Analyse de systèmes temps-réels de sûreté et mitigation de leurs interférences temporelles

Jean Guyomarc'H

université
PARIS-SACLAY

Analyse de systèmes temps-réels de sûreté et mitigation de leurs interférences temporelles

Analysis of safety-critical real-time systems and mitigation of their timing interferences

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580, sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : informatique
Unité de recherche : Université Paris-Saclay, ENS Paris-Saclay, CNRS, SATIE, 91190, Gif-sur-Yvette, France
Référent : Faculté des sciences d'Orsay

Thèse présentée et soutenue à Gif-sur-Yvette, le 12 octobre 2021, par

Jean GUYOMARC'H

Composition du jury :

Isabelle PUAUT Professeure Université de Rennes 1, IRISA	Présidente
Claire PAGETTI Ingénieure de recherche (HDR) ONERA, DTIS	Rapporteur & Examinatrice
Christine ROCHANGE Professeure Université Toulouse III, IRIT	Rapporteur & Examinatrice
Claire MAÏZA Maître de conférences Université Grenoble Alpes, VERIMAG	Examinatrice

Direction de la thèse :

Stéphane ESPIÉ Directeur de recherche Université Gustave Eiffel, SATIE	Directeur de thèse
Alain MÉRIGOT Professeur Université Paris-Saclay, SATIE	Co-Directeur de thèse
Bastien VINCKE Maître de conférences Université Paris-Saclay, SATIE	Co-Encadrant
Emmanuel OHAYON Architecte logiciel Krono-Safe	Co-Encadrant Tuteur en entreprise

Thèse de doctorat
NNT : 2021UPASG070

Remerciements

L'erreur fondamentale d'attribution est un biais cognitif récurrent qui nous rend prompts à attribuer nos succès à notre seul mérite et à reléguer les raisons de nos échecs à des causes externes — la notion de mérite est elle-même largement discutée. Les remerciements qui suivent insistent sur ce constat : ces trois ans de thèse et leur aboutissement n'est pas que le fruit de dur labeur, mais aussi une succession de coïncidences, d'événements pseudo-aléatoires, de chance et de malchance, de rencontres clés, de traits d'esprit et d'idées malencontreuses. Je rends ainsi hommage à la pluralité des échanges et événements qui ont jalonné ma route, ainsi qu'à leurs acteurs et actrices.

* * *

Cette thèse, véritable serpent de mer de mon parcours professionnel, n'aurait sans doute pas pu voir le jour sans les contributions combinées de Vincent David, Alain Mérigot, Sylvain Cotard, Didier Roux et Nicolas Laforest. Elle ne serait peut-être pas arrivée à son terme sans l'investissement et le soutien indéfectible de Bastien Vincke et Emmanuel Ohayon dans la lourde charge de mon encadrement ; je les remercie tout particulièrement. Je remercie également Stéphane Espié pour avoir endossé le rôle de directeur de thèse dans la dernière ligne droite. Le personnel de l'école doctorale STIC mérite mes félicitations : confronté à mon parcours administratif parfois tumultueux, il a su trouver des solutions et compromis permettant de satisfaire chacun.

Bien que n'ayant pas obtenu de subventions CIFRE, la société Krono-Safe a tout de même pris en charge l'intégralité du financement de ces travaux. Je suis une fois de plus reconnaissant de la confiance qui m'a été accordée par la direction. Je remercie le laboratoire SATIE qui m'a accueilli pendant ces trois ans et a rempli sa fonction d'entité académique d'encadrement.

De nombreux collègues et ami·e·s ont contribué, explicitement ou à leur insu, à améliorer ces travaux, par de stimulantes discussions, de sévères, mais constructives revues, en amenant la contradiction, et ainsi rendant mes discours plus robustes, ou encore par un boost de motivation. Je remercie ainsi et sans ordre particulier : François G., Matthieu T., Bilal E.M., Amira M., Fabien S., Guillaume G., Guillaume P., Vincent G., Jean-Baptiste H., Jérémy A., Paul G., Andres Q., Charlotte D., Erwan C., Robin G., Hugo D., Olivier B., Jean-Sébastien G., Damien C., Pierre-Alain D., Alan H., Jean-Marc L., Sylvain G., Brice D., Adrien M., Élodie C., et bien d'autres.

* * *

Je remercie particulièrement Claire Pagetti et Christine Rochange pour avoir accepté de rapporter ma thèse et d'y avoir consacré leur temps et leur expertise, mais également Claire Maïza et Isabelle Puaut pour avoir accepté de faire partie de mon jury. C'est également pour moi l'occasion de remercier, de manière plus générale, les relectrices et les relecteurs de mes articles, qui ont permis d'enrichir considérablement mes travaux.

* * *

Sur une touche plus personnelle, je profite de cette page pour remercier mes parents, qui m'ont permis — entre autres — d'accéder à ce niveau d'éducation, ma sœur, ainsi que Christiane Cheneaux pour m'avoir aiguillé vers l'ingénierie. C'est également l'occasion d'exprimer une pensée émue pour Hélène Peyre, Avelino Pérez ainsi que Gisèle et Pierre Huet ; chacun·e à leur manière ont contribué à mon parcours. Une autre, particulière, est dédiée à Laurent George, qui a été témoin de ma volonté d'effectuer cette thèse, mais qui n'aura pu en voir l'aboutissement.

Table des matières

Liste des publications et contributions

Conférences internationales

[Guy+20] Jean Guyomarc'h, François Guerret, Bilal El Mejjati, Emmanuel Ohayon, Bastien Vincke et Alain Mérigot. « Non-Simultaneity as a Design Constraint ». In : *27th International Symposium on Temporal Representation and Reasoning (TIME)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020. DOI : 10.4230/LIPIcs.TIME.2020.13.

[GH20] Jean Guyomarc'h et Jean-Baptiste Hervé. « Static and Verifiable Memory Partitioning for Safety-Critical Systems ». In : *International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Coimbra, Portugal : IEEE, oct. 2020, p. 79-84. DOI : 10.1109/ISSREW51248.2020.00041.

Posters

[GMO19] Jean Guyomarc'h, Alain Mérigot et Emmanuel Ohayon. « Reducing the variability of execution times in safety-critical real-time systems (poster) ». In : *International School on Rewriting*. Paris, France, juill. 2019.

Articles (non publiés)

[Guy+20] Jean Guyomarc'h, François Guerret, Emmanuel Ohayon et Bastien Vincke. « Towards Trusted Timing Analysis of Safety-Critical Multi-Core COTS Systems ». Déc. 2020.

[GV21] Jean Guyomarc'h et Bastien Vincke. « Concolic Execution as an Analysis Technique to Leverage Timing Anomalies Mitigations ». Mai 2021.

Chapitre 1

Introduction

Sommaire

1.1 Contexte

Les systèmes temps-réels régissent une partie des applications auxquelles la population des pays dits « industrialisés » est habituée. Un sous-ensemble consiste à contrôler des appareils et dispositifs dont la défaillance est susceptible d'affecter de manière très négative leur environnement. Ainsi, les accidents aériens provoquent des morts par centaines et les accidents nucléaires peuvent avoir des conséquences humaines et environnementales catastrophiques. Ces systèmes « temps-réels de sûreté » sont la cible des travaux de cette thèse.

L'enjeu de la sûreté de ces systèmes étant primordiale, il est impératif que les différents composants logiciels qui les constituent respectent des contraintes temporelles strictes : chaque action critique doit terminer en un temps borné, déterminé pendant la conception du système. Afin de respecter cette propriété, les systèmes temps-réels de sûreté sont principalement développés autour de stratégies de provision de temps. Des fenêtres de temps peuvent ainsi être dédiées à certains calculs afin de leur garantir le temps nécessaire à leur exécution complète. Dans la pratique, ce sont les temps d'exécution dans le pire des cas, ou WCET (Worst-Case Execution Time), qui dictent la taille de ces fenêtres temporelles.

L'estimation de ces WCET est rendue complexe par la nature des composants micro-architecturaux qui constituent le matériel. Leur caractère spéculatif, cherchant à améliorer les performances moyennes des programmes, dégrade certaines propriétés

temporelles des temps d'exécution, notamment le WCET ainsi que sa prédictibilité. Afin de compenser les incertitudes dans l'estimation de ces WCET, les industriels surestiment considérablement la borne haute de leur provision de temps. Cette mesure, indispensable au respect des critères de sûreté de fonctionnement, conduit toutefois à un surdimensionnement systématique du système. Si ce surdimensionnement est trop important, il sous-utilise les capacités de calcul du matériel, affectant les coûts de production, la consommation d'énergie et le temps de calcul des algorithmes.

Cette thèse s'inscrit dans une foule de travaux cherchant à mettre en place des stratégies permettant de réduire la surestimation des WCET, sans toutefois compromettre la sûreté du système. Leur mise en œuvre vise à concevoir des systèmes temps-réels de sûreté mieux dimensionnés, tirant pleinement parti de leurs composants matériels tout en réduisant les sources d'interférences temporelles. Ces travaux de recherche sont financés par la société Krono-Safe [1], éditeur de logiciel système à destination première des constructeurs aéronautiques.

1.2 Contributions

Afin d'explorer le problème posé, nous avons identifié les principales sources de nuisances liées à l'estimation du WCET. Il nous semble que les cas monocœurs et multicœurs peuvent être traités séparément, dans une certaine mesure. En effet, la présence de plusieurs cœurs d'exécution simultanés contribue fortement à la dégradation des propriétés temporelles des temps d'exécution, en particulier quand ceux-ci mobilisent simultanément une même ressource matérielle partagée. D'autre part, une grande partie des interférences temporelles au sein d'un seul cœur d'exécution provient de composants tels que les caches. Notons toutefois l'existence de certains cas particuliers, comme les caches partagés (p. ex. de niveaux L2 et L3), qui rendent cette frontière poreuse.

L'utilisation des mémoires étant centrale dans l'implémentation des stratégies de mitigations et de nos expériences, nous avons développé une méthode de partitionnement spatiale en prévision de nos travaux. Tout en prenant en compte les contraintes strictes de protection mémoire habituellement inhérentes aux systèmes temps-réels de sûreté, notre méthode permet d'exprimer des stratégies de placement en mémoire avec une fine granularité. Elle garantit un partitionnement spatial statique, entièrement calculé hors-ligne, donc vérifiable par des outils tiers prévus pour des cadres certifiés. Nous avons documenté et explicité notre démarche par le biais d'un article publié à la conférence ISSRE 2020 (*Industry Track*) [GH20].

Les interférences temporelles causées par les accès simultanés à des ressources partagées peuvent être mitigées à l'exécution par des mécanismes dynamiques ou prévenues hors ligne lors de la phase de conception du système. Nous orientons nos recherches dans cette dernière voie pour favoriser le respect des propriétés de sûreté inhérentes aux systèmes temps-réels de sûreté. Nous exploitons un modèle de calcul existant (les automates contraints en temps) et nous y rajoutons une sémantique de simultanéité entre de multiples automates liés par une horloge logique commune. Cette nouvelle sémantique permet, par contraposée, d'exprimer la non-simultanéité, que nous promouvons en contrainte de conception. Une application pratique de ce

1. https://www.krono-safe.com/

concept consiste à exprimer des applications multicœurs dont les tâches constituantes se décomposent en fenêtres temporelles identifiées. Certaines d'entre elles peuvent accéder à une ressource matérielle partagée. Ces fenêtres temporelles peuvent appartenir à un ou plusieurs groupes d'exclusion, qui expriment des propriétés de sûreté vérifiables. Ainsi, si deux fenêtres temporelles appartenant à un même groupe d'exclusion sont susceptibles de se chevaucher dans le temps — c'est-à-dire de s'exécuter simultanément — alors l'application est sensible aux interférences temporelles, car la ressource partagée peut être accédée simultanément par plusieurs cœurs. Dans le cas contraire et sous réserve que les groupes d'exclusion renseignés par l'utilisateur soient corrects et complets, l'application multicœur ne peut partager de ressource matérielle durant son exécution. Les algorithmes de vérification de ces propriétés de sûreté exploitent un parallèle que nous avons remarqué entre automates contraints en temps et l'expression de langages unaires sur des automates acceptants. Ces contributions ont fait l'objet d'un article publié à la conférence TIME 2020 [Guy+20a].

Le principe de non-simultanéité exposé plus haut est en pratique complexe à déployer : l'utilisateur doit identifier les accès mémoires réalisés par son application afin d'écrire des groupes d'exclusion complets. Les contributions qui suivent permettent de simplifier, voire d'automatiser sa mise en œuvre. Nous avons ainsi développé une analyse sur cible basée sur un principe de co-exécution. Cette co-exécution fait intervenir un logiciel particulier ayant pour seul objectif de stresser une ou plusieurs ressources matérielles partagées. Il bénéficie de l'exclusivité des ressources de calcul d'un cœur processeur et nous permet de caractériser le matériel utilisé ainsi que le logiciel de l'utilisateur. L'avantage de cette analyse est double. D'une part, elle permet d'identifier expérimentalement les faiblesses et atouts du matériel quand celui-ci est soumis à un stress contrôlé. D'autre part, notre analyse permet d'associer des portions de l'application aux ressources matérielles auxquelles celles-ci accèdent. Si l'application se base sur le modèle de calcul des automates contraints en temps, cette analyse permet de limiter l'occurrence des accès à des fenêtres temporelles identifiées. Celles-ci peuvent alors être associées à des groupes d'exclusion afin d'utiliser le principe de non-simultanéité. Ces travaux ne sont pas encore publiés [Guy+20b].

Les fondements de notre dernière contribution ont été présentés dans un poster durant l'école d'été sur la réécriture de 2019 [GMO19]. Il s'agit d'une analyse statique exploitant les principes de l'exécution concolique et les solutionneurs SMT (Satifiability Modulo Theories) [2]. Une application compilée est ainsi interprétée hors ligne par un mécanisme qui émule chaque instruction rencontrée, et qui grâce à un processeur virtuel, trace l'ensemble des accès mémoires réalisables. Naturellement, dans des conditions opérationnelles, le logiciel manipule des variables dont le contenu est inconnu à la compilation (p. ex. *via* des périphériques). Nous utilisons le principe de l'exécution symbolique afin d'explorer tous les chemins de code atteignables quand le programme manipule des données « inconnues » (symboliques). Cette technique nous permet d'explorer les effets du code applicatif compilé et d'en extraire un modèle des accès mémoires réalisés. Les données collectées sont exploitables par des algorithmes de placement mémoire ou de partitionnement des caches afin de générer une nouvelle application dont le profil mémoire peut exhiber une amélioration sensible des propriétés temporelles des temps d'exécution. Nous tirons pleinement parti des contraintes inhérentes au développement

2. *SMT-solver* en anglais.

des logiciels temps-réels de sûreté afin d'éliminer les cas pouvant rendre notre technique inopérante — comme les boucles non bornées ou le code automodifiant. Ces travaux ne sont pas encore publiés [GV21].

1.3 Méthodologie

La méthodologie globale proposée dans ces travaux, qui articule nos contributions précédemment exposées, est récapitulée en figure 1.1, et s'inscrit dans un contexte industriel.

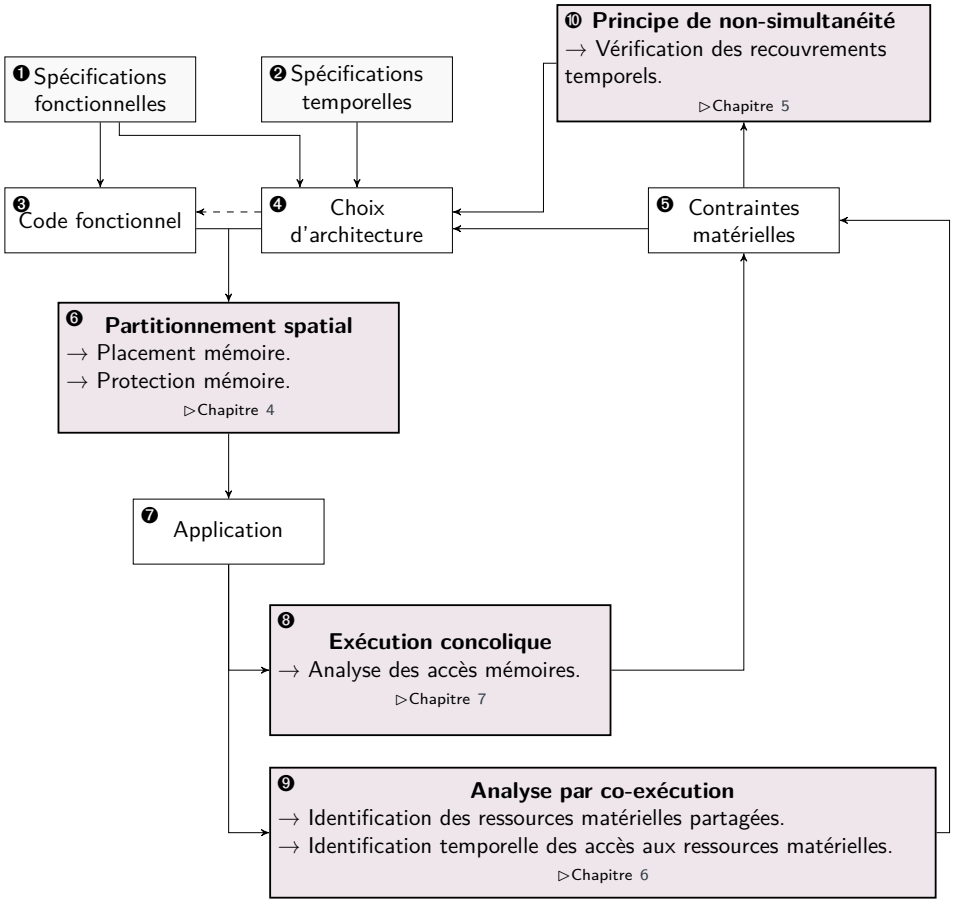


Figure 1.1 – Méthodologie globale proposée.

Les industriels se reposent sur des spécifications fontionnelles ❶ et temporelles ❷, qui leur sont dictées. À partir de ces spécifications et des contraintes matérielles ❺ (p. ex. le nombre, la taille et l'organisation mémoires disponibles), les industriels doivent établir des choix d'architecture ❹ (p. ex. une découpe en tâches, une stratégie d'utilisation des cœurs et des mémoires). Ces choix pourront orienter l'écriture du code

fonctionnel ❸, qui découle directement des spécifications fonctionnelles.

Une application ❼ est le produit logiciel destiné à être implanté sur une cible matérielle. Elle est générée à partir de code fonctionnel et des choix d'architecture. Nos travaux sur le partitionnement spatial ❻ contribuent à cette phase de production logicielle, en offrant des capacités de placement et de protection mémoire. Une fois l'application générée, nos travaux déclinent deux phases distinctes d'analyse, qui ont pour but commun de raffiner les contraintes matérielles. À terme, ces informations supplémentaires peuvent avoir une rétroaction sur les choix d'architecture, et ce afin de générer de nouvelles applications, exhibant de *meilleures* propriétés temporelles par rapport à de précédentes itérations.

1. Durant l'intégration logicielle, une exécution concolique ❽ vise à analyser les accès mémoires réalisés par le logiciel. Les données ainsi collectées permettent d'obtenir des informations précises quant à l'utilisation de la mémoire par les tâches, permettant d'affiner les choix d'architecture quant aux stratégies de placement mémoire.
2. Au moment de l'intégration matérielle, une analyse par co-exécution ❾ permet d'identifier les ressources matérielles partagées, ainsi que des traces d'exécution clairement délimitées dans le temps, qui permettent de déterminer les portions des tâches accédant à ces ressources. Les informations collectées permettent d'identifier les enjeux des accès simultanés aux ressources matérielles partagées, dans un cadre multicœurs.

L'avantage du principe de non-simultanéité ❿ que nous avons développé est double. Il peut d'une part être utilisé en amont, lors de la phase de conception, pour mettre en œuvre une architecture logicielle permettant de prévenir les accès simultanés aux ressources matérielles partagées. Il peut d'autre part être utilisé en aval, suite aux analyses par exécution concolique ou par co-exécution : les données issues de ces analyses permettent à terme de revoir l'architecture logicielle, si celle-ci s'avère trop sensible aux interférences temporelles causées par une utilisation suboptimale des ressources matérielles en multicœur.

Ainsi, cette méthodologie consiste à générer des applications de sorte qu'elles soient conformes de leurs spécifications de haut niveau, tout en affinant les choix d'architecture permettant une meilleure prise en compte des contraintes matérielles dans cette réalisation.

1.4 Plan

Le détail de nos travaux s'articule autour de quatre parties. Par commodité d'écriture, nous utilisons de nombreux termes et acronymes. Les principaux sont listés dans le glossaire, disponible en page 149.

I. La première partie rappelle la problématique générale, le cadre dans lequel s'inscrivent nos travaux et détaille une méthode de partitionnement spatial, prérequis à la mise en place des autres parties. Elle contient trois chapitres.
   — Le chapitre 2 (page 9) pose les bases du sujet en rappelant les principales thématiques qui gravitent autour des systèmes temps-réels de sûreté. Les

contraintes de sûreté de fonctionnement opérationnelles et liées au processus industriel auxquelles sont soumis nos travaux y sont décrites. Nous introduisons les différents mécanismes de partitionnement temporel propres aux systèmes temps-réels afin d'introduire le problème du WCET autour duquel s'articule notre sujet.

— Nos travaux s'intègrent à une technologie industrielle : ASTERIOS®. Elle est exposée au chapitre 3 (page 23) afin que nos contributions soient clairement identifiées par rapport au socle technique et scientifique existant.

— La première partie se conclut par le chapitre 4 (page 35), qui détaille une méthode de partitionnement mémoire statique que nous avons développée. Particulièrement adaptée aux systèmes de sûreté de fonctionnement, elle permet un placement du code et des données en mémoire avec une fine granularité, sans compromettre les exigences de protection mémoire. Cette capacité de placement est un prérequis à la mise en œuvre des stratégies développées dans les parties qui suivent.

II. La deuxième partie se focalise sur la problématique de l'utilisation des plateformes matérielles multicœurs pour concevoir des systèmes temps-réels de sûreté. Nous détaillons deux contributions majeures de nos travaux qui sont le principe de non-simultanéité et l'analyse par co-exécution.

