur le sujet. Dans [16], la CAST (*Certification Authorities Software Team*) expose certaines pistes de travail, et indique que le partage des caches mémoire dans les MCP est un point clef du problème. La modélisation des patrons de conception que nous proposons se focalisent sur ce point.

Plus généralement, plusieurs outils d'analyse de l'ordonnancement pour les MCP ont été proposés ces dernières années : STORM, RealtssMP, Yartiss, SimSo ou MAST en sont des exemples [35], [36], [37], [38], [39]. Certains proposent d'étudier les ressources matérielles partagées et les interférences qu'elles peuvent conduire.

Toutefois, ces outils ne proposent pas de mécanismes permettant de s'assurer que les modèles analysés sont conformes aux hypothèses d'applicabilité.

L'approche par patron de conception que nous proposons est une application des travaux menés sur Ravenscar [40].

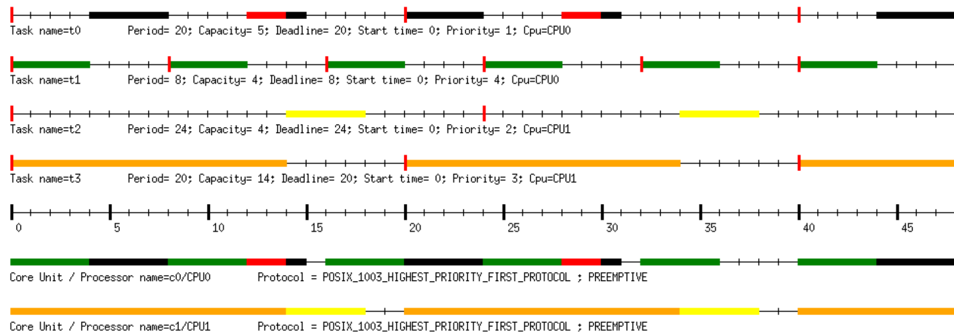

FIGURE 7. Ordonnancement partitionné, sur 2 processeurs (chacun disposant d'un cache d'instructions privé). Les zones rouges représentent le temps additionnel dû aux interférences sur l'utilisation du cache d'instructions en cas de préemption. Dans le cas du processeur CPU1, aucune pré-emption n'est observée entre les tâches t2 et t3.

L'utilisation de patrons de conception pour la vérification a été expérimentée dans le cadre de divers systèmes, et pour diverses propriétés telles que les performances, la sureté voire la sécurité [5], [41], [42], approche parfois étendue par la notion de contrats [43].

A notre connaissance, peu de travaux ont proposés d'appliquer une approche basée sur les patrons pour la vérification de MCP [44] lorsqu'il s'agit de prendre en compte les ressources matérielles partagées de l'environnement d'exécution.

VI. CONCLUSION

Avec la diffusion rapide des environnements d'exécution de type MCP, être en mesure de vérifier les propriétés temporelles de systèmes s'exécutant dans ces environnements est un défi majeur pour la communauté de l'analyse d'ordonnancement temps-réel.

L'outil *Cheddar* offre plusieurs méthodes d'analyse d'ordonnancement adaptées aux plateformes MCP. Il s'agit essentiellement de moyens de simulation, de tests de faisabilité ou de méthodes permettant de réaliser une exploration d'architecture du système à implanter.

Toutefois, l'utilisation de ces méthodes d'analyse reste difficile. En effet, un outil comme *Cheddar* peut offrir de nombreuses méthodes d'analyse et chaque méthode requiert que le système à analyser soit conforme à plusieurs hypothèses d'applicabilité. Par ailleurs, le contexte des MCP impose d'intégrer un modèle du support matériel d'exécution permettant d'expliciter les interferences provenant des ressources matérielles. Ces interférences sont difficiles à exhiber et à comprendre par l'utilisateur de l'outil d'analyse d'ordonnancement.

L'objectif de cet article est de formaliser les contraintes d'applicabilité des méthodes d'analyse implantées dans *Cheddar*, et en particulier les interférences dues aux ressources matérielles de l'environnement d'exécution. Nous proposons une première formalisation et nous montrons comment l'appliquer avec deux exemples.