— Le chapitre 5 (page 59) détaille le principe de non-simultanéité. Il s'appuie sur des automates temporels permettant de formaliser des fenêtres d'exécution bornées en temps. Ce principe permet de détecter, par une vérification hors ligne, les superpositions temporelles de fenêtres d'exécution d'intérêt. Une application directe de ce principe permet de concevoir des applications multicœurs garantissant l'absence d'accès simultanés à des ressources matérielles partagées.

— L'application du principe de non-simultanéité peut être facilitée par une méthode d'analyse basée sur un principe de co-exécution, détaillée au chapitre 6 (page 81). Cette méthode exploite les co-runners, structures d'exécution permettant de caractériser le matériel cible ainsi que le logiciel s'y exécutant. L'analyse qui en découle permet de fournir des éléments visant à contribuer à la confiance dans l'analyse temporelle des systèmes temps-réels de sûreté sur des plateformes multicœurs.

III. La troisième partie cherche à mettre en place de manière systématique des mitigations aux interférences temporelles causées par une utilisation sous-optimale des mémoires matérielles. Elle est constituée d'un chapitre unique : le chapitre 7 (page 109), qui détaille une nouvelle méthode d'analyse des systèmes temps-réels de sûreté. Grâce à un mécanisme d'exécution « concolique » particulier, nous extrayons un modèle précis des accès mémoires effectués par une application. Cette connaissance permet de mettre en œuvre des stratégies de placement mémoire exploitant les systèmes de caches ainsi que les différentes mémoires matérielles présentes.

IV. La dernière partie conclut cette thèse par son chapitre 8 (page 143). Nous y discutons des innovations amenées ainsi que de leurs limites, tout en ouvrant sur de futurs travaux de recherche.

I

Première partie

Problématique générale

Chapitre 2

Systèmes temps-réels de sûreté

Sommaire

2.1 Contexte général et positionnement

Les systèmes temps-réels sont décrits avec profusion dans la littérature scientifique. Les sections qui suivent permettent de restreindre le cadre d'application de nos travaux, et ainsi de circonscrire notre périmètre d'exploration.

2.1.1 Contraintes de sûreté de fonctionnement

Un *système cyber-physique* est défini comme l'intégration de mécanismes calculatoires à des processus interagissant avec un environnement physique [Lee08]. On

peut également les qualifier de systèmes *réactifs*, dans le sens où ils doivent répondre à des commandes extérieures [Loh+19]. Ceux-ci sont parfois contraints par des exigences de cadence et de latence, imposant l'exécution de calculs donnés entre des bornes temporelles ; on parle alors plus généralement de *systèmes temps-réels*. Quand le non-respect de ces contraintes temporelles contribue à une défaillance du système et qu'une défaillance est susceptible de mener à des conséquences jugées catastrophiques (c.-à-d. : perte de vies humaines, désastre écologique, *etc.*), ces systèmes sont qualifiés de *systèmes temps-réels de sûreté* [Kop11].

De nombreuses industries spécialisées, comme l'avionique, le ferroviaire, l'automobile ou le nucléaire civil ont comme mission de concevoir et développer de tels systèmes. Ceux-ci sont régis par des normes internationales strictes, qui veillent au bon respect des contraintes de sûreté de fonctionnement, comme l'ISO 26262 [ISO18], l'IEC 60880 [IEC06] ou la DO178-C [RTC11a].

Positionnement 1 (Domaine avionique)**.** *Nos travaux se concentrent particulièrement sur l'étude de systèmes temps-réels de sûreté dans le domaine avionique, c'est-à-dire devant se conformer aux critères de certification les plus exigeants de la DO-178C.*

2.1.2 Contraintes opérationnelles

Les systèmes temps-réels de sûreté existent depuis plus de cinquante ans : un des plus marquants de l'Histoire étant peut-être le système de guidage d'Apollo 11 [MB68]. Depuis, les générations de systèmes aéroportés se sont succédé, et des standards de développement comme l'IMA (Integrated Modular Avionics) ou l'ARINC-653 [Air15] se sont imposés, indices de la maturité de tels systèmes.

Toutefois, les besoins du marché de l'avionique évoluent. Ainsi la taille des programmes embarqués sur les systèmes avioniques croît de manière exponentielle à chaque génération : les nouvelles requièrent pour cela toujours davantage de puissance de calcul que les précédentes [War+13].

De plus, les composants électroniques vieillissent jusqu'à devenir défectueux, ce qui signifie que le matériel des générations précédentes doit être remplacé. L'évolution continue du matériel conduit les fondeurs à privilégier de nouvelles générations de matériel, au détriment des plus anciennes, qui deviennent obsolètes et ne sont plus manufacturées, le coût de maintenance des installations surpassant les gains de vente. Le coût de développement du matériel (en particulier des processeurs) étant considérable (plusieurs centaines de millions de dollars [Bur90 ; Uhl19]), les fondeurs privilégient les marchés de masse plutôt que de cibler des domaines industriels de niche comme l'avionique. Pour ces mêmes raisons économiques, ces niches préfèrent l'achat de composants sur étagère, ou COTS (Component Off-The-Shelf), à l'investissement dans des solutions matérielles particulières, suivant la tendance du « *Buy, don't build* » [LRS+94 ; Bak02].

Positionnement 2 (Composants matériels sur étagère)**.** *Nos travaux se limitent à l'étude des systèmes temps-réels de sûreté utilisant du matériel industriel sur étagère, c'est-à-dire fourni par une tierce partie, sans possibilité d'intervenir dans sa conception ou de le modifier.*

2.1.3 Contraintes liées au processus industriel

Les processus de développement des logiciels temps-réels de sûreté avioniques sont régis par la norme DO-178C. Ceux-ci sont généralement implémentés *via* des méthodes de développement en « V » ou en spirale [Boe88]. Le constructeur détient la feuille de route qui découpe le système final en de multiples composants, dont des entités parfois multiples sont responsables de leur livraison. Les sous-systèmes critiques (c'est-à-dire hébergeant des applications temps-réelles de sûreté) sont découpés en de multiples briques logicielles, dont la cohérence est assurée par une fonction d'*intégrateur*. Ces fonctions logicielles sont livrées contractuellement ; le code source original n'est souvent pas disponible et encore moins modifiable. Il incombe à l'intégrateur de s'assurer du respect des exigences temporelles de cet ensemble. Dans cette optique, l'intégrateur réalise en amont une conception temporelle de son système, qui peut être amendée en aval, au fil de l'intégration.

Positionnement 3 (Intégration à la chaîne de production logicielle)**.** *Les contraintes temporelles étant particulièrement sensibles au niveau de l'intégration, c'est à ce niveau que nous orientons nos travaux. Nous considérons ainsi que le code fonctionnel dit « métier » est toujours écrit par un tiers et que nous ne pouvons intervenir directement sur son développement.*

2.2 Concepts de partitionnement temporel

Comme évoqué en section 2.1, les systèmes temps-réels de sûreté doivent interagir avec un environnement physique, ils doivent ainsi traiter des événements extérieurs, comme le déclenchement d'un minuteur, l'activation d'un capteur, *etc.* Différents paradigmes de gestion des événements permettent de les concilier avec des contraintes temps-réelles.

2.2.1 Paradigme « event-triggered »

La plupart des systèmes temps-réels non soumis à des contraintes de sûreté de fonctionnement privilégient une approche dite *event-triggered*, dans laquelle les événements extérieurs sont immédiatement pris en charge. Celle-ci peut simplement se traduire par la levée d'une interruption matérielle et son acquittement. Les calculs peuvent être traités de manière différée. La figure 2.1 illustre ce principe :

— une tâche (notée α) est élue par l'ordonnanceur ;
— α est préemptée par une interruption levée suite à un événement matériel ;
— la routine d'interruption, ou ISR (Interrupt Service Routine), prend en charge cet événement, qui donne lieu à un traitement ultérieur (tâche β) ;
— la main est rendue à α, qui complète, laissant à son tour la main à l'ordonnanceur ;
— ce dernier permet l'élection de la tâche β.

Notons que d'autres stratégies que celle-ci sont possibles. Par exemple, l'ISR aurait pu se conclure par un appel à l'ordonnanceur permettant l'élection immédiate de β.

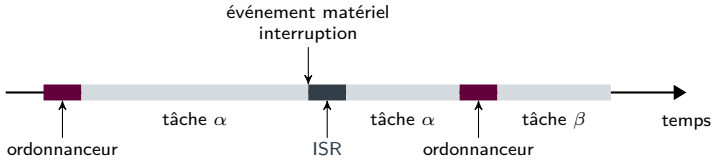


Figure 2.1 – Exemple d'enchaînement de traitements avec paradigme *event-triggered*. La survenue d'une interruption cause l'activation d'une ISR préemptant la tâche α en cours. À l'issue de l'ISR, α est reprise jusqu'à sa complétion. L'ordonnanceur permet ensuite l'élection d'une nouvelle tâche β, déclenchant des traitements différés en réponse à l'interruption matérielle.

Cette approche centrée sur la réception d'événements, couplée avec une politique d'ordonnancement adéquate, comme privilégier l'élection des tâches avec la plus proche échéance, c.-à-d. EDF (Earliest Deadline First), permet une implémentation réactive et dynamique. Celle-ci permet d'exploiter le potentiel calculatoire d'un processeur qui est uniquement sollicité en présence d'événements.

Toutefois, comme le flot de contrôle de l'exécution est piloté par des événements extérieurs dont les occurrences ne peuvent être prédites, il est ardu de prouver que de tels systèmes soient sûrs. C'est-à-dire que le bon fonctionnement du système n'est pas assuré *quels que soient* les événements reçus par le système. Ainsi, le système de guidage d'Apollo 11 évoqué en section 2.1.2, basé sur un système *event-triggered*, fut soumis à un flux d'interruption non contrôlé lors de la procédure d'alunissage. Les fonctions nominales critiques se sont retrouvées privées de temps d'exécution pour s'effectuer, mettant la mission et la survie des astronautes en péril [Tom88]. Aussi, les solutions basées sur le paradigme *event-triggered* sont-elles rarement utilisées dans les systèmes critiques avioniques. On lui préfère un paradigme alternatif apportant des garanties de sûreté.

2.2.2 Paradigme « time-triggered »

Le paradigme *time-triggered* s'oppose à l'*event-triggered* vu en section 2.2.1 en ne considérant qu'un seul type d'événement : l'écoulement du temps. Au vu des considérations physiques des systèmes étudiés, on peut raisonnablement considérer que le temps s'écoule de manière monotone et uniforme au sein d'un système donné.

Dans ces travaux, nous distinguons le temps *logique* du temps *physique*. Le temps logique est produit par une horloge de Lamport, qui garantit un ordre total entre tous les événements (ou *tics*) produits par cette horloge, chaque tic produisant une estampille unique [Lam78]. Ce sont ces horloges logiques qui cadencent l'écoulement des événements temporels. Si le temps logique peut être uniquement conceptuel, le temps physique est toujours exprimé en unités de temps (c.-à-d. en secondes, selon le système international des unités) et capture le temps qui s'écoule dans le monde physique. Si leurs unités de temps ne sont pas nécessairement les mêmes, dans la pratique celles-ci sont fortement corrélées. D'autres travaux, comme le modèle des réacteurs [Loh+19], différencient également le temps logique du temps physique.

Ce paradigme basé sur l'écoulement du temps est illustré par la figure 2.2, qui

présente un ordonnancement *time-triggered* alternatif au paradigme *event-triggered* de la figure 2.1 :

— à partir du tic logique λ_i, l'ordonnanceur s'exécute, conduisant à l'élection de la tâche α ;

— un événement matériel survient durant l'exécution de α, mais comme celui-ci n'est pas considéré comme tel, sa survenue n'a aucun impact sur l'exécution du système.

— α ayant complété avant la survenue du tic logique λ_j, l'ordonnanceur permet alors l'élection de la tâche β. Celle-ci peut consulter l'état du matériel, et ainsi prendre en charge l'événement matériel précédemment levé.

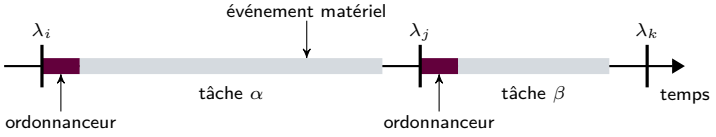


Figure 2.2 – Exemple d'enchaînement de traitements avec paradigme *time-triggered*, adaptation de l'exemple *event-triggered* de la figure 2.1. Chaque tic λ_i, λ_j, λ_k d'une horloge logique réveille l'ordonnanceur qui exécute une tâche dans un intervalle de temps donné. La survenue d'un événement matériel n'est ainsi traitée qu'à partir du prochain tic logique, et n'affecte pas l'exécution courante.

2.2.3 Multiplexage temporel statique

Le paradigme *time-triggered* décrit en section 2.2.2, associé à une politique d'ordonnancement statique (c.-à-d. calculée hors ligne) permet l'expression du TDM (Time-Division Multiplexing), qui est une variante particulièrement utilisée dans les systèmes avioniques, s'appuyant sur une découpe du temps en fenêtres temporelles de tailles fixes [Air15].

Les systèmes de sûreté préfèrent l'utilisation de politiques statiques, nécessitant moins de calculs à l'exécution (qu'il faudrait garantir valides), s'appuyant sur des résultats prévus hors-ligne. Si la vérification de ces résultats est plus aisée, le système est rigide par conception et ne peut évoluer à l'exécution. Cela contraint notamment à disposer d'un nombre fixé de tâches, et d'associer chaque tâche à un cœur processeur en particulier (c.-à-d. aucune migration n'est autorisée). Cette contrainte permet toutefois d'effectuer une analyse d'ordonnançabilité en amont, garantissant le respect des contraintes temporelles des tâches. De plus, elles permettent de s'affranchir d'anomalies engendrées par les politiques dynamiques [Pha+18].

Les politiques d'ordonnancement statiques se basent la plupart du temps sur une stratégie de provision de temps, mettant ainsi en place un mécanisme de partitionnement temporel, offrant de fortes garanties de sûreté. Sous réserve que le système soit correctement dimensionné [1], chaque tâche peut disposer du temps physique nécessaire à son

1. Le concept de « dimensionnement correct » est ici volontairement laissé flou ; il est détaillé en section 2.3.

exécution, comme illustré en figure 2.3. Cette technique conduit à une sous-utilisation du processeur [Heb+18]. Toutefois, comme les considérations de sûreté dominent, cette approche est privilégiée pour les systèmes temps-réels dont la sûreté est critique [WH09].

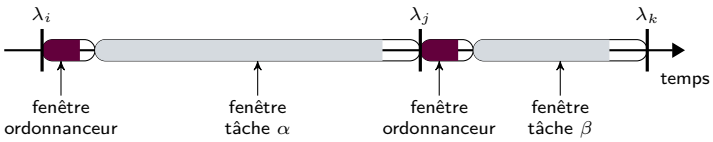


Figure 2.3 – Exemple d'enchaînement de multiplexage temporel statique. L'ordonnanceur et chaque tâche sont associés à des fenêtres temporelles précises, au-delà desquelles ils ne peuvent s'exécuter. Ici, le temps physique consommé dans chaque fenêtre est toujours strictement inférieur à la taille de la fenêtre.

Positionnement 4 (Partitionnement temporel statique)**.** *Nos travaux étudient uniquement des systèmes se basant sur des stratégies de multiplexage temporel statique. Nous laissons ainsi de côté les approches event-triggered et les politiques d'ordonnancement dynamiques.*

2.3 Problématique de provision de temps d'exécution

La section 2.2.3 conclut que les stratégies de partitionnement temporel sont à privilégier dans la conception de systèmes temps-réels de sûreté. Nous allons maintenant nous intéresser aux mécanismes de provision de temps permettant le dimensionnement du système, et aux problèmes associés.

2.3.1 WCET : temps d'exécution dans le pire cas

L'exécution du logiciel consomme du temps physique, qu'on appelle *temps d'exécution*. Afin de dimensionner un système basé sur un mécanisme de partitionnement de temps, comme vu en section 2.2.3, il est nécessaire d'évaluer ces temps, pour chaque tâche du système. Pour s'assurer du respect systématique des contraintes de temps du système, on cherche en particulier à déterminer les temps d'exécution dans le pire cas : on parle de WCET.

La détermination des WCET du logiciel permet la génération hors-ligne d'un plan d'ordonnancement statique, dans lequel figure la provision du temps nécessaire à l'exécution de toutes les tâches du système. On trouve de multiples exemples académiques ou industriels de cette pratique [Bon+12 ; Bec+16 ; ACD10 ; Air15]. Toutefois, les propriétés de sûreté apportées par cette provision de temps, indexée sur le WCET, ne sont valides que si l'estimation du WCET est *sound*[2]. La détermination de ces grandeurs est donc de première importance, car garante du bon fonctionnement des systèmes temps-réels de sûreté.

2. La propriété de *soundness* (en anglais) porte les notions d'intégrité d'un résultat et du fait qu'il soit rigoureusement correct. Nous préférons l'utilisation du terme anglais, car il traduit une idée formelle plus précise que ses possibles traductions françaises.

On retrouve trois grandes propriétés qui y sont fortement corrélées : la composition des temps [3] [HJR16], la décomposition en tâches [4] [PKP09] tout en augmentant la dispersion des temps d'exécution [NP12] [5] Ces trois propriétés sont définies comme suit [Dav20] [6].

Définition 1 (Timing compositionality)**.** *Les propriétés temporelles d'intérêt peuvent être déterminées comme une décomposition de constituants élémentaires. La vérification de cette propriété permet, par exemple, de considérer que le WCET d'une fonction en appelant deux autres est la somme des WCET de ces deux fonctions.*

Définition 2 (Timing composability)**.** *Les propriétés temporelles obtenues de l'analyse d'une tâche isolée sont transposables à un système composé de plusieurs tâches. Par exemple, le WCET d'une fonction appelée par une tâche α reste identique si une autre tâche β est ordonnancée conjointement à α.*

Définition 3 (Timing predictability)**.** *Cette propriété traduit la dispersion statistique des temps d'exécution observés. Cela peut être vu comme un ratio entre les meilleurs et pire temps.*

2.3.2 Méthodes d'analyse du WCET

Le WCET est une grandeur théorique : c'est une borne supérieure des temps d'exécution qui ne peut être atteinte que lors d'une conjonction bien précise d'événements. Une manière possible visant à déterminer le WCET d'une tâche, est d'exécuter une batterie de tests couvrants sur cette tâche, de collecter les temps d'exécution mesurés, et d'en déterminer la valeur maximale. Cette approche empirique permet d'observer les véritables temps de l'exécution du logiciel sur le matériel, puisqu'il s'agit d'observations expérimentales en conditions réelles. Toutefois, ces mesures n'apportent aucune garantie quant à la capture du « véritable » WCET. En effet, peut-être existe-t-il un état physique non rencontré pendant la prise de mesures, conduisant à des temps d'exécution supérieurs à ceux mesurés. Aussi, l'approche empirique d'estimation du WCET basée sur les mesures n'est-elle pas *sound* : il existe toujours une incertitude quant à la borne supérieure déterminée par cette méthode [Wil+08].

Une méthode alternative à celle basée sur les mesures consiste à construire un modèle du matériel, basé sur une connaissance rigoureuse de ses différents ensembles : caches, pipelines, mécanismes d'interconnexion, cohérence des caches, *etc.* En se basant principalement sur l'interprétation abstraite [CC77], de nombreux outils, comme aiT [HF04], Heptane [HRP17], OTAWA [Bal+10] ou encore SWEET [Lis14], pour n'en nommer que quelques-uns, visent ainsi à déterminer un WCET *sound* grâce à une analyse statique poussée du code. Des analyses sémantiques poussées permettent d'en affiner les résultats [Mai+17]. Sous réserve que le modèle du matériel sous-jacent soit exact, cette approche permet ainsi de fixer une borne garantie strictement supérieure aux temps d'exécution. De plus, comme elle ne nécessite que l'analyse du logiciel, celle-ci peut être utilisée en amont dans le cycle de production industriel, avant même les tests

3. Propriété *timing compositionality*.
4. Propriété *timing composable*.
5. Propriété *timing predictability*.
6. Nous utilisons les termes anglais car plus précis que leur traduction française.

d'intégration sur matériel, ce qui présente en avantage pratique important [Wil+08]. Cette distinction entre temps d'exécution déterminés par la mesure et par l'analyse est récapitulée en figure 2.4.

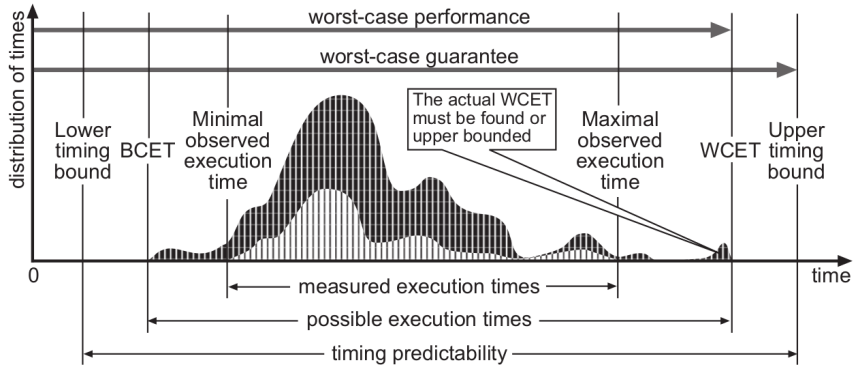


Figure 2.4 – Récapitulatif des principales propriétés d'intérêt des temps d'exécution (figure de WILHELM et al. [Wil+08]).

Toutefois, aussi séduisante que soit la méthode d'analyse statique (facile à déployer, plus en amont dans le cycle de développement, et avec des garanties de sûreté), celle-ci peut, en pratique, être limitée par l'opacité du matériel, qui reste une propriété intellectuelle des fabricants. Aussi, les modèles du matériel sont-ils parfois imparfaits, et pour préserver la propriété de *soundness*, des compromis dégradent souvent la propriété de *tightness*[7]. Cela se traduit par une surestimation systématique des WCET obtenus, qui varie en fonction des outils et de leurs cas d'usage.