Pour évaluer ces propositions, nous souhaitons prototyper ces patrons au sein du produit *AADL Inspector* [1] qui intègre déjà l'outil Cheddar. Pour ce faire, une solution possible consiste à exploiter les capacités de la technologie LMP (Logic Model Processing) [45], [46]. LMP est actuellement utilisée pour l'intégration de Cheddar au sein d'*AADL Inspector* et consiste à exprimer les éléments descriptifs de l'architecture étudiée (comme le modèle de l'environnement d'exécution en terme d'unité de calcul et de ressource matérielle) par une base de faits *prolog* et l'ensemble des contraintes à appliquer (comme les contraintes d'accès ou de déploiement) par une base de règles *prolog*.

1. http://www.ellidiss.fr/public/wiki/wiki/inspector

REMERCIEMENTS

Cheddar est financé par Ellidiss Technologies, par Brest Métropole, Région Bretagne, par le CD du Finistère, par Thalès TCS et par Campus France PESSOA programmes numéro 27380SA et 37932TF.

RÉFÉRENCES

[1] J. Stankovic, S. H. Son, J. Hansson *et al.*, "Misconceptions about real-time databases," *Computer*, vol. 32, no. 6, pp. 29–36, 1999.

[2] S. Fürst, "Challenges in the design of automotive software," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. European Design and Automation Association, IEEE, Mar 2010, pp. 256–258.

[3] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar : a flexible real-time scheduling framework," *ACM SIGAda Ada Letters*, vol. 24, no. 4, pp. 1–8, December 2004, ACM Press, New York, USA.

[4] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, and J. Legrand, "An Ada design pattern recognition tool for AADL performance analysis," *Ada Letters*, vol. 31, no. 3, pp. 61–68, November 2011.

[5] A. Plantec, F. Singhoff, P. Dissaux, and J. Legrand, "Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns," in *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation-Volume Part I*. Springer-Verlag, 2010, pp. 4–17.

[6] V. Gaudel, "Applicabilité des méthodes d'analyse et interopérabilité des outils de développement pour systèmes embarqués temps-réel critiques," Ph.D. dissertation, December 2014.

[7] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, *Scheduling in Real-Time Systems*. Wiley Online Library, 2002.

[8] P. Dissaux and F. Singhoff, "Stood and cheddar : Aadl as a pivot language for analysing performances of real time architectures," in *Proceedings of the European Real Time System conference. Toulouse, France*, vol. 32, 2008.

[9] P. H. Feiler, B. A. Lewis, and S. Vestal, "The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems," in *Proceedings of the Conference on Computer Aided Control System Design, of the International Conference on Control Applications, and of the International Symposium on Intelligent Control*. IEEE, 2006, pp. 1206–1211.

[10] C. Fotsing, F. Singhoff, A. Plantec, V. Gaudel, S. Rubini, S. Li, H. N. Tran, L. Lemarchand, P. Dissaux, and J. Legrand, "Cheddar architecture description language," *Lab-STICC technical report*, 2014.

[11] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols : An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[12] A. Burns, B. Dobbing, and G. Romanski, "The ravenscar tasking profile for high integrity real-time programs," in *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*. London, UK : Springer-Verlag, 1998, pp. 263–275.

[13] W. Barnes, "ARINC 653 and why is it important for a safety-critical RTOS," *Boards & Solutions*, p. 16, 2004.

[14] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, Oct 2007, pp. 2.A.1–1–2.A.1–10.

[15] J. Windsor and K. Hjortnaes, "Time and space partitioning in spacecraft avionics," in *in Proceedings of the Third International Conference on Space Mission Challenges for Information Technology*. IEEE, July 2009, pp. 13–20.

[16] "Multi-core processors (rev 0), position paper (CAST32a)," Certification Authorities Software Team, Tech. Rep., 2016.

[17] R. Obermaisser and D. Weber, "Architectures for mixed-criticality systems based on networked multi-core chips," in *Proceedings of the 19th International Conference on Emerging Technology and Factory Automation (ETFA)*. IEEE, Sept 2014, pp. 1–10.

[18] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[19] S. Rubini, C. Fotsing, P. Dissaux, F. Singhoff, and H. N. Tran, "Scheduling analysis from architectural models of embedded multi-processor systems," *ACM SIGBED Review*, vol. 11, no. 1, 2014.

[20] J. Goossens, "Ordonnancement temps réel multiprocesseur," in *Systèmes temps réel T. 2 - Ordonnancement, réseaux et qualité de service*, ser. Traité IC2, Information - Commande - Communication, N. Navet, Ed. Hermès - Lavoisier, 2006, ch. 2, p. 336.