Les approches dites hybrides combinent analyse statique et prise de mesures [Ber+07]. Cette approche permet d'éviter les limitations des approximations des modèles, qui sont d'autant moins précis que le matériel est complexe, tout en tirant parti des observations empiriques décrivant le réel. Des outils comme TimeWeaver [Käs+19] ou RapiTime [BMB10] proposent aux industriels une alternative aux méthodes précédemment évoquées. Celles-ci peuvent être combinées avec des approches probabilistes (pWCET) [Cuc+12 ; War+13 ; Abe+14] que nous ne détaillons pas ici.

2.3.3 Interférences temporelles

Comme évoqué en section 2.1.2, les fabricants de COTS matériels visent un public bien plus large que celui des systèmes temps-réels de sûreté, et privilégient ainsi la garantie de performances en moyenne. La maximisation des performances moyennes du matériel est rendue possible par l'intégration d'unités micro-architecturales, qui présentent l'inconvénient de dégrader de manière peu ou non prédictible les pires temps d'exécution [KP08]. Dans ces travaux, on qualifie les sources de ces dégradations d'*interférences temporelles*.

7. On traduit la *tightness*, dans le cadre du WCET, comme l'écart entre la borne supérieure des temps d'exécution pouvant survenir et la borne supérieure obtenue par analyse statique.

Définition 4 (Interférence temporelle)**.** *On appelle* interférence temporelle *toute perturbation par un effet de bord du matériel des propriétés temporelles vues en section 2.3.1. Ce terme englobant concentre les artefacts liés à l'implémentation du matériel comme l'utilisation que le logiciel en fait.*

Afin de clarifier cette définition, nous listons plus bas les principales sources de perturbation des propriétés temporelles. Nous en distinguons trois :

1. les mémoires caches ;
2. les accès simultanés aux ressources matérielles partagées ;
3. les anomalies temporelles.

Mémoires caches

L'utilisation de mémoires caches [Smi82] est une source bien connue et très largement documentée d'interférences temporelles [Kir89 ; Wil+08 ; Kos+16]. Si les mémoires caches offrent un potentiel de réduction des latences d'accès aux mémoires principales, elles sont assujetties aux principes de localités spatiales et temporelles des accès mémoires ainsi qu'aux problématiques de cohérence de leur contenu. Un cache effectuant un accès mémoire doit d'abord s'assurer que la plage d'adresses consultées réside déjà en cache et vérifier sa cohérence (c.-à-d. que la mémoire n'a pas été modifiée depuis un autre cœur processeur). Dans le cas contraire, le cache doit invalider tout ou partie de son contenu et effectuer l'accès mémoire demandé. Ce surcoût d'opérations est susceptible de dégrader significativement certaines propriétés temporelles, comme le WCET. À l'inverse, un accès mémoire valide vis-à-vis du cache permet d'éviter de solliciter la mémoire et les bus afférents. La combinaison de ces deux cas influe sur la *timing predictability* : les meilleurs temps d'exécution sont améliorés (plus petits) tandis que les pires temps d'exécution peuvent être détériorés (plus grands).

Un cœur processeur dispose souvent de deux caches séparés : l'un est destiné à stocker uniquement des données tandis que l'autre ne stocke que des instructions. Des architectures plus avancées peuvent disposer de caches partagés entre plusieurs cœurs, qui sont souvent unifiés. Ainsi, plus une architecture est complexe, plus la hiérarchie mémoire est profonde : un cœur accédant à une mémoire consulte d'abord un cache de premier niveau, puis celui de deuxième niveau *via* un bus partagé, et ainsi de suite jusqu'à accéder à la mémoire physique ciblée. L'utilisation de plusieurs tâches sur un même cœur — qui est une approche hégémonique dans les systèmes de sûreté que nous étudions — complexifie encore la gestion des caches. Les caches partagés, même si peu utilisés dans les systèmes de sûreté industriels, rajoutent un niveau de difficulté supplémentaire.

Une utilisation efficace des caches en ce qui concerne les propriétés temporelles comme le WCET est ardue. Son principal obstacle réside dans l'adéquation entre l'algorithme et l'architecture matérielle : l'intégration du logiciel à un matériel donné influe de manière significative sur ses propriétés temporelles et complexifient d'autant leur analyse.

Accès simultanés aux ressources matérielles partagées

Le multicœur amène une nouvelle couche de complexité qui s'ajoute aux difficultés d'analyse en monocœur : la présence d'unités d'exécution concurrentes rend possible les accès simultanés à des ressources matérielles partagées. Quand des fils d'exécution implantés sur des cœurs différents accèdent simultanément à une ressource partagée, le matériel doit arbitrer ces accès concurrents afin que chaque cœur puisse effectuer sa requête indépendamment. Cela a pour effet de congestionner les bus d'accès à ces ressources, augmentant notamment les latences des accès mémoires, ce qui mécaniquement dégrade le WCET des tâches.

Il est généralement estimé que les techniques d'analyse du WCET développées pour le monocœur, appliquées telles quelles à un cadre multicœur, conduisent à des valeurs de WCET évoluant linéairement en fonction du nombre de cœurs activés [Pel+10 ; Bin+14a]. Aussi, cette source d'interférences temporelles est-elle particulièrement étudiée [Weg17 ; Axe+14 ; Mai+19 ; Ves+12], et la certification de systèmes temps-réels de sûreté sur des architectures matérielles multicœurs sur étagère reste un problème industriel ouvert.

Les autorités de certification avionique ont publié un document de positionnement quant aux objectifs à atteindre : le CAST-32A [Cer16]. Depuis sa publication, de nombreux travaux cherchent spécifiquement à répondre aux exigences fixées par ce document [Agi+17 ; Bie+18 ; Bon+18 ; VE20 ; RTP18 ; VT20 ; BPS19]. Plus généralement, la recherche des *canaux d'interférences* et la mitigation des interférences temporelles engendrées par certains profils d'utilisation font partie des points d'intérêt actuels de la recherche sur les systèmes temps-réels.

Anomalies temporelles

Les anomalies temporelles consistent en un sous-ensemble de ce que nous nommons « interférences temporelles » — qui est plus englobant. Il s'agit d'un concept subtil et mouvant, dont la définition exacte est ardue [Rei+06]. Elles résultent d'interactions complexes au sein des processeurs et de leur opacité au regard des abstractions servant à présenter le comportement de ces processeurs. On peut considérer qu'elles découlent de l'implémentation du matériel et de la manière dont sont ordonnancées les *instructions* par le processeur [LS99a]. Il convient de rappeler qu'il ne s'agit pas d'un ordonnancement logiciel, mais bien de la manière dont un processeur traite un flot d'instruction de manière non-ordonnée (*out-of-order*), quoique cela ne soit pas toujours le cas.

Les principales sources de ces anomalies temporelles, identifiées dans la littérature, sont les pipelines [The04], prédicteurs de branchements [BP05] et les TLB (Translation Lookaside Buffer) [Ish+13]. Notons que les défauts et succès de cache ne sont pas considérés comme étant des anomalies temporelles. Toutefois, ces anomalies peuvent contribuer aux interférences engendrées par lesdits caches.

2.4 Réduction des interférences temporelles

Les multiples sources d'interférences temporelles listées en section 2.3.3 peuvent dépeindre un tableau bien sombre quant à l'estimation fiable d'un WCET. Toutefois, les problèmes posés sont surmontés dans la pratique — les systèmes temps-réels de

sûreté qui nous entourent en témoignent. Nous développons ici une liste non exhaustive de stratégies utilisées pour réduire ou prévenir les principales sources interférences temporelles.

2.4.1 Utilisation de matériel dédié

Une branche de la recherche en systèmes temps-réels s'oriente dans la conception de matériel dédié aux applications sensibles au WCET, c'est-à-dire des architectures matérielles avec des priorités temporelles prédictibles [RUS14]. On peut ainsi mentionner les machines PRET [EL07 ; LRZ17], ou les travaux du projet MERASA [Ung+10 ; Ung+16]. Leur but est de concevoir du matériel robuste aux anomalies temporelles — et aux interférences temporelles de manière plus large — qui permet d'héberger du logiciel spécialisé, compilé spécifiquement pour ces architectures. Si l'avènement de telles solutions pourrait simplifier la problématique du WCET, ces approches n'ont pas encore atteint une maturité industrielle suffisante. Aussi, et en vertu du positionnement 2, nous ne les étudions pas davantage dans ces travaux.

2.4.2 Contrôle des caches

Comme évoqué précédemment, les caches sont particulièrement difficiles à maîtriser dans le cadre d'applications industrielles, dont le développement fait souvent intervenir de multiples sous-traitants, chacun en charge de briques logicielles particulières. Sans cohérence d'ensemble lors du développement, la conception d'applications permettant une bonne exploitation des caches semble ardue. En effet, c'est principalement la composition des tâches constituant l'application qui mettent les caches en défaut.

Une technique triviale pour éviter les désagréments des caches consiste à les désactiver complètement. Certes peu élégante, cette solution permet d'éviter la dégradation de certaines propriétés temporelles des temps d'exécution. Elle n'est toutefois pas satisfaisante : les gains des caches sont dans la pratique plus avantageux que leurs inconvénients. Les industriels recourent ainsi généralement à des solutions hybrides : les caches les moins problématiques sont activés, mais restent contraints. Une approche fréquente consiste à forcer l'ordonnanceur à vider les caches à chaque élection de tâche. La tâche nouvellement élue dispose ainsi d'un cache dans un état qui est toujours déterminé (vide).

D'autres approches consistent à utiliser les caches comme des mémoires indépendantes. En fonction du matériel, les caches peuvent être adressables : c'est-à-dire qu'un utilisateur peut les reconfigurer en tant que mémoires. Quand cette fonctionnalité n'est pas présente, elle peut être émulée en combinant un préchargement des caches et leur verrouillage. Cette méthode consiste à effectuer des accès choisis à des plages d'adresses mémoires, afin de peupler le cache avec leur contenu. Le cache peut ensuite être verrouillé afin d'éviter l'éviction de leur contenu. Cette technique, souvent utilisée pour charger des instructions, permet d'utiliser un cache comme une mémoire locale dont le contenu est immuable. Ce cas particulier d'utilisation peut servir à isoler un noyau de son application en le faisant résider dans ces mémoires particulières.

Le partitionnement des caches est une autre technique, qui consiste à dédier une fraction d'un cache à une tâche donnée [Wol93 ; Mue95 ; Mit17]. Au lieu d'avoir toutes

les tâches en concurrence sur un ou plusieurs caches, cette pratique permettrait à chaque tâche de disposer exclusivement d'un sous-ensemble d'un cache. L'élection d'une tâche ne bouleverserait ainsi pas le contenu du cache utilisé par une tâche précédente. Il est toutefois à noter que l'utilité pratique de ce genre d'approche a été récemment questionnée [Alt+16]. En effet, il est ardu de déterminer dans quelle mesure chaque tâche doit utiliser le cache. Une mauvaise estimation peut ainsi conduire à des dégradations encore plus importantes des propriétés temporelles de l'application.

2.4.3 Prévention des accès simultanés multicœurs

Principes généraux

Au-delà des périphériques, les principales ressources partagées entre les cœurs sont les mémoires partagées, pouvant héberger des instructions ou des données. Certaines cartes matérielles disposent de mémoires locales aux cœurs (p. ex. *scratchpads*) ; celles-ci sont non seulement plus rapides, mais permettent également de réduire la congestion des bus des mémoires partagées si celles-ci sont utilisées. Elles permettent à chaque cœur processeur d'effectuer des calculs locaux requérant des données sans solliciter de mémoire partagée. Des systèmes tirant avantage de ces composants ont déjà été conçus et ont pu être montrés viables dans l'optique de réduire le partage de ressources [Tab+19 ; Whi+12]. Certaines mémoires partagées peuvent permettre, sous certaines conditions, des accès simultanés depuis différents cœurs sans engendrer d'interférences temporelles significatives, en fonction des plages d'adresses auxquelles les cœurs accèdent, et en fonction de la découpe en bancs mémoires du matériel. Il est également possible d'utiliser les caches comme des mémoires locales, notamment quand ceux-ci sont adressables. Dans le cas contraire, des stratégies de partitionnement de cache peuvent être mises en place. Il peut également être intéressant d'étudier des approches de partitionnement dynamiques de caches. Si l'ordonnanceur multitâches cause une partie des problèmes d'utilisation des caches, on peut imaginer des solutions proactives à son initiative [Thi+11].

Un partage problématique de ressources matérielles peut être limité ou évité, afin d'élaborer des systèmes robustes à ces interférences temporelles [SBP19]. Des solutions dynamiques, appliquées à l'exécution permettent de tempérer ces perturbations. D'autres solutions peuvent être préventives en étant prises en compte dans la conception du système. Ce qui suit n'est bien sûr pas une liste exhaustive, mais montre quelques exemples remarquables tirés de l'état de l'art.

Mitigations à l'exécution

Le *Single Core Equivalence framework* [Man+15] est un ensemble de techniques permettant un partitionnement dynamique des ressources partagées, visant à établir une isolation entre les différents cœurs à l'exécution et ainsi se ramener à un contexte assimilable à celui du monocœur, qui est mieux maîtrisé. Les trois principaux leviers utilisés sont le verrouillage de cache par coloration [8] [Man+13], MemGuard [Yun+13] et PALLOC [Yun+14]. Elles permettent d'affecter des portions de caches à certaines tâches, de réguler la bande passante des bus mémoires et d'améliorer l'allocation dynamique de

8. En anglais : *colored cache lockdown*.

pages mémoires en fonction de l'affinité des tâches avec certains bancs mémoires. Ces trois techniques ont été implémentées par leurs auteurs sur un noyau Linux, aussi nous ne les détaillons pas davantage, car leur caractère dynamique semble peu propice aux systèmes temps-réels de sûreté devant respecter des contraintes de certification.

Le modèle d'exécution PREM [Pel+11] permet de découper l'exécution d'un programme en phases de calcul et d'accès mémoires. Une stratégie d'ordonnancement dynamique intelligente utilisant ces concepts permet de limiter les accès simultanés aux ressources matérielles à l'exécution. D'autres travaux s'appuyant sur ce modèle d'exécution contribuent à cet effort [Bak+12]. Cette approche reste toutefois dynamique, alors que nous préférons des solutions statiques, offrant de plus fortes garanties de sûreté de fonctionnement.

L'étude de points d'intérêt temporels [CC18] permet d'identifier des zones de code sensibles aux interférences, et d'insérer des moyens de mitigation dans un binaire déjà compilé. En fonction de la technique de mitigation choisie, leur caractère dynamique peut varier.

D'autres approches se reposent sur un hyperviseur hébergeant des partitions (au sens ARINC-653 [Air15]) dans des machines virtuelles dédiées [Jea+13]. L'objectif sous-jacent étant de disposer d'un logiciel de contrôle permettant l'administration du logiciel applicatif que constituent les partitions. Cette technique permet d'identifier les événements matériels sensibles — c'est-à-dire sources d'interférences — et de contrôler leur occurrence, afin de réduire leurs impacts négatifs sur le WCET.

Mitigations par conception

Le TDM présenté en section 2.2.3 a été particulièrement étudié en tant que moyen de mitigation par conception. Cela est rendu possible par l'allocation de plages de temps immuables réservées statiquement, offrant des capacités de détection des erreurs temporelles satisfaisantes pour les systèmes temps-réels de sûreté [JCD11]. Les solutions basées sur le TDM peuvent être construites manuellement, requérant une charge de travail importante, mais peuvent aussi être générées. En particulier, certains systèmes automatiques cherchent à intégrer des zones d'exclusion temporelle [Bon+12 ; Bec+16 ; FJK20] en se basant sur des outils d'analyse temporelle ainsi que des modèles du matériel. L'objectif étant d'ajouter aux contraintes d'ordonnançabilité des contraintes d'exclusion mutuelle entre fenêtres temporelles, évitant ainsi par conception les accès simultanés aux ressources matérielles partagées pouvant être réalisés sur ces plages de temps.

2.5 Motivations de nos travaux

Nous avons parcouru dans ce chapitre les principes fondamentaux qui régissent les systèmes temps-réels de sûreté, en nous concentrant sur un périmètre bien précis qui découle de notre positionnement décrit en section 2.1. Si ces systèmes sont déployés avec succès sur le terrain depuis plusieurs décennies, ils doivent s'adapter aux évolutions du matériel dont ils sont tributaires. L'estimation d'un WCET combinant *soundness* et *tightness* reste ainsi un exercice délicat. De plus, l'avancée inéluctable des processeurs multicœurs dans les applications de sûreté force à reconsidérer les approches existantes.

Soulignons que les interférences causées par le matériel n'existent que de par la manière dont le matériel est utilisé. Ce truisme explique d'une part pourquoi les industriels sous-utilisent généralement ces composants, mais permet de proposer une méthode optimisant leur utilisation dans un cadre temps-réel de sûreté. Deux pistes de réflexion se dégagent de ce constat. Elles se déclinent en des axes de recherches indépendants que nous explorons dans ces travaux.

1. Les problèmes liés à l'utilisation du multicœur sont intrinsèques à la conception de l'applicatif. Nous sommes convaincus qu'une application prenant en compte, dès sa conception, des contraintes de simultanéité peut offrir de fortes garanties de sûreté quant à l'utilisation des ressources matérielles partageables. Nous étudions ce problème en profondeur en partie II.
2. Les interférences au sein d'un cœur, causées principalement par les caches sont un problème d'optimisation. À cause du modèle de développement des industriels (dont : la pluralité des équipes et l'exécution multitâches), il semble impossible de concevoir un applicatif tirant pleinement parti des caches sans remettre en cause ce modèle. Nous pensons que c'est au moment de leur intégration sur matériel qu'il est possible de les coordonner. La partie III détaille nos travaux sur cet axe de recherche.

Ces deux axes de recherche contribuent ensemble à l'élaboration de stratégies de réduction des interférences temporelles d'une application temps-réelle de sûreté, qui est l'objectif général de cette thèse.

Chapitre 3

Cadre de travail industriel

Sommaire

3.1 Présentation

Ces travaux étant financés par l'entreprise Krono-Safe [1], nous avons pu accéder sans restriction au produit commercial ASTERIOS® [2], qui est issu de deux décennies de recherche académique sur les systèmes temps-réels de sûreté. Asterios consiste en une suite outillée et un noyau temps-réel [3] permettant la réalisation d'applications temps-réelles de sûreté telles que décrites au chapitre 2. Nous détaillons dans ce chapitre l'état de l'art de cette technologie [4], afin de clarifier le positionnement de nos travaux

1. https://www.krono-safe.com
2. Dans le reste de cette thèse, nous préférons écrire « Asterios », par souci de clarté.
3. On peut qualifier par abus de langage ce noyau de RTOS (Real-Time Operating System).
4. Uniquement touchant à nos travaux.

vis-à-vis de ce cadre de travail. Nous détaillons également ici les architectures matérielles sur étagère que nous utilisons dans ces travaux.

3.1.1 Origine d'Asterios

La technologie Asterios est issue du projet OASIS [Dav+98 ; Cam+08 ; Lou+11], développée au CEA [5] à la fin des années 1990 et au début des années 2000. Ciblant le nucléaire civil, ce projet a notamment été utilisé pour réaliser un système d'affichage qualifié (QDS) en partenariat avec Framatome-ANP [Dav+04]. Une déclinaison d'OASIS pour l'automobile voit le jour au sein du CEA au début des années 2010 : PharOS [CAD09 ; Aus+10 ; Lem+11 ; Cha+13].

Krono-Safe est fondée en 2011 comme *spin-off* du CEA, avec comme mission de développer et commercialiser les technologies issues de PharOS et d'OASIS dans les domaines industriels comme l'avionique, l'automobile, le ferroviaire ou les automates industriels. Un premier prototype Kron-OS [Cha+14 ; DBC14] est ainsi produit peu après, avant d'aboutir au produit commercial Asterios [FJK20 ; Car+20 ; MOT20].

3.1.2 Chaîne outillée et noyau temps-réel

La technologie Asterios gravite autour d'une chaîne de compilation, contrôlée *via* l'outil en ligne de commande `psyko`. Une vision macro est présentée en figure 3.1 : l'utilisateur fournit à la chaîne de compilation des fichiers objets contenant son code fonctionnel compilé, des paramètres de configuration et des sources PsyC. Le modèle de programmation PsyC est détaillé plus tard en section 3.3 ; nous pouvons considérer ici le PsyC comme un langage de programmation permettant l'expression des exigences temporelles des tâches. La chaîne analyse le PsyC en profondeur et permet la génération d'exécutables indépendants nommés *partitions*. Une partition contient un ensemble de tâches et chaque partition correspond dans la pratique à un niveau de criticité différent [6]. Un exécutable nommé « runtime », spécifique à une application (c'est-à-dire à un ensemble de partitions) est également généré. Ce dernier contient les données globales de l'application ainsi que les tables de configuration du noyau : tampons de communication, descripteurs de tâches, plan d'ordonnancement pré-calculé, *etc.*

Chacun de ces exécutables est destiné à être vérifié hors ligne par une chaîne inverse : Asterios Checker [MOT20]. Cet outil de vérification permet la démonstration que les exécutables générés par `psyko` sont conformes aux exigences d'entrée des utilisateurs. Cette stratégie — que nous ne détaillons pas plus ici [7] — permet d'éviter que `psyko` ne soit assujetti aux contraintes de développement très strictes des plus hauts niveaux de criticité (c.-à-d. DAL-A).

Les partitions et la « runtime » ne sont pas autosuffisantes ; elles requièrent d'être pilotées par *Asterios RTK*, qui est le noyau temps-réel porté pour une cible matérielle particulière. Ce composant est exclusivement développé par Krono-Safe et est fourni à l'utilisateur sous la forme d'un exécutable binaire. Dans sa version certifiée, il est

5. Commissariat à l'énergie atomique et aux énergies alternatives — `https://www.cea.fr`.

6. Dans ces travaux, on considère la criticité comme relative aux exigences de sûreté de fonctionnement. Deux niveaux de criticité différents peuvent être des tâches DAL-A (à criticité forte) et DAL-C (à criticité moindre).

7. Asterios Checker est détaillé dans le papier de Methni, Ohayon et Thurieau [MOT20].

accompagné d'un dossier de certification permettant à l'utilisateur de justifier son usage auprès des autorités de certification.

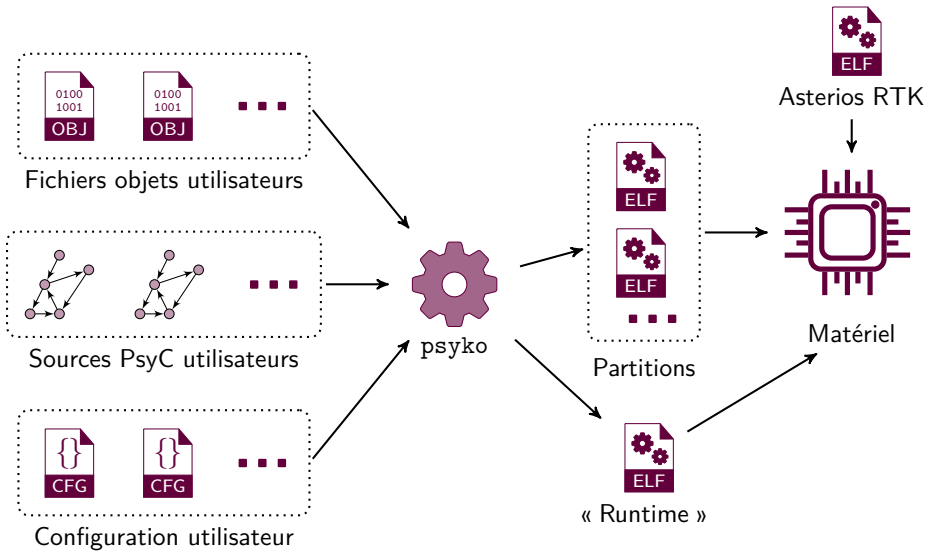


Figure 3.1 – Vue d'ensemble de la chaîne outillée Asterios. Les fichiers fournis par l'utilisateur permettent la génération d'exécutables indépendants, destinés à être implantés sur matériel avec un noyau temps-réel sur étagère.

Dans ce chapitre, nous présentons trois concepts majeurs de l'approche Asterios sur lesquels s'appuient nos travaux :

— le modèle de calcul des automates contraints par le temps (section 3.2) ;

— le modèle de programmation PsyC (section 3.3) ;

— le modèle d'exécution d'Asterios (section 3.4).

3.2 TCA : automates contraints par le temps

Asterios s'appuie sur un modèle de calcul particulier : les automates contraints par le temps, ou TCA (Time-Constrained Automata), implicitement utilisés par les prédécesseurs d'Asterios jusqu'à leur formalisation par Lemerre et al. [Lem+10], sur laquelle se sont appuyés des travaux ultérieurs [JCD11]. Ce modèle de calcul est utilisé comme fondation d'une de nos contributions décrite au chapitre 5, aussi nous le détaillons ici.

Le formalisme TCA définit un *bloc* comme étant une séquence de calculs contrainte par au moins une des contraintes suivantes (considérant des unités de temps homogènes).

— *After* indique que l'exécution d'un bloc ne peut débuter qu'à partir d'une date précise. On parle aussi de date de début au plus tôt, ou ESD (Earliest Start Date).

— *Before* indique que l'exécution d'un bloc doit terminer avant l'occurrence d'une date particulière (on parle d'échéance — *deadline* en anglais).

Ces automates sont formalisés comme des graphes dirigés où les arcs représentent les blocs et les nœuds signifient les contraintes temporelles appliquées aux arcs les joignant. Un nœud est susceptible de cumuler ces deux classes de contraintes. En comptant ce cas particulier, trois types de nœuds sont utilisables et sont représentés comme suit :

— les nœuds *after*, ou ▷, comme indiqué par le nœud S en figure 3.2 ;

— les nœuds *before*, ou ◁, non représentés ici, car très peu utilisés ;

— les nœuds de *synchronisation*, ou ◇, qui cumulent les contraintes *before* et *after*, comme indiqués par les nœuds A, B, C, D et E en figure 3.2.

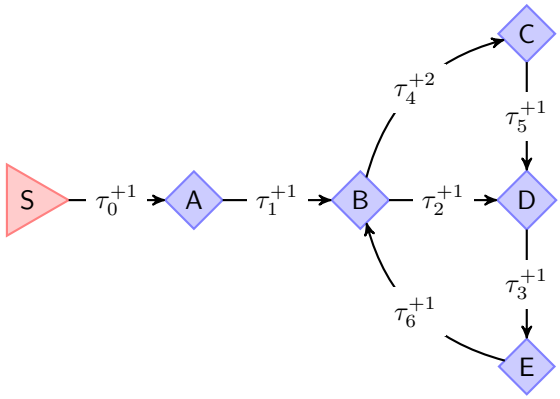


Figure 3.2 – Exemple d'automate contraint par le temps, débutant au nœud S. τ_0 doit compléter avant d'atteindre A, auquel succède τ_1 qui doit compléter avant d'atteindre B. À partir du nœud B, deux branchements sont possibles : le premier impose à τ_2 de compléter avant D tandis que le second force τ_4 à compléter avant C puis à τ_5 de compléter avant D. Finalement, lorsque D est atteint, τ_3 doit compléter avant E, puis τ_6 avant B. Ce comportement est ensuite répété depuis B.

Définition 5 (Trivialité d'un automate contraint par le temps)**.** *Un TCA est dit* trivial *si et seulement si chaque nœud de cet automate dispose d'un seul arc de sortie (p. ex. figure 3.3). À l'inverse, si au moins un nœud dispose de plus d'un successeur, le TCA est dit* non trivial *(p. ex. voir figure 3.2).*

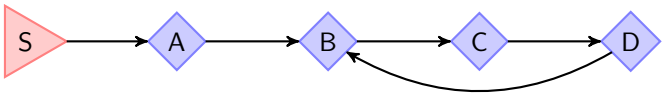


Figure 3.3 – Exemple de TCA trivial, décrivant ainsi un comportement périodique : après avoir débuté au nœud S, le nœud A est accepté. Puis la séquence de nœuds B, C et D est périodiquement répétée.

Ce modèle propose des techniques de simplification de graphes permettant la détection de TCA invalides ou la suppression de contraintes redondantes. Celles-ci sont formellement définies dans l'article d'origine ; dans nos travaux tous les TCA que nous considérons sont présentés dans leur forme irréductible.

Une propriété intéressante des TCA réside dans leur capacité à en dériver des *modèles d'exécution*, comme des stratégies d'ordonnancement, préservant les contraintes temporelles délimitant les blocs. Ainsi, le mécanisme original des TCA exploité par OASIS et PharOS consiste en une variation de l'ordonnancement EDF appelée *EDF-dyn*. Celui utilisé par Kron-OS puis par Asterios est différent et est détaillé en section 3.4.

3.3 Modèle de programmation PsyC

Le PsyC est un modèle de programmation parallèle et synchrone, largement compatible avec le langage C (d'où son nom). Il s'agit d'une extension du langage C qui permet la déclaration de tâches temps-réelles, la définition de canaux de communication entre plusieurs tâches et permet d'associer des contraintes temporelles aux déclarations impératives du C. Introduit avec OASIS par David et al. [Dav+98], le PsyC permet l'expression du modèle de calcul TCA [Lem+10]. Il est encore utilisé par Asterios pour cela, même si ce langage a subi des évolutions propres aux travaux de Krono-Safe.

3.3.1 Expression des horloges logiques

Les horloges constituent une composante clef du modèle de programmation PsyC, car toutes les constructions temporelles y sont associées. Une horloge peut être considérée comme une suite arithmétique entière positive (c_n) telle que $c_n = o + pn$, où $o \in \mathbb{N}$ et $p \in \mathbb{N}$ dénotent respectivement l'origine [8] de l'horloge et sa période. Cela s'exprime en PsyC comme en figure 3.4, où « realtime » est une source de temps externe, c'est-à-dire le générateur primaire des tics. Ici, on considère que « realtime » est une horloge comptant le temps physique, car il s'agit de l'utilisation la plus courante.

```
1 clock c1 = 1 * realtime;
2 clock c2o1 = 2 * realtime + 1;
```

Figure 3.4 – Déclaration de deux horloges en PsyC, basées sur la source de temps « realtime ». Leur représentation graphique est montrée en figure 3.5.

3.3.2 Utilisation des agents pour décrire des TCA

Les tâches s'exécutant dans Asterios ont une durée de vie virtuellement illimitée : elles sont créées au démarrage du système et ne sont détruites qu'en cas d'erreur fatale ou lorsque le système atteint une date de fin de vie, mettant fin à son exécution complète ; ces tâches sont nommées *agents*. Le PsyC permet d'exprimer des agents, qui

8. En anglais : *offset*. C'est la valeur initiale de l'horloge à l'occurrence de son premier tic.

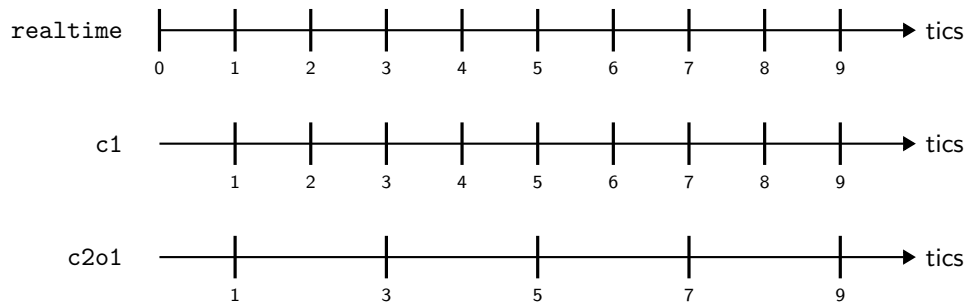


Figure 3.5 – Illustration des horloges décrites en figure 3.4.

entremêlent instructions impératives et contraintes temporelles, décrivant *in fine* des blocs d'instructions contraints dans le temps, c'est-à-dire des TCA.

Les instructions de flot de contrôle héritées du C (c.-à-d. les branchements conditionnels) permettent la définition de successeurs multiples, donc de TCA non triviaux. Comme montré en figure 3.6, les instructions `advance` permettent la déclaration de contraintes de *synchronisation*, fixant l'échéance des instructions qui les précèdent et déterminant les dates de début au plus tôt de celles qui suivent. Un mécanisme d'annotations, préfixé par « `@` » permet de nommer une instruction `advance`. Nous l'avons introduit dans le langage PsyC pour faciliter certains des travaux que nous détaillons en partie II.

Par analogie avec les machines de Turing, on considère qu'un agent est constitué d'un ruban infini d'actions élémentaires, ou EA (Elementary Action), où une EA représente l'exécution d'un bloc d'instructions impératives, délimité par deux instructions PsyC `advance`. L'EA est ainsi la brique d'exécution fondamentale d'un agent.

3.3.3 Communications inter-agents

Les agents sont des tâches isolées les unes des autres, et les données qu'elles échangent doivent transiter *via* des canaux de communication bien identifiés. Les dernières versions d'Asterios permettent à l'utilisateur d'utiliser de la mémoire partagée, toutefois cette option n'est pas privilégiée, aussi nous nous concentrons ici sur les moyens de communication mis à disposition par le PsyC. Ceux-ci implémentent le *principe de visibilité* [LO12], garant de leur déterminisme, qui s'énonce comme suit :

— Soit Λ une horloge logique.

— Soient d_{esd} et d_{ddl} respectivement les dates logiques de début au plus tôt et d'échéance d'une EA, en tics de l'horloge logique Λ.

— Un agent produisant des données dans un canal de communication ne peut les estampiller qu'avec une date logique $d_\lambda \geq d_{ddl}$.

— Un agent ne peut consommer des données depuis un canal de communication que si l'estampille associée à cette donnée est telle que $d_\lambda \leq d_{esd}$.

```
clock clk = 1 * realtime;
agent task
( uses realtime, defaultclock clk, starttime 1 )
{
  /* Point d'entrée de l'agent */
  body start {
    /* τ_0^{+1} */
    @A advance 1;
    /* τ_1^{+1} */
    jump loop; /* ← Transition immédiate à "body loop" */
  }

  /* Comportement périodique de l'agent */
  body loop {
    @B advance 1;

    if (condition) {
      /* τ_4^{+2} */
      @C advance 2;
      /* τ_5^{+1} */
    } else {
      /* τ_2^{+1} */
    }
    @D advance 1;
    /* τ_3^{+1} */
    @E advance 1;
    /* τ_6^{+1} */
    /* ← Le "body loop" répète */
  }
}
```

Figure 3.6 – Déclaration d'un agent PsyC permettant l'expression du TCA montré en figure 3.2. L'horloge choisie ici (`clk`) est affectée à une valeur arbitraire à titre d'exemple.

Il est intéressant de noter que la combinaison des TCA et du principe de visibilité permet l'expression du paradigme LET (Logical Execution Time) [KS12], que l'industrie automobile cherche à intégrer.

Les variables temporelles, ou VT (Variable Temporelle), [Dav+98] expriment des flots discrets de données qu'un agent producteur met à disposition d'autres agents. Elles sont associées à des horloges pouvant être distinctes de celles des agents, et ainsi exprimer des mécanismes de consommation et de production asymétriques complexes. Dans nos travaux, nous utilisons exclusivement ce moyen de communication, mais d'autres existent, comme les *message box* [JCD11] ou les *streams* (Asterios implémente les VT et les streams).

Les figures 3.7 et 3.8 montrent la syntaxe PsyC permettant d'exprimer une communication entre deux agents *via* une VT. Un agent (ici nommé « sender ») écrit dans une VT à chaque tic logique et un autre (nommé « receiver ») effectue une lecture de cette VT à chaque tic logique. Toutefois, les VT sont associées à une horloge logique qui peut être différente — c'est le cas dans cet exemple. Les interactions entre ces agents

et la VT sont résumés en figure 3.9, qui montre une application concrète du principe de visibilité précédemment énoncé.

```
// Mêmes horloges qu'en figure 3.5
source realtime;
clock c1 = 1 * realtime;
clock c2o1 = 2 * realtime + 1;

// Variable temporelle publiée selon c2o1
temporal unsigned int data with c2o1;

// Agent périodique candencé sur c1
agent sender (uses realtime, defaultclock c1)
{
  // La VT data est émise uniquement pour l'agent « receiver »
  display { data : receiver; }
  // Données globales de l'agent
  global { unsigned int counter = 1u; }
  // Corps de la tâche
  body start {
    // Mise à jour de la VT par l'agent
    data = counter;
    // Incrément et fin de l'EA en cours
    counter += 1u;
    advance 1;
  }
}
```

Figure 3.7 – Agent PsyC produisant une VT consommé par l'agent montré en figure 3.8.

3.4 Modèle d'exécution d'Asterios

L'approche Asterios s'articule autour d'une chaîne de compilation dédiée, comme vu en section 3.1.2. Nous avons expliqué que celle-ci génère un exécutable particulier (« runtime »), qui contient nombre d'informations calculées hors ligne, mises à disposition d'un noyau temps réel. Le modèle d'exécution d'Asterios en tire parti pour permettre un partitionnement spatial et temporel de l'application.

3.4.1 Exécution des agents

Asterios repose sur le concept de *couche système* (ou « psyslayer ») qui est hérité d'OASIS [Dav+04] : le contexte d'exécution de chaque agent est couplé avec un contexte d'exécution dit *système* en charge d'effectuer les communications entre agents et plus généralement de gérer l'évolution du système vu par l'agent. Cette séparation permet de considérer le noyau sur lequel s'appuie Asterios comme un *micro-noyau* où les fonctions applicatives sont déléguées à des composants exécutés dans un contexte séparé de celui du noyau.

Les agents accèdent ainsi aux services proposés par le micro-noyau *via* l'exécution d'appels systèmes, traités par la couche système. Le cœur du noyau étant principalement

```
// Mêmes horloges qu'en figure 3.5
source realtime;
clock c1 = 1 * realtime;
clock c2o1 = 2 * realtime + 1;

// Variable temporelle publiée selon c2o1
temporal unsigned int data with c2o1;

// Agent périodique candencé sur c1, débutant au deuxième tic de
// l'horloge logique sur laquelle la VT est basée.
agent receiver (uses realtime, defaultclock c1, starttime 2 with c2o1)
{
  // La VT data est consultée avec une profondeur d'historique de 1
  consult { 1 $ data; }
  // Corps de la tâche
  body start {
    // Lecture de la dernière valeur produite dans la VT.
    const unsigned int received_data = $[0]data;
    advance 1;
  }
}
```

Figure 3.8 – Agent PsyC consulant la VT produite par l'agent montré en figure 3.7.

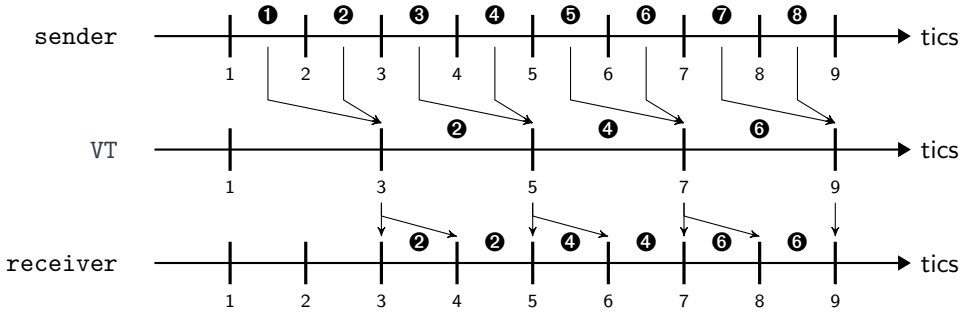


Figure 3.9 – Illustration du flot temporel des données liées à la VT partagée entre les agents montrés en figures 3.7 et 3.8. Les valeurs de ❶ à ❽ représentent la valeur d'un compteur incrémenté à chaque EA de « sender » qui transite vers « receiver » *via* la VT. Les flèches indiquent l'application du principe de visibilité et montrent les données visibles.

chargé de la gestion des différents contextes d'exécution, de l'élection des tâches et du respect de l'isolation entre tâches.

3.4.2 Cloisonnement entre tâches

Les agents étant des tâches critiques, il convient de limiter les interactions non désirées entre elles. Pour cela, Asterios propose des mécanismes de cloisonnement spatial et temporel, qui est une approche répandue dans l'industrie avionique [WH09].

Ainsi, une tâche ne doit pas être autorisée à modifier de la mémoire à laquelle une autre accède, sans l'avoir explicitement indiqué. Bien que les aspects de sûreté de fonctionnement priment sur ceux de sécurité, il peut être intéressant dans certains contextes que les tâches ne puissent s'inspecter mutuellement, afin d'éviter des fuites d'informations. Le modèle d'exécution d'Asterios implémente ainsi un mécanisme de partitionnement spatial auquel nos travaux ont contribué, comme détaillé au chapitre 4. Ainsi, plusieurs mécanismes successifs, à la compilation et à l'exécution permettent de prévenir et de détecter les accès mémoire inter-tâches n'étant pas explicitement spécifiés par le concepteur de l'application. Ce concept s'étend également au noyau temps-réel et aux données de « runtime ».

En tant que partie d'un système temps-réel de sûreté, les applications Asterios sont souvent soumises à de fortes contraintes de cadence et de latence, comme expliqué en section 2.2. Comme ces applications sont écrites en PsyC pour permettre l'expression de TCA, plusieurs politiques d'ordonnancement peuvent être appliquées. Dans le domaine avionique, une stratégie de partitionnement temporel strict basé sur le principe du TDM est privilégiée. Un plan d'ordonnancement statique est généré hors ligne par la chaîne de compilation, conformément à la politique RSF (Repetitive Sequence of Frames) [ACD10]. Il s'agit de la composition des contraintes temporelles de toutes les tâches implantées sur un même cœur. Ce mécanisme d'ordonnancement est *préemptible*, c'est-à-dire que le code utilisateur constituant les EA peut ne pas s'exécuter d'un seul tenant. La RSF est basée sur un mécanisme de provision de temps d'exécution, appelés *budgets* : chaque EA doit disposer d'un budget de temps, qui dans la pratique est équivalent au WCET de cette EA. Ce temps doit être déterminé par l'utilisateur d'Asterios et est fourni comme entrée de la chaîne outillée.

Figure 3.10 – Exemple de RSF générée à partir des agents décrits en figures 3.7 et 3.8. « sender » est de couleur prune et « receiver » en gris. La quantité de temps allouée à chaque agent est ici arbitraire.

La figure 3.10 montre un exemple d'un tel plan d'ordonnancement si l'on considère une application monocœur constituée des deux agents vus précédemment en figures 3.7 et 3.8. Dans cette représentation, chaque barre verticale marque un tic d'ordonnancement, qui correspond au tic d'une horloge logique calculée en fonction des contraintes de cadencement du système. Un *intervalle* est délimité par deux tics adjacents. Chaque intervalle contient une ou plusieurs *frame*, chaque *frame* représentant une quantité de temps physique allouée à un agent dans lequel ses EA peuvent s'exécuter. La portion d'un intervalle qui n'est pas occupée par une *frame* correspond à du temps dit *idle*. Celui-ci permet d'exécuter des tâches de fond particulières et de garantir certaines marges nécessaires à la sûreté temporelle du système. Dans cet exemple, le dernier intervalle de la RSF « boucle » sur lui-même : les contraintes temporelles qu'il décrit s'appliquent jusqu'à la date de fin de l'application qui est calculée hors ligne. Lorsque

le système atteint cette date, il s'arrête de manière contrôlée. La forme de la RSF dépend des algorithmes de son calcul, aussi est-elle parfois surprenante. Par exemple, les trois derniers intervalles sont identiques et pourraient être fusionnés en un seul. Cette suboptimalité, qui est évidente dans un cas trivial comme celui-là, résulte de différentes stratégies cherchant à produire un résultat viable de manière plus générale.

3.5 Étude de la plateforme matérielle de référence

Dans la suite de nos travaux, nous utilisons presque exclusivement une plateforme matérielle de référence : le MPC5777M, un microcontrôleur PowerPC du fabricant NXP [NXP17]. Cette cible matérielle vise les applications de sûreté et est conçue pour permettre des certifications ASIL-D (le plus haut de niveau de sûreté dans le domaine de l'automobile, selon l'ISO 26262 [ISO18]). Ainsi, son implémentation matérielle permet une faible variation des temps d'exécution : une séquence donnée d'instructions s'exécutera toujours avec le même nombre de cycles CPU entre différentes exécutions. Bien sûr, le logiciel final est composé de branchements, qui, combinés avec des mises à jour des caches, l'ordonnancement et les accès simultanés aux ressources matérielles partagées, conduisent à des variabilités observables.