[21] J. Goossens, E. Grolleau, and L. Cucu-Grosjean, "Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms," *Real-time systems*, vol. 52, no. 6, pp. 808–832, 2016.

[22] N. C. Audsley, K. Tindell, and A. Burns, "The end of the line for static cyclic scheduling ?" in *RTS*, 1993, pp. 36–41.

[23] K. Tindell, *Adding time-offsets to schedulability analysis*. University of York, Department of Computer Science, 1994.

[24] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and microprogramming*, vol. 40, no. 2-3, pp. 117–134, 1994.

[25] J. C. Palencia and M. G. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*. IEEE, 1998, pp. 26–37.

[26] S. Li, F. Singhoff, S. Rubini, and M. Bourdellès, "Scheduling analysis of tasks constrained by tdma : Application to software radio protocols," *Journal of Systems Architecture*, vol. 76, pp. 58–75, 2017.

[27] S. Altmeyer, R. I. Davis, and C. Maiza, "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *Real-Time Systems*, vol. 48, no. 5, pp. 499–526, 2012.

[28] Y. Oh and S. H. Son, "Tight performance bounds of heuristics for a real-time scheduling problem," *Submitted for Publication*, 1993.

[29] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son, "Assigning real-time tasks to homogeneous multiprocessor systems," *submitted for publication*, 1994.

[30] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[31] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[32] H. N. Tran, F. Singhoff, S. Rubini, and J. Boukhobza, "Cache-aware real-time scheduling simulator : implementation and return of experience," *ACM SIGBED Review*, vol. 13, no. 1, pp. 22–28, 2016.

[33] H. N. Tran, "Cache memory aware priority assignment and scheduling simulation of real-time embedded systems," Ph.D. dissertation, Jan 2017.

[34] N. C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," in *Technical Report YCS 164, Dept. Computer Science*. University of York, UK, 1991.

[35] M. G. Harbour, J. G. García, J. P. Gutiérrez, and J. D. Moyano, "Mast : Modeling and analysis suite for real time applications," in *Real-Time Systems, 13th Euromicro Conference on, 2001*. IEEE, 2001, pp. 125–134.

[36] R. Urunuela, A.-M. Déplanche, and Y. Trinquet, "Storm a simulation tool for real-time multiprocessor scheduling evaluation," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*. IEEE, 2010, pp. 1–8.

[37] A. Díaz-Ramírez, D. K. Orduño, and P. Mejía-Alvarez, "A multiprocessor real-time scheduling simulation tool," in *Electrical Communications and Computers (CONIELECOMP), 2012 22nd International Conference on*. IEEE, 2012, pp. 157–161.

[38] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, and M. Qamhieh, "Yartiss : A tool to visualize, test, compare and evaluate real-time scheduling algorithms," in *WATERS 2012*. UPE LIGM ESIEE, 2012, pp. 21–26.

[39] M. Chéramy, P.-E. Hladik, and A.-M. Déplanche, "Simso : A simulation tool to evaluate real-time multiprocessor scheduling algorithms," in *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2014, pp. 6–p.

[40] P. Amey and B. Dobbing, "Static analysis of ravenscar programs," in *ACM SIGAda Ada Letters*, vol. 23, no. 4. ACM, 2003, pp. 58–64.

[41] M. De Sanctis, C. Trubiani, V. Cortellessa, A. Di Marco, and M. Flamminj, "A model-driven approach to catch performance antipatterns in adl specifications," *Information and Software Technology*, vol. 83, pp. 35–54, 2017.

[42] A. Motii, B. Hamid, A. Lanusse, and J.-M. Bruel, "Guiding the selection of security patterns for real-time systems," in *Engineering of Complex Computer Systems (ICECCS), 2016 21st International Conference on*. IEEE, 2016, pp. 155–164.

[43] G. Brau, J. Hugues, and N. Navet, "A contract-based approach to support goal-driven analysis," in *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*. IEEE, 2015, pp. 236–243.

[44] A. Magdich, Y. H. Kacem, A. Mahfoudhi, and M. Kerboeuf, "A uml/marte-based design pattern for semi-partitioned scheduling analysis," in *WETICE Conference (WETICE), 2014 IEEE 23rd International*. IEEE, 2014, pp. 300–305.

[45] P. Dissaux and B. Hall, "Merging and processing heterogeneous models," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[46] P. Dissaux and P. Farail, "Model verification : Return of experience," in *Proceedings ERTS conference*, 2014.