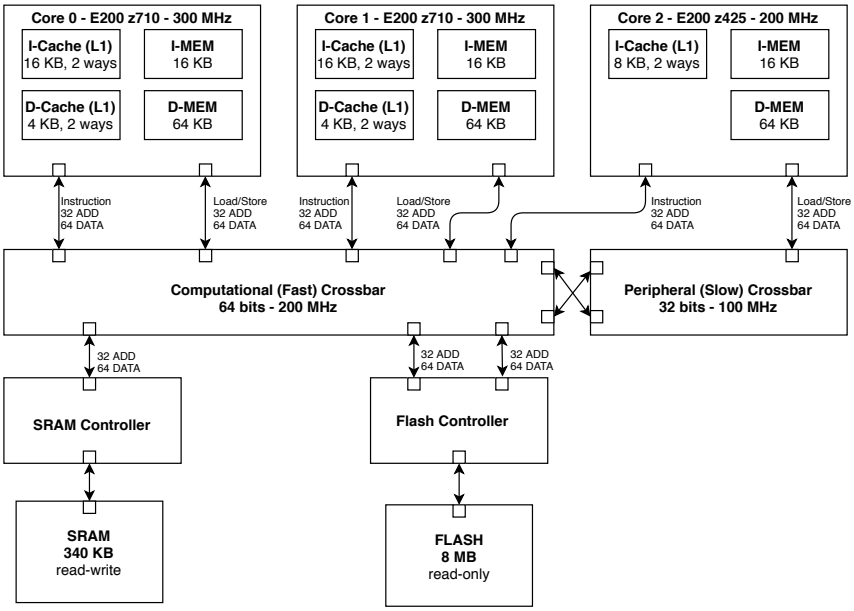


Figure 3.11 – Diagramme d'architecture simplifié du microcontrôleur MPC5777M.

Cette architecture hétérogène est composée de trois cœurs Power ISA 2.06 VLE [9] visibles : deux processeurs e200z710 cadencés à 300 MHz (*single-issue*) et un processeur e200z425 cadencé à 200 MHz (*dual-issue* — c.-à-d. que le cœur n° 2 a la capacité

9. L'extension VLE permet des instructions de taille variable (sur 16 ou 32 bits).

d'exécuter simultanément deux fois plus instructions que le n° 1). L'un des processeurs e200z710 est exécuté en doublon avec un quatrième processeur e200z709 qui n'est pas programmable, mais permet de vérifier sa bonne exécution (*lockstep*). Cette carte dispose de 404 kB de mémoire SRAM (lecture-écriture), de 8 MB de mémoire FLASH (lecture seule). Elle présente aussi des mémoires d'instructions et de données internes aux cœurs (I-MEM et D-MEM) complémentant les mémoires partagées SRAM et FLASH. Plus particulièrement, chaque cœur peut accéder simultanément à ses mémoires locales sans créer d'interférence sur les autres cœurs. Les composants-clés de cette architecture sont montrés en figure 3.11. Seuls des caches de premier niveau (L1) sont disponibles — il n'y a pas de caches L2 ou L3 — où les données et les instructions sont séparées, sauf pour le cœur n° 2 qui dispose uniquement d'un (plus petit) cache d'instructions. Les accès aux données réalisés par le cœur n° 2 transitent *via* le crossbar périphérique, avant de rejoindre le crossbar calculatoire, qui permet d'atteindre les contrôleurs de SRAM et de FLASH. Les cœurs n° 0 et n° 1 contournent le crossbar périphérique pour effectuer ces accès. Notons la présence de caches additionnels (pas indiqués sur la figure), qui sont présents au niveau du contrôleur de FLASH, sur chaque bus (un pour chaque port). Ces caches disposent de quatre entrées, sont associatifs par ensemble, disposent de deux voies et permettent le stockage de 256 octets.

3.6 Conclusion

Nous avons vu en section 3.4.2 qu'Asterios propose un mécanisme de partitionnement temporel, critique pour les applications avioniques. Celui-ci requiert l'évaluation de WCET par l'utilisateur. Krono-Safe cherche à y contribuer indirectement en proposant des approches permettant de faciliter l'estimation des WCET et d'établir des techniques de mitigations. C'est cet angle d'attaque du problème du WCET au sein d'Asterios qui motive nos travaux. La suite de ces travaux détaille nos contributions, elles se hissent sur l'existant d'Asterios pour analyser des applications temps-réelles de sûreté et chercher à mitiger leurs interférences temporelles.

Chapitre 4

Méthode de partitionnement spatial

Sommaire

4.1 Motivations

Dans le chapitre 2, nous avons rappelé notre positionnement vis-à-vis des systèmes temps-réels de sûreté, et particulièrement insisté sur l'une des problématiques de ce domaine en section 2.3.1 : la détermination du WCET. Appliqué à notre contexte de

travail Asterios, détaillé au chapitre 3, il s'agit en particulier du WCET du noyau et des EA constituant une application. Dans la section 2.3.3, nous avons brièvement mentionné l'impact significatif du multicœur et des mémoires caches sur le WCET.

Au début de nos travaux, Asterios ne disposait pas d'un mécanisme de placement mémoire nous satisfaisant. Nous avons donc commencé par mettre en œuvre une solution de placement mémoire permettant d'effectuer un placement fin qui sert nos travaux, tout en prenant en compte les contraintes de protection mémoire et de certification inhérentes aux systèmes temps-réels de sûreté. En effet, le placement mémoire doit être intimement lié aux considérations de protection mémoire pour fournir une solution acceptable. Ces travaux ont donné suite à une publication à la conférence ISSRE 2020 (*Industry Track*) [GH20].

4.2 État de l'art du partitionnement spatial statique

Le partitionnement spatial est un sujet très exploré : plus de quarante ans de littérature décrivent des solutions d'isolation inter-tâches s'assurant de l'impossibilité d'accès invalides à l'exécution [Sal74]. C'est une option proposée par défaut sur la plupart des systèmes d'exploitation contemporains sur une vaste gamme d'objets, tant que ceux-ci fournissent un support matériel. Nos travaux ciblant les systèmes temps-réels de sûreté, nous privilégions l'étude de solutions statiques, écartant les approches dynamiques plus répandues [Ach20].

Le partitionnement mémoire est principalement implémenté grâce à des extensions matérielles dédiées, comme les unités de protection ou de gestion de la mémoire — respectivement MPU (Memory Protection Unit) ou MMU (Memory Management Unit). En faisant l'hypothèse d'une configuration adéquate par le logiciel, ces dernières permettent la levée d'exceptions matérielles en cas d'accès invalides à la mémoire détectés au cours de l'exécution du système. Cette fonctionnalité permet de satisfaire à la fois des contraintes de sécurité et de sûreté [WH09]. Les MMU sont souvent plus complexes que les MPU car offrent une gestion plus avancée de la mémoire, comme l'adressage virtuel et la notion de pages mémoires. Pour être utilisable dans un contexte Asterios, notre solution doit gérer ces deux cas, indépendamment du modèle mémoire sous-jacent (avec ou sans adressage virtuel).

Le micro-noyau PikeOS est documenté comme configurant statiquement la MMU lors du démarrage, de telle sorte à ce que cette dernière ne puisse être reconfigurée durant l'exécution du système, en dehors d'un cadre prévu hors-ligne [BO16]. Il n'est pas précisé si cette configuration est basée uniquement sur des tables générées hors ligne, ou si celles-ci sont en parties calculées dynamiquement au moment du démarrage. D'autres systèmes, orientés dans le domaine avionique, mentionnent des approches similaires, sans toutefois donner davantage de détails [Per+16].

Les travaux les plus proches de notre attendu sont issus de l'approche OASIS, d'où Asterios est originaire. Ils mentionnent une technique de partitionnement spatial gérée par le noyau temps-réel OASIS, lequel configure la MMU d'un processeur Intel en chargeant des tables générées statiquement hors-ligne [Cam+08 ; Lou+11 ; DD07]. Ce système présente ainsi la caractéristique de déléguer complètement la construction des tables de protection à des outils hors-ligne, évitant d'avoir à certifier du logiciel s'exécutant sur le système final. Toutefois, leur méthode de génération n'est pas documentée.

Nous avons ainsi observé que des techniques de partitionnement spatial avec des parts variables de logique à l'exécution sont mentionnés dans la littérature, sans toutefois être explicitement documentés. Il est en effet probable que ces techniques soient considérées par d'aucun comme propriété intellectuelle, restreignant leur diffusion.

4.3 Définition d'une méthode statique et vérifiable

Nous présentons dans cette section une méthode que nous avons élaborée pour mettre en place une politique de partitionnement spatial statique et vérifiable, qui puisse être appliquée aux systèmes temps-réels de sûreté.

4.3.1 Utilisation de chaînes de compilation qualifiées

Les chaînes de compilation à destination des systèmes temps-réels sont principalement construites autour d'un compilateur, qui convertit des fichiers sources (souvent écrits avec le langage de programmation C) en fichiers objets. Un éditeur de liens [1] a pour charge de générer un exécutable binaire à partir de multiples fichiers objets. Son exécution est souvent contrôlée via un script dédié — le script d'édition de liens — qui décrit l'agencement final des objets en mémoire.

Les industriels des systèmes temps-réels de sûreté ont souvent recours à des chaînes de compilation sur étagère dites *qualifiées*. Celles-ci ont suivi des procédures de validation particulières, comme décrites dans la norme DO-330 [RTC11b]. Ces briques logicielles ne peuvent ainsi pas être altérées ou remplacées par des équivalents n'ayant pas été soumis aux mêmes processus de validation. Ainsi, la méthode que nous proposons ici a comme objectif d'utiliser exclusivement des chaînes de compilation qualifiées, fournies par des tiers, capables de réaliser les opérations de compilation et d'édition de liens. Pour cela, nous ne supportons que les chaînes de compilation ayant les caractéristiques suivantes.

1. L'éditeur de liens doit pouvoir être configuré par lecture d'un script d'édition de liens, ou au moins disposer d'une fonction équivalente. Par la suite, nous supposons implicitement que cette fonction est systématiquement réalisée par un script d'édition de liens, cela étant la méthode la plus courante.
2. Le compilateur et l'éditeur de liens doivent être appelés indépendamment.
3. L'éditeur de liens est déterministe : il produit toujours le même résultat pour le même ensemble ordonné d'entrées (sur sa ligne de commande) ; le respect de cette exigence est capital.

4.3.2 Identification des spécificités matérielles

Comme vu en section 4.2, la fonctionnalité de protection mémoire à l'exécution du système est implémentée par de multiples composants matériels, susceptibles d'amener des contraintes particulières sur le placement mémoire. Elles se traduisent souvent par

1. On considère ici un éditeur de liens (*linker* en anglais) comme étant le programme capable de réaliser la fonction dite de « link+locate », c'est-à-dire la capacité à résoudre des symboles et à les positionner en mémoire.

des exigences d'alignement : une plage d'adresses mémoires à protéger avec certaines permissions devra être alignée suivant certaines règles dictées par le matériel. Les exemples qui suivent illustrent cette diversité.

— La MPU des uniprocesseurs Cortex-M3 requiert que les adresses à protéger soient alignées sur une adresse multiple de la taille de la région à protéger.

— La MPU du microcontrôleur MPC5777M permet la protection de plages mémoires arbitraires sans contrainte d'alignement.

— La MMU des plateformes T1042 et P2020 impose que les pages mémoires à protéger soient alignées sur des adresses correspondant à des puissances de 4 kB, c'est-à-dire que la taille d'une page doit être de 4^n kB, avec $n \in [\![1; 11]\!]$.

L'identification de ces contraintes dictées par le matériel (p. ex. alignement, MMU ou MPU, niveaux de pagination, *etc.*) influencent directement les prochaines étapes de la méthode que nous allons détailler.

4.3.3 Abstraction de placement mémoire

Le placement mémoire consiste à positionner des objets (c.-à-d. les résultats issus d'une compilation) à des adresses mémoires particulières. Il peut être désirable de grouper ensemble plusieurs objets devant être protégés avec des droits d'accès similaires. Les scripts d'édition de liens précédemment évoqués en section 4.3.1 permettent de décrire l'association entre les *sections d'entrée* des objets pris en entrée de l'éditeur de liens et les *sections de sortie* du binaire final auquel elles sont intégrées. Cette primitive de placement mémoire permet une granularité au niveau de la section objet. Certains compilateurs sont capables de générer des sections par symbole, permettant ainsi l'utilisation de cette primitive avec une granularité plus fine. Toutefois, son utilisation seule est laborieuse et passe peu à l'échelle sur des applications de taille modérée. Nous considérons ainsi un niveau d'abstraction plus élevé, mettant l'accent sur une facilité d'utilisation, sans perdre en finesse de placement. Pour cela, nous introduisons les concepts qui suivent. Ceux-ci sont illustrés en figure 4.1.

Définition 6 (Groupes)**.** *La réalisation de politiques de placement mémoire peut nécessiter d'agréger de multiples sections d'entrée en une seule section de sortie. Par exemple, il peut être intéressant de placer toutes les sections d'entrée de plusieurs objets contenant des constantes (à protéger en lecture seule) au sein d'une seule section de sortie. Chaque objet de l'application doit appartenir à au moins un groupe. Les groupes décrivent une application surjective explicitant les relations de placement.*

Définition 7 (Domaines)**.** *Les systèmes utilisant des MPU disposent d'un nombre fixe et réduit de descripteurs mémoires. Il peut être intéressant de grouper ensemble des sections de sortie à protéger avec les mêmes permissions afin de limiter le nombre de descripteurs à utiliser. Ainsi, un domaine est une collection de sections de sortie adjacentes partageant les mêmes droits d'accès mémoires.*

Définition 8 (Régions)**.** *Les régions décrivent les différentes mémoires adressables disponibles et peuvent donc être représentées comme des plages fixes d'adresses physiques. Elles permettent l'implantation des domaines, dans la limite de leurs capacités.*

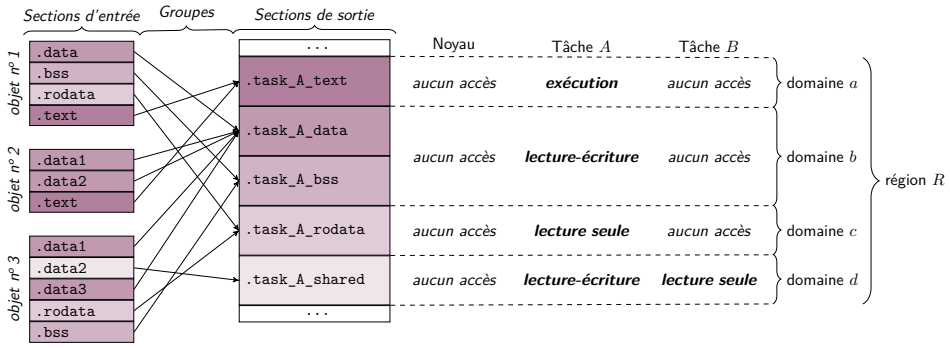


Figure 4.1 – Illustration de stratégie de placement mémoire, où les sections d'entrée de trois objets utilisés par une tâche A sont associés aux sections de sortie constituant l'exécutable final, comme indiquées par les flèches. Celles-ci traduisent l'utilisation des groupes. Par exemple, la section `.data2` de l'objet n° 3 doit être contenue dans la section `.task_A_shared` du binaire final. À droite des sections de sortie du binaire sont récapitulés les permissions mémoires assignées à différents contextes d'exécution, ainsi que les regroupements en domaines (a, b, c et d) et leur implantation dans une unique région R.

4.3.4 Génération automatisée d'un binaire partitionné

Un des produits attendus de notre méthode de partitionnement spatial consiste en un fichier objet contenant les tables statiques de protection mémoire de l'applicatif logiciel. Celles-ci doivent associer des plages d'adresses à des droits d'accès. Pour des raisons de sûreté et de sécurité, ces tables doivent contenir une entrée les décrivant. En effet, il n'est pas envisageable que les tables de protection mémoire puissent être modifiées au cours de l'exécution, car cela pourrait annuler les garanties apportées par le cloisonnement spatial.

Comme notre méthode s'appuie sur des chaînes de compilation sur étagère, nous considérons l'édition de liens comme atomique et pure : nous ne pouvons que constater les résultats de cette opération, sans pouvoir la manipuler autrement que *via* ses entrées. Ainsi, les plages d'adresses consommées ne peuvent être connues qu'après complétion de l'édition de liens. De plus, comme l'éditeur de liens est susceptible d'effectuer des optimisations pouvant affecter le contenu des sections d'entrée (comme la *relaxation*), les tailles manipulées ne peuvent être déterminées avec certitude *a priori*. Cela pose problème pour décrire des pages mémoires de taille variable : en fonction du matériel, il peut être requis que ces pages soient alignées sur leur taille (cf. section 4.3.2), qui n'est pas connue avant l'édition de liens. On ne peut donc pas fournir à un script d'édition de liens les contraintes adéquates, puisque celles-ci ne sont connues qu'après sa complétion. Une dépendance cyclique émerge ainsi dans la plupart des cas, qui doit être résolue pour implémenter ce système de partitionnement spatial.

Une manière d'éviter cette dépendance cyclique consiste à utiliser uniquement des pages mémoires de taille fixe. Cela force une utilisation suboptimale de la pagination, dégradant les performances à l'exécution (c.-à-d. que plus de pages doivent être utilisées

pour couvrir une zone mémoire plus importante). Une autre consisterait à spécifier les plages d'adresses à utiliser à l'éditeur de liens : leurs tailles étant ainsi connues à l'avance, il devient possible d'exploiter les pages de taille variable et de les protéger. Cette méthode requiert cependant une expertise humaine conséquente, faisant intervenir un processus fastidieux, possible source d'erreurs. Ainsi, la méthode que nous proposons consiste à briser la dépendance cyclique problématique de manière automatique et contrôlée.

La vue d'ensemble de notre méthode est proposée en figure 4.2. Elle est itérative et convergente. On désigne par « générateur » un programme dont l'écriture dépend des contraintes matérielles vues en section 4.3.2, et des possibilités du script d'édition de liens évoquées en section 4.3.1. Le générateur prend en entrée une configuration de partitionnement spatial, conforme à l'abstraction décrite en section 4.3.3. Après la première itération, l'exécutable binaire issu de la passe précédente sert également d'entrée. Ce générateur a pour objectif de générer un script d'édition de liens et un fichier objet contenant des tables de protection mémoire, à l'aide du compilateur sélectionné sur étagère. Ces produits servent d'entrées à l'éditeur de liens, qui récupère également les fichiers objets de l'applicatif logiciel, afin de produire un exécutable. Une fois cet exécutable finalisé (c.-à-d. qu'il contient une table de protection autodescriptive) il peut être soumis à un processus de validation permettant de garantir que sa génération est conforme vis-à-vis des contraintes de partitionnement exprimées par l'utilisateur.

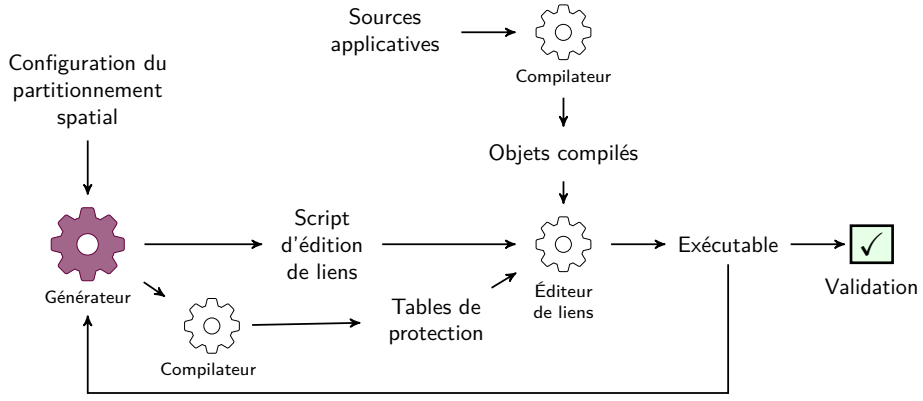


Figure 4.2 – Méthode générale de partitionnement spatial, agnostique d'une chaîne de compilation sur étagère. Elle permet la génération de tables de protection mémoires autodescriptives embarquées dans l'exécutable à protéger. Une implémentation de cette méthode est détaillée en section 4.5.

Dans le cas contraire, une nouvelle itération est lancée et le générateur dispose alors de l'exécutable précédemment généré, pour analyse. Avant que la première édition de liens ne soit effectuée, il est nécessaire de générer des tables de protection, afin que cette opération puisse s'effectuer (autrement, l'éditeur de liens rencontrerait un symbole non défini et lèverait une erreur). Aussi, en première instance, un symbole factice est généré, pointant sur des données sans aucune signification (les valeurs contenues dans la table n'étant pas encore connues). Lors de l'itération $n+1$, le générateur est à même d'extraire

les résultats du placement mémoire de l'itération n, *via* des analyseurs d'exécutables tels que LIEF [Tho17]. Ces nouvelles informations fournissent au générateur davantage de contexte permettant de générer des tables de protection mises à jour. Ce processus itératif termine quand les structures de données générées contiennent toutes les adresses et tailles des domaines à protéger, incluant le domaine contenant les tables elles-mêmes.

Des contraintes matérielles faibles peuvent simplifier significativement l'implémentation de cette méthode, mais nous nous concentrons ici sur le cas général. La section 4.5 présente un cas complexe d'implémentation permettant d'éprouver la robustesse de cette approche.

4.3.5 Stratégie de validation du partitionnement spatial

Les tables de protection mémoire générées grâce à la méthode que nous avons présentée précédemment peuvent être considérées, du point de vue de la norme DO-178C, comme des PDI [2], c'est-à-dire une configuration embarquée sur cible influençant l'exécution du système, sans modifier son code exécutable [RTC11a]. Ces tables sont typiquement stockées dans le binaire dit de « runtime » évoqué en section 3.1.2. Des analyseurs binaires (p. ex. LIEF [Tho17]) sont capables d'extraire ces informations depuis l'exécutable final. Celles-ci peuvent être exploitées par des outils dédiés (qui doivent être qualifiés [RTC11b]), capables de montrer que les consignes d'entrée du processus de génération ont été correctement traduites lors de la production de l'exécutable final. La suite *Asterios Checker* développée à Krono-Safe [MOT20] implémente cette fonctionnalité. Cela implique que le générateur lui-même ne nécessite pas d'être soumis à un processus de certification ou de qualification, car la vérification de ses sorties par rapport à ses entrées est suffisante à garantir sa validité.

La bonne exécution du placement mémoire peut être vérifiée en constatant que l'agencement des adresses dans l'exécutable final sont conformes aux spécifications inscrites dans la configuration de l'abstraction de placement décrite en section 4.3.3. Il est intéressant de noter que travailler sur cette abstraction permet de factoriser les algorithmes de validation : si certaines portions sont forcément inhérentes à la cible matérielle, la plupart en reste agnostique, ce qui simplifie l'effort de développement de l'outil de validation.

D'un point de vue industriel, cette démarche présente l'avantage de grandement réduire les efforts de certification. En effet, il est moins coûteux de valider des données générées hors ligne que de développer du code embarqué conforme aux exigences strictes de certification.

4.4 Implémentation pour la plateforme MPC5777M

Nous avons présenté en section 4.3 une méthode de partitionnement spatial en réponse aux considérations détaillées en section 4.1. Pour la suite de nos travaux, il est nécessaire de l'implémenter sur les cibles matérielles qui auront retenu notre attention. Nous décrivons dans cette section celle pour le MPC5777M, déjà abordé en section 3.5. Le P2020, qui a aussi étudié, est présenté en section 4.5.

2. *Parameter Data Items*, détaillés au §2.5.1 de la DO-178C.

4.4.1 Sélection de la chaîne de compilation

Le cadre industriel de nos travaux, détaillé au chapitre 3, contraint l'utilisation d'une chaîne de compilation : *Wind River Diab Compiler*. Cette chaîne outillée comprend un compilateur C (`dcc`) et un éditeur de liens (`dld`) reproductible, configurable *via* un script d'édition de liens. Il valide ainsi toutes les contraintes listées en section 4.3.1.

4.4.2 Analyse du matériel de gestion de la mémoire

Le MPC5777M dispose d'une MPU qui comporte vingt-quatre descripteurs mémoires permettant de fixer des permissions à autant de zones mémoires différentes, comme suit :

— six régions d'instructions de tailles arbitraires ;

— douze régions de données de tailles arbitraires ;

— six régions programmables (données ou instructions) de tailles arbitraires.

Les entrées MPU sont accédées *via* des registres MPU dédiés (*MPU Assist Registers*, ou `MAS`), qui peuvent être écrits grâce à des instructions dédiées. Dans cette implémentation, seuls deux parmi les quatre mis à disposition sont nécessaires : `MAS0` et `MAS1`. Ils permettent notamment d'indiquer les entrées à réserver, de spécifier les permissions que les descripteurs doivent appliquer ainsi que les tâches concernées par ces restrictions.

4.4.3 Génération des descripteurs mémoires

Les descripteurs mémoires doivent contenir les plages d'adresses à protéger et les valeurs pré-calculées à stocker dans les registres `MAS0` et `MAS1`. Comme la MPU du MPC5777M ne requiert aucune contrainte d'alignement, il est trivial d'obtenir les adresses à protéger. En effet, le script d'édition de liens peut être généré pour contenir deux symboles supplémentaires par région mémoire à protéger : l'un marquant son début et l'autre sa fin, comme montré en figure 4.3. L'éditeur de liens peut ainsi effectuer lui-même la résolution d'adresses par ces symboles, permettant ainsi l'écriture des descripteurs MPU directement depuis un langage de programmation de haut niveau, comme montré en figure 4.4.

```
1 .kernel_code_init (TEXT) ALIGN(8) :
2 {
3   __start_kernel_init = . ;
4   kernel_mpu_enable.o (".text*");
5   /* [...] */
6   __end_kernel_init = . ;
7 }
```

Figure 4.3 – Extrait de script d'édition de liens `dld` généré pour le MPC5777M.

L'implémentation du partitionnement spatial pour MPC5777M est ainsi résumée en figure 4.5. La simplicité du matériel et des contraintes associées permet d'aboutir à une solution plus simple que la méthode générale (figure 4.2) : une unique itération est

```c
struct kernel_mpu_entry
{
  const uint32_t mas0;
  const uint32_t mas1;
  const void *const start_address;
  const void *const end_address;
};

/* Symboles définis depuis le script d'édition de liens */
extern const char __start_kernel_init[];
extern const char __end_kernel_init[];
/* [...] */

/* Table d'entrées MPU */
static const struct kernel_mpu_entry kernel_code_entries[] =
{
  {
    .mas0 = /* valeur pré-calculée */,
    .mas1 = /* valeur pré-calculée */,
    .start_address = __start_kernel_init,
    .end_address = __end_kernel_init,
  },
  /* [...] */
};
```

Figure 4.4 – Création des descripteurs MPU grâce à la génération de structures C.

nécessaire, mais suffisante. Le générateur est chargé de calculer les valeurs des registres `MAS0` et `MAS1` et génère un script d'édition de liens et les tables de protection mémoire de concours : les tables référencent des symboles que le script d'édition de liens définit. Les sorties du générateur sont ainsi directement fournies à l'éditeur de liens avec les sources applicatives compilées, ce qui permet la génération d'un exécutable contenant des tables de protection mémoire le décrivant.

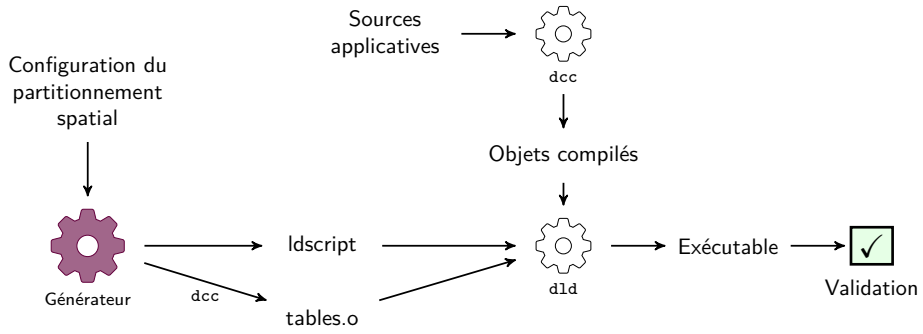


Figure 4.5 – Implémentation du partitionnement spatial pour MPC5777M.

4.5 Implémentation pour la plateforme P2020

Dans cette section, nous présentons une implémentation de la méthode décrite en section 4.3 pour la plateforme P2020, un SoC PowerPC de la famille des QorIQ du fabricant NXP [Fre12]. L'implémentation sur P2020 est bien plus complexe que celle sur MPC5777M montrée en section 4.4, de par le matériel mis à disposition sur cette plateforme. Sa validation expérimentale est détaillée plus tard, en section 4.6.

4.5.1 Présentation du P2020

La plateforme QorIQ P2020 du fabricant NXP se base sur deux cœurs PowerPC e500v2 [NXP11]. Son schéma d'architecture est montré en figure 4.6. On note la présence d'une mémoire partagée unique (DDR2 ou DDR3) à laquelle tous les cœurs accèdent *via* le bus système, en passant par le module de cohérence. Notons que chaque cœur dispose de caches de données et d'instructions séparés (L1) et qu'ils se partagent un cache L2. Nous ne détaillons pas davantage l'architecture du P2020 car nous nous intéressons plus particulièrement aux mécanismes de protection mémoire, qui sont décrits en section 4.5.3.

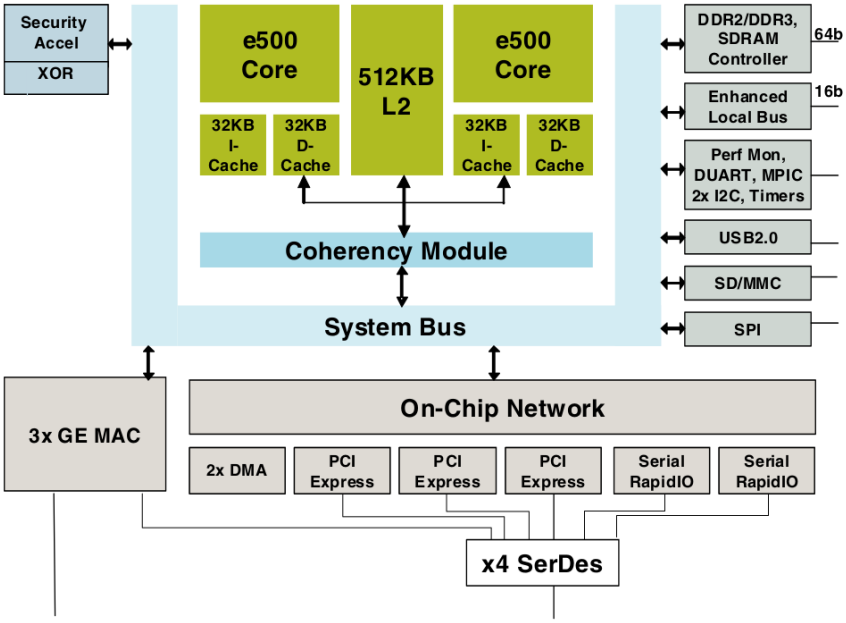


Figure 4.6 – Schéma d'architecture du P2020. Figure extraite du guide de développement du P2020 DS [Fre09, p. 4].

4.5.2 Sélection de la chaîne de compilation

La chaîne de compilation retenue pour nos travaux sur P2020 est identique à celle utilisée sur MPC5777M, décrite en section 4.4.1. Elle est justifiée par le cadre industriel de nos travaux.

4.5.3 Analyse du matériel de gestion de la mémoire

Les différents composants intervenant dans la traduction d'adresses du P2020 sont montrés en figure 4.7. La protection mémoire est implémentée par une MMU et chaque cœur dispose de deux niveaux de cache MMU [3]. Les caches de premier niveau (les pages de données et d'instructions sont séparées) sont gérés exclusivement par le matériel et sont donc exclus de nos travaux, puisqu'ils ne sont pas contrôlables par le logiciel. Les caches de second niveau sont unifiés (ils contiennent à la fois des pages de données et d'instructions) et disposent de deux TLB (TLB_0 et TLB_1), qui sont exclusivement contrôlées par du logiciel utilisateur. Si les tailles des pages sont fixées à 4 kB pour la TLB_0, celles-ci sont de tailles variables pour la TLB_1 (de 4 kB à 4 GB). Ces pages doivent être alignées sur leur taille.

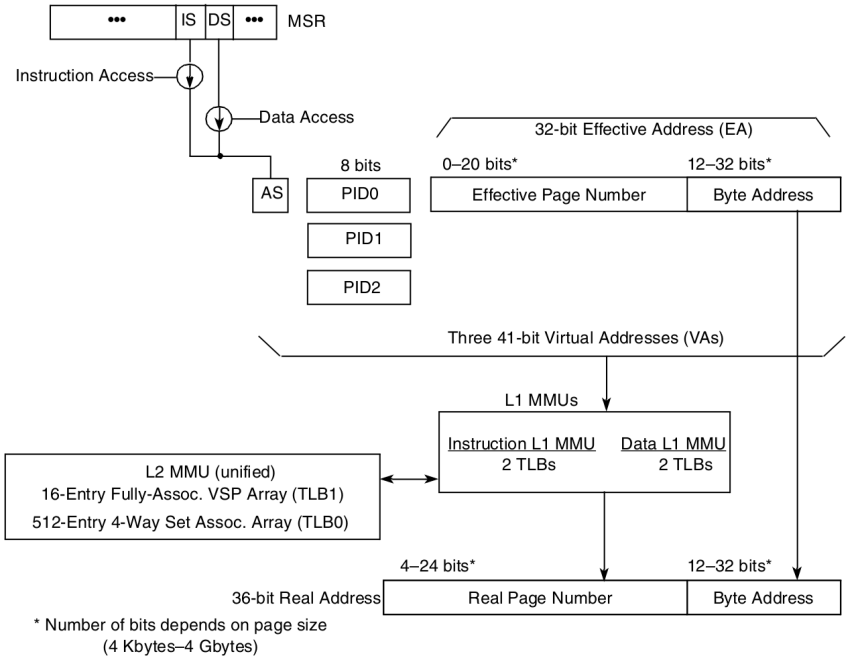


Figure 4.7 – Description des mécanismes de traduction d'adresse du P2020. La MMU y apparait comme un élément central avec ses différents niveaux de caches ainsi que les TLB programmables TLB_0 et TLB_1. Figure extraite du manuel de référence du cœur PowerPC e500v2 [NXP11, p. 12-5].

3. Nous insistons sur le fait que ces deux caches sont internes à la MMU et sont dédiés à la gestion des pages mémoires. Il ne s'agit pas des mémoires caches complémentant la mémoire principale.

Les pages MMU peuvent être chargées dans les TLB grâce à des registres MMU dédiés (*MMU Assist Registers*, ou `MAS`). Après la configuration des `MAS`, l'exécution de l'instruction `tlbwe` [4] permet le chargement effectif des pages mémoires. Au moment des changements de contextes, les entrées des TLB sont expulsées grâce à l'opération *Flash Invalidate*, déclenchée par une écriture de registre `MAS`. Ces entrées sont utilisées par les différents contextes d'exécution des applications Asterios, permettant la mise à jour des droits d'accès mémoires adéquats. À noter que les entrées utilisées par le noyau doivent être chargées dans les TLB à tout instant. En effet, dans le cas d'une interruption ou d'un appel système, le code basculera sur un vecteur d'interruption, sans permettre l'opportunité d'effectuer un changement de permissions, bloquant l'exécution du code noyau. Ainsi, les entrées du noyau sont chargées en TLB_1 avec un bit spécial [5] pour éviter leur invalidation.

4.5.4 Identification des données nécessaires à l'exécution

Les TLB programmables (TLB_0 et TLB_1) sont hétérogènes. Aussi, le logiciel permettant leur gestion doit disposer de différentes informations à l'exécution.

Utilisation de la TLB_0

La TLB_0 dispose de pages de taille fixe de 4 kB et de 512 entrées ; aussi un domaine mémoire pourra être couvert par plusieurs pages de 4 kB. Il peut être ainsi nécessaire de générer plusieurs descripteurs de pages par domaine. Comme la TLB_0 est associative par voie et par ensemble [6], il est nécessaire de connaître l'adresse finale de chaque page afin de déterminer dans quel ensemble du cache sera chargée la page mémoire. C'est seulement alors qu'il sera possible de calculer l'index de la voie correspondante. Les informations suivantes doivent ainsi être mises à disposition du noyau en charge de charger les descripteurs mémoires à chaque changement de contexte ; elles sont stockées en tant que structures constantes et contiennent :

— l'adresse de début du domaine à protéger ;
— son statut vis-à-vis des politiques de mémoire cache (p. ex. *write-through*, *cache-inhibited*, *etc.*) ;
— les permissions mémoires ;
— le nombre de pages de 4 kB à créer ;
— pour chaque page à créer, la valeur *Entry Select*, indiquant l'ensemble et la voie du cache de pages à utiliser.

Utilisation de la TLB_1

La TLB_1 dispose de moins d'entrées que la TLB_0, mais supporte des pages de tailles variables, ce qui permet à un domaine d'être protégé par une seule page mémoire. De manière similaire à la TLB_0, les informations suivantes doivent être embarquées pour être utilisées par le noyau :

4. *TLB write entry*.
5. `IPROT` : *invalidation protected*.
6. En anglais : *set/way associative*.

— l'adresse de début du domaine à protéger ;
— son statut vis-à-vis des politiques de mémoire cache (p. ex. *write-through*, *cache-inhibited*, *etc.*) ;
— les permissions mémoires ;
— la valeur $n \in [\![1; 11]\!]$, telle que la taille de la page mémoire soit de 4^n kB ;
— pour chaque page à créer, la valeur *Entry Select*, indiquant l'index de l'entrée au sein de la TLB_1 ;
— une valeur booléenne indiquant si une page mémoire peut être expulsée du cache de pages.

4.5.5 Génération des descripteurs mémoires

Les descripteurs mémoires doivent contenir les adresses, tailles et valeurs *Entry Select* de tous les domaines mémoires à protéger, ce qui inclut le domaine mémoire les contenant. Afin de collecter les diverses informations requises à l'exécution telles que décrites en section 4.5.4, nous présentons ici une possible implémentation de la méthode plus générale présentée en section 4.3. Au vu des contraintes de la plateforme P2020, nous avons élaboré une chaîne de quatre opérations d'édition de liens, comme montré en figure 4.8. Nous détaillons chacune de ces opérations pour en expliquer l'importance.

Première édition de liens : déduction des tailles des domaines applicatifs

Avant qu'une première édition de liens ait pu compléter, aucune information ne permet de générer des tables de protection, car aucune adresse ou taille n'est encore connue. Il est néanmoins requis que ces tables *existent*, afin que le symbole associé soit défini, et ce, afin de permettre à l'édition de liens de compléter sans erreurs. Aussi, des premières tables *vides* sont générées, pour qu'un symbole soit créé. L'exécutable ainsi généré ne permet pas d'être exécuté, mais il peut être inspecté : les tailles des sections de sortie (et donc des domaines) peuvent être déduites par une analyse du binaire. On peut ainsi obtenir les informations suivantes :

— la taille de chaque domaine applicatif, à l'exception du domaine couvrant les tables de protection mémoire ;
— le nombre total de pages à utiliser.

Deuxième édition de liens : déduction des tailles de chaque domaine

Des résultats de l'étape précédente, il est possible de connaître le nombre d'entrées à réserver dans la TLB_0 et dans la TLB_1. Cela permet de figer la taille des tables de protection, puisque le nombre total d'entrées à allouer est connu. Leur contenu reste toutefois encore inconnu, puisque les adresses n'ont pas pu être déterminées avec certitude. En effet, la deuxième édition de liens s'effectue en modifiant la taille d'un des objets précédemment utilisés. Nous privilégions l'hypothèse que cette modification peut avoir un impact sur l'allocation des adresses par l'éditeur de liens, ne pouvant démontrer l'inverse. À cette étape, comme les tables sont creuses, l'exécutable généré est encore invalide. À l'issue de cette opération, une inspection du binaire permettra l'extraction de la taille effective des tables de protection mémoire.

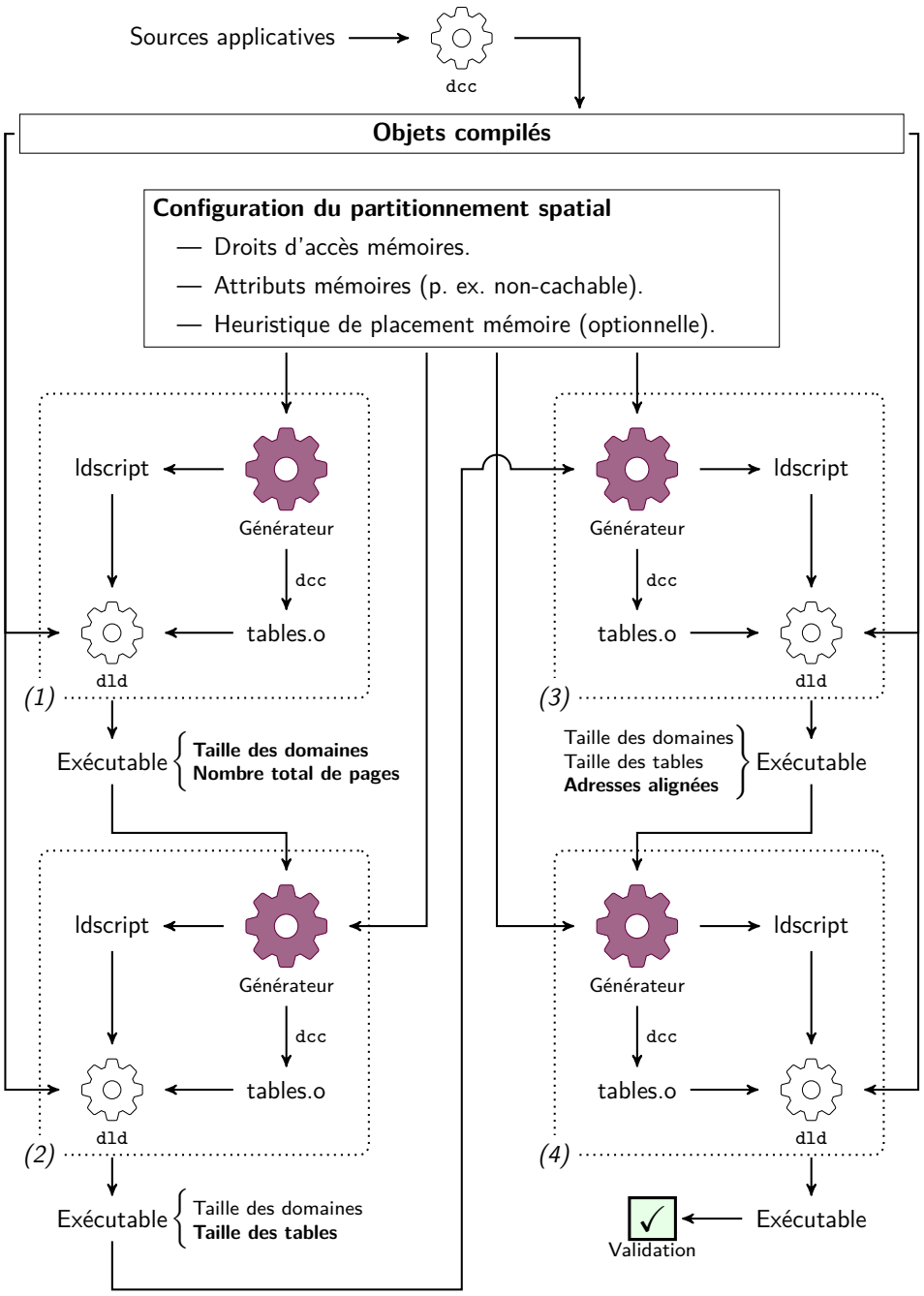


Figure 4.8 – Implémentation de la méthode générale en figure 4.2 pour la plateforme P2020, basée sur un enchaînement de quatre éditions de liens.

Troisième édition de liens : adresses de chaque domaine

Toutes les tailles étant connues à l'issue de l'étape précédente, il est désormais possible pour le générateur de produire un script d'édition de liens alignant chaque domaine sur sa taille. Cela aura pour effet de figer définitivement l'allocation mémoire. Toutefois, les adresses des domaines étant encore inconnues, les tables ne peuvent être générées avec des informations relatives au placement. Ce n'est qu'après cette troisième édition de liens que les valeurs des adresses sont connues.

Quatrième édition de liens : exécutable final

Après la troisième édition de liens, les adresses finales alignées sur la taille des domaines à protéger ont pu être récupérées après analyse du binaire. Toutes les informations de protection peuvent être renseignées dans les tables de protection qui sont générées avec des valeurs exactes. Cette dernière édition de liens permet la génération d'un binaire contenant des tables de protection autodescriptives. Ce binaire final est ensuite soumis à un processus de validation permettant de vérifier que l'agencement mémoire est bien en accord avec la spécification de partitionnement spatial d'entrée.

4.6 Validation expérimentale

Nous reprenons ici l'implémentation de notre méthode pour la cible P2020, présentée en section 4.5. L'objectif est de montrer un exemple concret illustrant la configuration du partitionnement spatial d'entrée ainsi que les multiples résultats intermédiaires jusqu'à l'exécutable final. Nous montrons ici ce mécanisme appliqué au RTOS, il est similaire pour l'application que l'utilisateur compile *via* la chaîne outillée présentée en section 3.1.2.

4.6.1 Configuration de partitionnement spatial

La configuration du partitionnement spatial implémentée dans Asterios s'exprime au format JSON [ISO17] [7]. On choisit de placer le RTOS en L2/SRAM : le cache L2 du P2020 peut être reconfiguré pour être utilisé comme une mémoire SRAM adressable. Nous utilisons ainsi une unique *région* pour la décrire. On considère une taille de 512 MB (soit 0x80000 octets) et son adresse physique de début est 0x80000000. Pour ne pas présenter l'intégralité du RTOS, on se limite à deux « modules » de code exécutable :

1. le code du RTOS (c.-à-d. les instructions qu'il exécute) ;
2. du code fourni par le RTOS que les agents exécutent (une sorte de bibliothèque, appelée « libpsy »).

Le premier doit recevoir les droits d'exécution seule, mais les agents ne sont pas autorisés à y accéder. Le second est exécutable par les agents, mais pas par le noyau [8]. On décrit également un domaine contenant des données en lecture seule, car il est amené à contenir les tables de protection mémoire.

7. Dans ces travaux, nous nous autorisons une déviation par rapport au standard en utilisant des représentations hexadécimales de valeurs entières.

8. Par souci de clarté, on omet ici le contexte d'exécution « psyslayer » mentionné en section 3.4.1.

```
{
  "regions": [
    {
      "name": "l2sram",
      "physical_address": 0x80000000,
      "size": 0x80000,
      "domains": [
        {
          "protection": {
            "kernel": "EXECUTE",
            "all_agents": "NO_ACCESS",
          },
          "output_sections": [
            {
              "name": ".kernel_code",
              "type": "CODE"
            }
          ]
        },
        {
          "protection": {
            "kernel": "NO_ACCESS",
            "all_agents": "EXECUTE",
          },
          "output_sections": [
            {
              "name": ".libpsy_code",
              "type": "CODE"
            }
          ]
        }
        {
          "protection": {
            "kernel": "READ_ONLY",
            "all_agents": "NO_ACCESS",
          },
          "output_sections": [
            {
              "name": ".kernel_rodata",
              "type": "RODATA"
            }
          ]
        }
      ]
    }
  ]
}
```

Figure 4.9 – Description d'un partitionnement mémoire du code du RTOS et de la « libpsy ». Un domaine de données en lecture seule est précisé afin de placer les tables de protection mémoire.

On peut ainsi écrire un premier fragment de configuration, comme en figure 4.9. Trois domaines sont utilisés pour décrire les contraintes de protection mémoire et

chaque domaine contient ici une unique section de sortie. Celles de type « `CODE` », indiquent qu'il s'agit d'instructions — cette information est susceptible d'être utilisée par le script d'édition de liens. Ainsi, les sections de sortie `.kernel_code` et `.libpsy_code` contiennent-elles le code exécutable qui nous intéresse ici et la section de sortie `.kernel_rodata` contient les données en lecture seule du RTOS. Aussi, cette figure décrit la partie droite de la figure 4.1. Les définitions des *groupes* sont renseignées dans la figure 4.10, qui montre un fragment de configuration s'ajoutant à celui de la figure 4.9.

```
{
  "groups": [
    {
      "destination": ".libpsy_code",
      "name": "group_libpsy_code",
      "sources": [
        ".text"
      ]
    },
    {
      "destination": ".kernel_code",
      "name": "group_kernel_code",
      "sources": [
        ".text"
      ]
    },
    {
      "destination": ".kernel_rodata",
      "name": "group_kernel_rodata",
      "sources": [
        ".rodata"
      ]
    },
  ]
}
```

Figure 4.10 – Exemple de groupes associant les sections d'entrée d'objets compilés du RTOS aux sections de sortie décrites dans la configuration du partitionnement mémoire en figure 4.9.

Une fois les groupes renseignés, il suffit d'y associer des fichiers objets. Ainsi, leurs sections d'entrée exécutables sélectionnées par « `.text` » (dans notre cas) sont utilisées pour peupler les sections exécutables de sortie associées aux groupes. Un exemple partiel de cette association est montré en figure 4.11. Le RTOS contient bien plus de fichiers sources, mais nous n'en montrons ici qu'une partie par souci de clarté.

4.6.2 Génération de l'exécutable final

La combinaison des figures 4.9 à 4.11 décrit une configuration de partitionnement spatial. Nous détaillons ici les artefacts intermédiaires produits par l'enchaînement des quatre éditions de liens, comme décrit en figure 4.8 ainsi que le contenu de l'exécutable final.

```
{
  "objects": [
    {
      "id": "libpsy_vt.o",
      "groups": [
        "group_libpsy_code"
      ]
    },
    {
      "id": "kernel_rsf.o",
      "groups": [
        "group_kernel_code"
      ]
    }
    {
      "id": "protection_tables.o",
      "groups": [
        "group_kernel_rodata"
      ]
    }
  ]
}
```

Figure 4.11 – Exemple d'association d'objets aux groupes.

Script d'édition de liens

Les scripts d'édition de liens générés pour chacune des quatre éditions de liens sont sensiblement identiques : seules les contraintes d'alignement (directive `ALIGN`) changent — on les note « ??? ». Nous montrons en figure 4.12 des fragments choisis de ces fichiers et leur correspondance avec la configuration du partitionnement spatial. On constate la présence d'une section de sortie supplémentaire : celle qui héberge l'objet contenant les tables de protection mémoire. Il s'agit de données constantes résidant dans une section d'entrée `.rodata`. Notons la présence de symboles délimitant le début et la fin de chaque section. Ils sont utilisés pour déterminer les plages d'adresses à protéger.

Tables de protection mémoire

Contrairement à la solution proposée pour le MPC5777M (section 4.4), la MMU du P2020 requiert que les zones à protéger soient alignées sur les tailles des pages mémoires qui les protègent. Aussi, à la fin de la troisième édition de liens, nous sommes en mesure de fournir une adresse fixe — pour le MPC5777M, nous pouvions laisser à l'éditeur de liens le soin de calculer l'adresse lui-même.

La figure 4.13 montre un exemple de définition des tables de protection mémoire MMU. La variable globale `kernel_descriptor` contient les descripteurs des TLB_0 et TLB_1. Ici, seule la TLB_1 est utilisée, ce qui explique qu'il y n'ait aucun descripteur pour la TLB_0. Les trois descripteurs utilisés par la TLB_1 sont listés dans la variable `tlb1_descriptors`. Chaque entrée du tableau indique les valeurs requises pour mettre en œuvre la protection mémoire. Les champs se déclinent comme suit.

— `start_addr` indique l'adresse de début de la zone à protéger.

```
/* Description de la région mémoire */
MEMORY { l2sram : org = 0x80000000, len = 0x80000 }

SECTIONS
{
  /* La première section de sortie est placée au début de sa région */
  .kernel_code (TEXT) 0x80000000 ALIGN(???) :
  {
    __start_kernel_code = . ;
    "kernel_rsf.o" (".text")
    /* [...] */
    __end_kernel_code = . ;
  } > l2sram

  /* Les sections de sortie qui suivent sont placées automatiquement
   * à la suite si une adresse physique n'est pas précisée */
  .libpsy_code (TEXT)
    BIND(ADDR(.kernel_code) + SIZEOF(.kernel_code)) ALIGN(???) :
  {
    __start_libpsy_code = . ;
    "libpsy_vt.o" (".text")
    /* [...] */
    __end_libpsy_code = . ;
  }

  /* Données en lecture seule. Contient les tables de protection
   * mémoire. */
  .kernel_rodata (CONST)
    BIND(ADDR(.libpsy_code) + SIZEOF(.libpsy_code)) ALIGN(???) :
  {
    __start_kernel_rodata = .;
    "protection_tables.o" (".rodata")
    /* [...] */
    __end_kernel_rodata = .;
  }
}
```

Figure 4.12 – Extrait de script d'édition de liens `dld` correspondant à la configuration de partitionnement spatial présentée aux figures 4.9 à 4.11.

- `esel` indique le placement de la page dans la TLB_1.
- `tsize` renseigne sur la taille de la page mémoire à utiliser : les valeurs « 2 » et « 3 » représentent respectivement des tailles de 16 kB et 64 kB.
- `iprot` à « 1 » indique que la page ne peut être expulsée du cache de pages.
- La valeur `0x4` de `mas2_attr` indique que la cohérence mémoire doit être active sur la page mémoire.
- `mas3_attr` permet de spécifier les permissions mémoires comme suit :
  - `0x11` : lecture et exécution sont autorisés pour le mode superviseur ;
  - `0x22` : lecture et exécution sont autorisés pour le mode utilisateur ;
  - `0x1` : seule la lecture est autorisée pour le mode superviseur.

```
static const struct tlb1_descriptors tlb1_descriptors[] =
{
  /* Descripteur du code du noyau pour tous les agents et le noyau */
  {
    .start_addr  = 0x80000000,
    .esel        = 0,
    .tsize       = 3,
    .iprot       = 1,
    .mas2_attr   = 0x4,
    .mas3_attr   = 0x11,
  },
  /* Descripteur de la libpsy, pour tous les agents et le noyau */
  {
    .start_addr  = 0x80010000,
    .esel        = 1,
    .tsize       = 2,
    .iprot       = 1,
    .mas2_attr   = 0x4,
    .mas3_attr   = 0x22,
  },
  /* Descripteur des tables de protection mémoire */
  {
    .start_addr  = 0x80020000,
    .esel        = 2,
    .tsize       = 3,
    .iprot       = 1,
    .mas2_attr   = 0x4,
    .mas3_attr   = 0x1,
  },
};

/* Description globale du noyau. Chaque descripteur mémoire décrit
 * l'utilisation de la TLB0 et de la TLB1 */
const struct kernel_spatial_descriptor kernel_descriptor =
{
  .domains_tlb0       = NULL,
  .domains_tlb1       = tlb1_descriptors,
  .domains_tlb0_count = 0,
  .domains_tlb1_count = 3,
};
```

Figure 4.13 – Tables MMU de protection mémoire de l'exécutable final. Les adresses (alignées) de chaque zone à protéger sont connues, ainsi que leurs tailles.

Examen du binaire final

Une fois la dernière édition de liens effectuée, on examine le binaire généré avec l'outil `nm` fourni par les `binutils`. Ce programme liste les symboles ainsi que leurs types et adresses. Un extrait, en tableau 4.1, montre les adresses des symboles délimitant les régions mémoires à protéger.

L'étude de ces symboles permet de vérifier la cohérence du contenu des tables de protection mémoire écrites en figure 4.13. Les tailles à protéger sont ainsi conformes des valeurs `tsize`. Notons que la section `.kernel_rodata` est particulièrement large,

Symbole	Adresse	Type	Taille
`__start_kernel_code`	`0x80000000`	Absolu (A)	30 kB
`__end_kernel_code`	`0x80007804`	Absolu (A)	
`__start_libpsy_code`	`0x80010000`	Absolu (A)	4,8 kB
`__end_libpsy_code`	`0x80011310`	Absolu (A)	
`__start_kernel_rodata`	`0x80020000`	Absolu (A)	33,4 kB
`__end_kernel_rodata`	`0x800285c0`	Absolu (A)	

Table 4.1 – Extrait de certains symboles du binaire du RTOS.

car elle contient un tampon mémoire de 32 kB. Celui-ci sert au vidage des caches L1. Cet examen sommaire de l'exécutable généré permet de s'assurer de la cohérence de l'ensemble : la configuration du partitionnement mémoire est bien observée dans le binaire final.

4.7 Conclusion

Dans ce chapitre, motivés par le constat que le placement mémoire affecte le WCET de manière significative, nous avons élaboré une méthode de partitionnement spatial permettant de concilier placement et protection mémoire. La granularité en termes de placement et de protection permise est fine et s'ajuste aux contraintes de l'application. Nos travaux se plaçant dans un cadre industriel particulier, notre méthode prend en compte les exigences issues du monde de la sûreté de fonctionnement et les contraintes de certification. Son approche automatique permet de générer des tables de protections autodescriptives intégrées à l'exécutable final et de valider le partitionnement spatial de manière incrémentale. Cette technique permet de réduire les coûts liés à la certification. Elle a été éprouvée sur une variété de cartes matérielles et de chaînes de compilation et pleinement intégrée à la chaîne outillée Asterios. Nous avons montré ici un cas simple basé sur la MPU du MPC5777M et un cas plus complexe exploitant la MMU du P2020.

Dans la suite de nos travaux, nous nous reposons systématiquement sur la méthode décrite dans ce chapitre pour réaliser nos opérations de placement mémoire.

II

Deuxième partie

Analyse et réduction des interférences multicœurs

Chapitre 5

Principe de non-simultanéité

Sommaire

5.1 Motivations et positionnement

Nous avons évoqué au chapitre 2 les problèmes que posent l'utilisation des plate-formes multicœurs sur étagère au sein de systèmes temps-réels de sûreté. Ceux-ci sont provoqués par les accès simultanés à des ressources matérielles partagées (comme de la

mémoire partagée ou des périphériques) depuis plusieurs cœurs. Un aperçu de l'état de l'art des solutions visant à réduire ou éliminer ces interférences a déjà été présenté en section 2.4.3.

Nous avons décrit le modèle de calcul TCA en section 3.2 : une approche formalisée permettant d'associer des contraintes temporelles à des séquences de calculs. Comme vu précédemment, nous cherchons à mettre en œuvre une mitigation des interférences temporelles par conception, de sorte à ne présenter aucun caractère dynamique, et ce, afin de faciliter la certification des systèmes se basant sur nos travaux. L'idée principale est d'exploiter le paradigme *time-triggered* pour mettre en œuvre des sections critiques gérées par l'écoulement du temps. Ce mécanisme diffère des sections critiques utilisées classiquement dans les modèles de programmation impératifs et non temporisés [Ray13]. En effet, ici, les dates auxquelles les sections critiques débutent et s'achèvent sont toutes déterminées à la compilation, offrant ainsi des propriétés de sûreté additionnelles, telle que l'absence d'interblocage. Notons bien que cette approche repose principalement sur le *time-triggered* et non sur le TDM.

Contrairement aux approches existantes statiques qui s'appuient sur la génération d'un modèle d'exécution exhibant certaines propriétés choisies, nous préférons exprimer les contraintes d'exclusion temporelle au niveau du modèle de calcul, qui intervient plus en amont dans la conception du système. Cela nous permet de séparer les contraintes d'ordonnançabilité et d'exclusion temporelle en deux problèmes orthogonaux, résolus indépendamment. Grâce au principe de non-simultanéité que nous développons dans ce chapitre, nous pensons permettre aux concepteurs d'applications temps-réelles de sûreté d'exprimer une conception temporelle prenant en compte des contraintes d'exclusion temporelle, afin de mitiger, voire d'éviter, les interférences temporelles causées par des accès simultanés aux ressources matérielles partagées.

Le principe de non-simultanéité a fait l'objet d'une publication, acceptée et présentée à la conférence TIME 2020 [Guy+20a].

5.2 Expression de la simultanéité au sein des TCA

Nos travaux visent à intégrer la notion de simultanéité au modèle de calcul TCA. Pour cela, nous y rajoutons quelques notions, détaillées dans cette section.

Définition 9 (Simultanéité)**.** *Dans nos travaux, nous considérons que la simultanéité est appliquée à des fenêtres de calculs qui s'exécutent dans des intervalles bornés en temps. Ainsi, deux fenêtres d'exécution sont simultanées si leurs exécutions sont susceptibles de se recouvrir dans le temps.*

5.2.1 Transitions temporelles

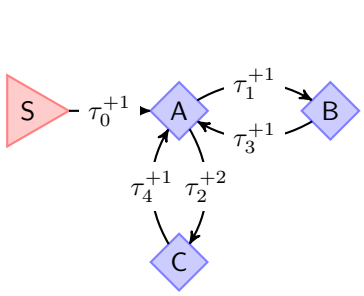
Considérons une horloge logique, structure causant l'avancement du temps global d'un système et associons à un TCA exactement une horloge logique. On définit une *transition temporelle* comme étant l'ensemble ordonné de blocs contraints par exactement une contrainte *after* et une contrainte *before*. Cette structure est associée à un intervalle de temps couvrant ces deux bornes, c'est-à-dire la différence entre la date de l'échéance (associée au *before*) et la date de début au plus tôt (associée à l'*after*).

Cette durée, notée t est par conséquent strictement positive, et s'exprime comme un nombre fini de tics de l'horloge logique. On note ces transitions temporelles τ^{+t}. Par souci d'économie, on peut simplifier cette notation par τ quand sa durée n'est pas une information indispensable.

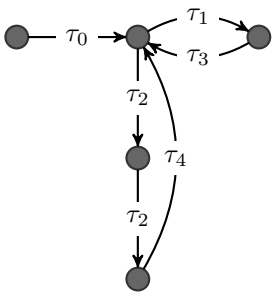
5.2.2 TCA isochrones

Considérons un TCA dans lequel chaque bloc est délimité par exactement une contrainte *after* et une contrainte *before*. Ceux-ci doivent alors inclure un nœud d'entrée unique ayant une contrainte *after*, tous les autres nœuds étant des synchronisations faisant partie d'un graphe connecté dans lequel chaque nœud dispose d'au moins un successeur. Ainsi, il existe au moins un cycle dans ce graphe et le TCA peut être trivial ou non trivial (cf. définition 5).

Ce type de TCA peut être transformé en TCA isochrone en découpant les transitions temporelles en fragments unitaires, de sorte que la somme des durées de chaque transition temporelle résultante soit égale à la durée de la transition temporelle d'origine. Dans les représentations graphiques sous-jacentes, on peut représenter chaque nœud par ●. La figure 5.1 illustre cette transformation où le TCA en figure 5.1a peut être exprimé sous la forme d'un TCA isochrone en figure 5.1b



(a) TCA non trivial constitué uniquement de transitions temporelles.



(b) TCA isochrone équivalent à celui de la figure 5.1a.

Figure 5.1 – Transformation d'un TCA en automate isochrone. On remarque en particulier que τ_2^{+2} est découpée en deux transitions temporelles unitaires τ_2.

5.2.3 Groupes d'exclusion

On définit une application contrainte en temps comme un ensemble de TCA isochrones partageant la même horloge logique de base. Plus spécifiquement, chaque tic de cette horloge logique indique la complétion d'une transition temporelle de chaque TCA isochrone faisant partie de l'application. On peut les considérer comme *synchrones* entre eux.

Une application contrainte en temps peut être associée avec un ensemble de *groupes d'exclusion*, où un groupe d'exclusion consiste en un ensemble de transitions temporelles ne devant pas s'exécuter simultanément. Ils permettent ainsi l'expression d'une propriété

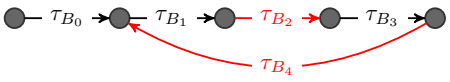
de sûreté [AS87] : « rien de mal ne peut arriver » [1]. Cette propriété doit être vérifiée pour chaque groupe d'exclusion sur le résultat de la composition de tous les TCA appartenant à une application contrainte en temps sur laquelle s'applique le groupe d'exclusion.

5.2.4 Exemple

Les figures 5.2 et 5.3 permettent de donner une intuition du mécanisme de simultanéité et de sa vérification. Deux TCA isochrones A et B, alloués à des cœurs d'exécution différents présentent certaines transitions temporelles d'intérêt appartenant au même groupe d'exclusion $G = \{\tau_{A_1}, \tau_{B_2}, \tau_{B_4}\}$.

Dans cet exemple, la conception temporelle des automates A et B laisse suggérer, comme en figure 5.3, que les transitions temporelles de G ne peuvent se recouvrir dans le temps quand A et B s'exécutent simultanément.

(a) TCA A alloué à un cœur c_A décrivant une tâche périodique : après l'exécution de τ_{A_0}, la séquence τ_{A_1} et τ_{A_2} est répétée.



(b) TCA B alloué à un cœur c_B décrivant une tâche périodique : après l'exécution de τ_{B_0}, la séquence τ_{B_1}, τ_{B_2}, τ_{B_3} et τ_{B_4} est répétée.

Figure 5.2 – Exemple d'application contrainte par le temps, composée de deux TCA isochrones triviaux. Notons τ_{A_1}, τ_{B_2} et τ_{B_4} les transitions temporelles ne devant pas se recouvrir dans le temps.

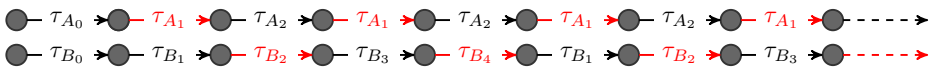


Figure 5.3 – « Dépliage » infini des automates A (dessus) et B (dessous) de la figure 5.2. Cela suggère un motif répétitif dans lequel les transitions temporelles du groupe d'exclusion $G = \{\tau_{A_1}, \tau_{B_2}, \tau_{B_4}\}$ ne peuvent se recouvrir dans le temps.

5.3 Formalisation du problème

Nous avons introduit en section 5.2 les notions d'applications contraintes en temps et des groupes d'exclusion permettant de spécifier la propriété de sûreté que les transitions temporelles qui les composent ne peuvent s'exécuter simultanément. Nous présentons ici des algorithmes permettant d'automatiser la vérification de cette propriété.

Les TCA peuvent présenter une combinaison virtuellement illimitée de comportements temporels, car les TCA n'ont, par défaut, pas de borne temporelle haute. Ainsi, les dates auxquelles sont déclenchées les transitions temporelles peuvent résulter en une infinité de séquences des cycles qui les composent. La figure 5.4 illustre cette explosion

1. « *Bad things do not happen during execution of a program.* »

des états possibles en dépliants ceux visités entre les tics zéro et sept du TCA montré en figure 5.1.

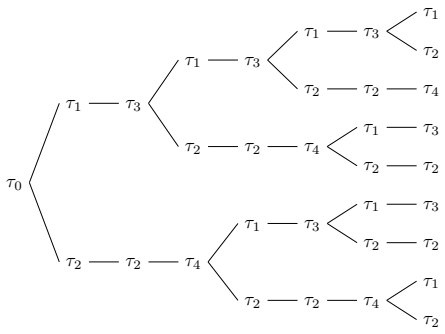


Figure 5.4 – Arbre des exécutions possibles « dépliées » des comportements temporels du TCA non trivial montré en figure 5.1 après sept tics d'horloge.

Comme les applications contraintes en temps sont composés de TCA isochrones qui partagent tous la même horloge logique, ils sont aussi *synchrones*. Ainsi, comme chaque tic provoque l'activation immédiate d'une transition temporelle par automate, cela implique que chaque transition peut être activée pour un nombre possiblement infini de dates (chaque date étant un tic d'horloge, donc un entier naturel). Par exemple, dans la figure 5.1a, τ_{A_0} n'est activable qu'à la date zéro, tandis que τ_{A_1} est activé pour toutes les dates impaires. Un TCA peut ainsi être considéré comme un automate fini dans lequel :

— chaque état (à l'exception de l'état initial) peut être marqué comme *acceptant* ;
— l'incrément de temps, associé à toutes les transitions temporelles peut être vu comme le symbole d'un *alphabet unaire* (grâce à la propriété d'isochronisme) ;
— l'ensemble de dates auxquelles un état peut être atteint est donné par l'ensemble des *tailles de mots* menant à cet état. À noter que cet ensemble peut être infini, si l'état fait partie d'un cycle.

Par voie de conséquence, l'ensemble des dates auxquelles un état peut être atteint s'exprime comme le langage régulier d'un alphabet unaire, accepté par l'automate où seulement cet état est marqué comme acceptant. Nous pouvons ainsi utiliser un résultat de la théorie des langages : l'expression d'un langage unaire régulier s'exprime comme l'union de suites arithmétiques que l'on note $\{c + dk | k \in \mathbb{N}\}$ où c et d sont deux constantes positives [Saw13]. On peut également l'exprimer comme la paire notée (c, d).

Les transitions temporelles sont délimitées par des états de l'automate. Aussi, les dates auxquelles une transition temporelle donnée est activée correspond à l'ensemble des dates auxquelles son état d'origine est atteint.

5.4 Dates d'activation des transitions

Nous avons montré en section 5.3 que déterminer l'ensemble des dates auxquelles une transition temporelle s'activait était équivalent à déterminer l'expression d'un

langage régulier unaire sur un automate acceptant uniquement l'état de début de cette transition. Nous nous appuyons maintenant sur un algorithme de l'état de l'art proposé par SAWA pour trouver les solutions de ce problème, dans un cas général, sans faux négatif ni faux positif [Saw13].

5.4.1 Notations

Soit un automate fini unaire et non déterministe, ou UNFA (Unary Non-deterministic Finite Automaton), noté $\mathcal{A}$ possédant $n \geq 2$ états et m transitions, tel que $\mathcal{A} = (Q, \delta, I, F)$. Q est l'ensemble des états ($|Q| = n$), $\delta \subseteq Q \times Q$ désigne la relation de transition d'états, $I \subseteq Q$ est l'ensemble des états initiaux de l'automate et $F \subseteq Q$ est l'ensemble des états acceptants.

Nous réutilisons les notations de [Saw13] : $q \xrightarrow{x} q'$ indique qu'il existe un chemin de taille x de $q \in Q$ à $q' \in Q$. Sur un UNFA, un chemin de taille x correspond à un mot x de sorte qu'un mot de taille x soit accepté par $\mathcal{A}$ s'il existe un chemin de taille x de $q_i \in I$ à $q_f \in F$. Ainsi, le langage $\mathcal{L}(\mathcal{A})$ accepté par $\mathcal{A}$ est l'ensemble de tous les mots acceptés par $\mathcal{A}$.

5.4.2 Expression du langage unaire

Dans [Saw13], l'algorithme *UNFA-Arith-Progessions* permet de construire à partir d'un UNFA $\mathcal{A}$ l'ensemble des suites arithmétiques $\mathcal{R}$ exprimant le langage $\mathcal{L}(\mathcal{A})$. Il opère avec une complexité spatiale en $O(n+m)$ et une complexité algorithmique en $O(n^2(n+m))$.

Appliquer cet algorithme aux TCA isochrones permet d'obtenir l'ensemble exhaustif de dates auxquelles un nœud temporel est atteint. L'essence de cet algorithme réside dans l'expression d'un chemin α de $q_i \in I$ à $q_f \in F$ passant par $q \in Q$ de sorte que $q_i \xrightarrow{c_1} q \xrightarrow{c_2} q_f$. Si q appartient à un cycle de taille d, alors la taille de α s'exprime comme la paire $(c_1 + c_2, d)$, ou $(c_1 + c_2, 0)$ dans le cas contraire. Ainsi, $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ avec $|\mathcal{R}_1| \leq n^2$ et $|\mathcal{R}_2| \leq n$. $\mathcal{R}_1$ contient tous les mots de taille $x < n^2$ écrits $(x, 0)$ tandis que $\mathcal{R}_2$ contient tous les autres mots de $\mathcal{L}(\mathcal{A})$ (avec $x \geq n^2$) et est exprimé comme un ensemble de suites arithmétiques (au plus n).

5.4.3 Adaptation de l'algorithme UNFA-Arith-Progessions

Appliquer l'algorithme *UNFA-Arith-Progessions* pour chacun des $n-1$ états pouvant être marqués comme acceptant [2] conduirait à une complexité algorithmique totale en $O(n^3(n+m))$. Nous proposons ici une modification de cet algorithme pour déterminer l'ensemble des dates auxquelles les transitions temporelles sont atteignables tout en préservant la complexité algorithmique originelle.

— Pour $q \in Q$, $sl(q)$ est défini comme la taille du cycle le plus court incluant q. Si q ne fait partie d'aucun cycle, alors $sl(q)$ n'est pas défini.

— Un état q est dit *important* si q appartient à une composante fortement connectée C (ce qui implique que $sl(q)$ est défini) et $sl(q)$ est minimal pour chaque état de C.

2. L'état initial unique des TCA ne peut jamais être atteint depuis un autre état.

— Les ensembles S_i sont calculés de telle manière que chaque ensemble contienne tous les états atteignables depuis un état initial s en i étapes : $S_i = \{q \in Q : s \xrightarrow{i} q\}$ avec $i \in [0, n^2[$.

— Soit Imp l'ensemble des états importants de $\mathcal{A}$.

— Soit $Q_{imp} = S_{n-1} \cap Imp$ les états importants pouvant être atteints après exactement $n-1$ étapes depuis l'état initial s.

— Soit $D = \{sl(q) | q \in Q_{imp}\}$ l'ensemble des plus petites tailles de cycles parmi les états importants de Q_{imp}.

— Étant donné que les TCA isochrones ne disposent que d'un seul état initial, I peut être écrit $I = \{s\}$.

— Nous redéfinissons F comme l'ensemble d'états pouvant être marqués comme acceptant. Ainsi : $F = Q \backslash \{s\}$.

— Nous définissons $\mathbb{T}_q$ comme l'ensemble des transitions temporelles pouvant être activées à l'état q (c'est-à-dire les transitions sortantes de q).

En utilisant la définition des TCA isochrones, nous pouvons exprimer $\mathcal{R}_1$ tel que $\mathcal{R}_1 = \{(i, 0) | i \in [1, n^2[\}$. Cela permet de construire un premier ensemble de dates auxquelles les états acceptant sont atteignables. Dans ce cas, nous pouvons réutiliser cette formule afin de déterminer l'ensemble des dates pour lequel une transition temporelle τ est atteignable : $\mathcal{D}_{1,\tau}$ comme montré dans l'algorithme algorithme 1.

Algorithme 1 : Construction de l'ensemble de dates $\mathcal{D}_{1,\tau}$.

```
for τ ∈ T_s do
    D_{1,τ} = {(0, 0)}
for i ∈ [1, n²[ do
    for q ∈ S_i do
        for τ ∈ T_q do
            D_{1,τ} = D_{1,τ} ∪ {(i, 0)}
```

Comme la formule d'origine exclut l'état initial, nous rajoutons que les transitions atteignables depuis l'état initial sont toutes atteignables à la date zéro (par définition). Ensuite, nous rajoutons les transitions temporelles activées à un état q, puisque chaque état est associé à un ensemble de dates.

Le second ensemble de dates $\mathcal{R}_2$ est construit grâce aux ensembles T_i contenant tous les états à partir desquels un état final (acceptant) peut être atteint en i étapes, comme en équation (5.1). La paire $(c' + n - 1, d)$ est ensuite ajoutée à $\mathcal{R}_2$ pour $c' \in [n^2 - 2n; n^2 - n - 1]$ et chaque $d \in D$ tel que $c' \geq n^2 - n - d$ s'il existe $q \in Q_{imp}$ avec $sl(q) = d$ pour $q \in T_{c'}$.

$$T_i = \{q \in Q | \exists q_f \in F : q \xrightarrow{i} q_f\} \text{ avec } i \in [0, n^2 - n - 1] \tag{5.1}$$

Une conséquence de cette formulation dans l'algorithme *UNFA-Arith-Progessions* contraint que les différentes transitions temporelles amenant à $q_f \in F$ sont entremêlées dans la construction des ensembles T_i en équation (5.1). Afin de préserver les

dates spécifiques aux transitions temporelles, nous proposons de créer un ensemble les discriminant, comme en équation (5.2).

$$T_{i,q_f} = \{q \in Q : q \xrightarrow{i} q_f\} \text{ avec } i \in [0, n^2 - n - 1] \text{ et } q_f \in F \tag{5.2}$$

En considérant chaque $q_f \in F$ et chaque $\tau \in \mathbb{T}_{q_f}$, le même algorithme peut être utilisé pour calculer $\mathcal{D}_{2,\tau}$ comme montré en algorithme 2 en substituant T_i par T_{i,q_f}. Enfin, l'ensemble de dates $\mathcal{D}_\tau$ caractérisant l'ensemble des dates auxquelles la transition temporelle τ est atteignable est déterminé comme $\mathcal{D}_\tau = \mathcal{D}_{1,\tau} \cup \mathcal{D}_{2,\tau}$.

Ainsi, au lieu d'exécuter l'algorithme d'origine pour les $n-1$ états acceptants, nous construisons les ensembles T_{i,q_f} une seule fois. De plus, leur construction ne requiert pas plus d'opérations que pour les ensembles T_i : seule l'organisation des données change. Ceci nous permet de préserver la complexité arithmétique de base ($O(n^2(n+m))$) puisque l'algorithme 2 ne l'accroît pas.

Algorithme 2 : Construction de l'ensemble de dates $\mathcal{D}_{2,\tau}$.

```
for q ∈ Q_imp do
    for c' ∈ [n² − 2n, n² − n − 1] do
        for q_f ∈ F do
            if q ∈ T_{c',q_f} and c' ≥ n² − n − sl(q) then
                for τ ∈ T_{q_f} do
                    D_{2,τ} = D_{2,τ} ∪ {(c' + n − 1, sl(q))}
```

5.4.4 Cas des TCA triviaux

La méthode de calcul des dates auxquelles les transitions temporelles des TCA isochrones sont atteignables présentée en section 5.4.3 peut être largement simplifiée dans le cas des TCA triviaux. En effet, en vertu de leur définition, les automates sont déterministes et permettent des simplifications par rapport aux UNFA.

Ainsi, les dates auxquelles un état est atteignable s'expriment sous la forme d'une unique suite arithmétique. Si l'on considère la représentation de l'automate sous forme de graphe, les nœuds n'appartenant pas à l'unique cycle peuvent s'écrire $(i, 0)$ où i peut être déterminé de manière triviale après parcours du graphe. À l'inverse, les nœuds appartenant au cycle s'expriment comme (c, d) où c est la première date à laquelle le nœud est rencontré et d est la taille du cycle. Par exemple, dans la figure 5.2a, les dates auxquelles τ_{A_1} est activée s'expriment comme $\{1 + 2k | k \in \mathbb{N}\}$. De même, celles de τ_{A_0} sont $\{0\}$ et τ_{A_2} sont $\{2 + 2k | k \in \mathbb{N}\}$.

5.5 Intersection des dates

Pour un groupe d'exclusion G, vérifier que l'intersection des dates qui caractérisent les transitions temporelles qui le composent est vide permet de valider que ces transitions

temporelles ne peuvent se recouvrir dans le temps, ce qui traduit la propriété de non-simultanéité. Ainsi, une intersection non vide est un contre-exemple montrant que cette propriété de sûreté n'est pas vérifiée.

Après avoir calculé les dates $\mathcal{D}_\tau$ auxquelles chaque transition temporelle τ est atteignable, il est nécessaire de déterminer quand celles-ci se recoupent, afin de pouvoir vérifier la propriété de non-simultanéité. De par la méthode de construction des ensembles $\mathcal{D}_\tau$, on sait que G est défini comme une union de singletons et de suites arithmétiques notées $\{c + dk | k \in \mathbb{N}\}$. Notons D_a et D_b deux sous-ensembles distincts de dates. Il est nécessaire de vérifier la non-intersection pour chaque paire (D_a, D_b) du groupe d'exclusion G.

Intersection de deux constantes

Dans ce cas trivial, où $D_a = \{h_a\}$ et $D_b = \{h_b\}$ sont deux singletons, ces ensembles partagent une date en commun si et seulement si $h_a = h_b$.

Intersection entre une constante et une suite arithmétique

Si $D_a = \{h_a\}$ est un singleton et $D_b = \{c_b + d_b k | k \in \mathbb{N}\}$ est un ensemble infini représentant une suite arithmétique, ils peuvent présenter au plus une date en commun (qui serait alors h_a). C'est le cas quand il existe $x \in \mathbb{N}$ tel que $h_a = c_b + d_b x$, soit quand $h_a \geq c_b$ et quand d_b divise $h_a - c_b$

Intersection entre deux suites arithmétiques

Soient $D_a = \{c_a + d_a k | k \in \mathbb{N}\}$ et $D_b = \{c_b + d_b k | k \in \mathbb{N}\}$ deux suites arithmétiques. Leur intersection n'est pas vide si et seulement si l'équation diophantienne linéaire $\alpha x + \beta y = \gamma$ admet une solution pour $x \in \mathbb{Z}$ et $y \in \mathbb{Z}$, avec $\alpha = d_a$, $\beta = -d_b$ et $\gamma = c_b - c_a$.

Les équations diophantiennes linéaires sont des structures bien connues ayant reçu beaucoup d'attention. Prouver l'existence de leurs solutions et les déterminer est un problème déjà résolu [And94]. Ainsi, l'équation diophantienne montrée plus haut admet une solution dans $\mathbb{Z}^2$ si et seulement si le plus grand diviseur commun de α et β divise γ. Si cette équation n'a pas de solution, alors l'intersection entre D_a et D_b est vide. Dans le cas contraire, il existe une solution dans $\mathbb{Z}^2$ et D_a et D_b partagent une infinité de dates en commun — ce qui invalide la propriété de sûreté de non-simultanéité. Ainsi, pour une solution (x_0, y_0) de cette équation, l'ensemble de solutions $\{(x_0 + d_b k, y_0 + d_a k) | k \in \mathbb{Z}\}$ peut toujours être construit et cet ensemble de solutions dans $\mathbb{Z}^2$ contient une infinité de paires où chaque membre est un entier naturel.

5.6 Preuve de concept

Les algorithmes de détermination des dates présentés en section 5.4 et la méthode de calcul des intersections de dates présentée en section 5.5 ont été intégrées à un démonstrateur permettant de tester la propriété de non-simultanéité sur des graphes

apparentés aux TCA isochrones. Il s'agit d'un exécutable écrit en langage D [3], open-source [4], sous licence Apache-2.0, prenant comme entrée la spécification de chaque graphe avec une liste de groupes d'exclusion et génère un rapport contenant les dates d'intersection (si elles existent) et indiquant si la propriété de sûreté exprimée est validée.

5.6.1 Exemple d'exécution triviale

Nous illustrons le principe de fonctionnement de cet exécutable servant de preuve de concept : `mcti-detect`, en reprenant l'exemple donné en figure 5.2. Le graphe de la tâche A, est décrit au format JSON en figure 5.5. La manière d'exprimer la tâche B est similaire, nous ne la montrons pas ici par souci de clarté. Nous décrivons deux cas, pour deux groupes d'exclusion différents : l'un valide et l'autre invalide.

```
{
  "graph": [
    {
      "source": "a",
      "targets": [
        {
          "node": "b",
          "transition": "A0"
        }
      ]
    },
    {
      "source": "b",
      "targets": [
        {
          "node": "c",
          "transition": "A1"
        }
      ]
    },
    {
      "source": "c",
      "targets": [
        {
          "node": "b",
          "transition": "A2"
        }
      ]
    }
  ],
  "start": "a"
}
```

Figure 5.5 – Description au format JSON de l'automate en figure 5.2a.

3. https://dlang.org/
4. https://github.com/krono-safe/mcti-detect

Groupe d'exclusion valide

Nous reprenons le même groupe d'exclusion que dans la figure 5.2. Écrit au format JSON, celui-ci peut être écrit comme suit :

```
{
  "groups": [
    ["A1", "B2", "B4"]
  ]
}
```

Notre outil d'analyse vérifie automatiquement que cette propriété de sûreté ne présente aucun contre-exemple. Comme vu précédemment, ce groupe d'exclusion est en effet valide, aussi la sortie suivante est affichée. Elle indique les dates auxquelles chaque transition temporelle est atteignable. Par exemple, la transition B_4 (B4) est atteignable pour la première fois à la date 4, puis tous les quatre tics.

```
B0: [{ 0 }]
B1: [{ 1 + 4k }]
B2: [{ 2 + 4k }]
B3: [{ 3 + 4k }]
B4: [{ 4 + 4k }]
A0: [{ 0 }]
A1: [{ 1 + 2k }]
A2: [{ 2 + 2k }]
No intersection found
```

Groupe d'exclusion invalide

Afin de vérifier que des contre-exemples sont bien détectés, spécifions un groupe d'exclusion que l'on peut vérifier comme invalide après examen de la figure 5.3 :

```
{
  "groups": [
    ["A1", "B3", "B4"]
  ]
}
```

Ceci est confirmé par l'exécution de `mcti-detect`, qui confirme qu'un recouvrement entre deux transitions est possible, et en particulier à la date 3 (au quatrième tic, sachant que la date zéro marque le début de l'exécution) :

```
B0: [{ 0 }]
B1: [{ 1 + 4k }]
B2: [{ 2 + 4k }]
B3: [{ 3 + 4k }]
B4: [{ 4 + 4k }]
A0: [{ 0 }]
```

```
A1: [{ 1 + 2k }]
A2: [{ 2 + 2k }]
Found intersection between ‘A1’ and ‘B3’ at date ‘3’
```

5.6.2 Vérification d'automates non triviaux

Prenons un cas d'application plus complexe, constituée de deux TCA isochrones E et G non triviaux, implantés sur deux cœurs différents. Admettons que les exigences de l'application imposent l'échange de données entre E et G par l'intermédiaire d'une ressource mémoire partagée. Dans ce cas, nous faisons l'hypothèse que les contraintes temporelles sont figées : aucun nœud ne peut être ajouté ou retiré. Notons que cela n'est pas le cas quand l'application est conçue incrémentalement ; c'est ici un exemple illustratif. L'objectif est d'utiliser le principe de non-simultanéité pour garantir que les accès aux ressources matérielles partagées ne peuvent être réalisés que dans des fenêtres d'exécution ne se recouvrant pas dans le temps.

Conception initiale

Une première conception de cette application est montrée en figure 5.6. Ici, les transitions temporelles E_2, E_4, E_6, E_{11}, G_3, G_6, G_8 et G_{14} accèdent à une même ressource matérielle partagée, et font donc partie d'un même groupe d'exclusion. Après analyse automatique par notre outil, nous détectons un contre-exemple : E_{11} et G_6 peuvent se recouvrir dans le temps à la date 12, comme montré en tableau 5.1.

Date	0	1	2	3	4	5	6	7	8	9	10	11	**12**
E	E_0	E_1	E_4	E_{12}	E_6	E_7	E_8	E_{10}	E_{11}	E_7	E_8	E_{10}	$\boldsymbol{E_{11}}$
G	G_0	G_1	G_{10}	G_{11}	G_9	G_5	G_6	G_1	G_{12}	G_{17}	G_{13}	G_{15}	$\boldsymbol{G_6}$

Table 5.1 – Contre-exemple montrant la trace menant au recouvrement de E_{11} et G_6 à la date 12.

Proposition d'une conception alternative

Comme la conception en figure 5.6 ne respecte pas le groupe d'exclusion garantissant un partage de ressource sûr, nous essayons une autre conception, impliquant d'autres transitions temporelles dans le partage des ressources : E_2, E_4, E_6, E_{11}, G_3, G_5, G_8, G_{14} et G_{15}, comme présenté en figure 5.7. Notre outil permet de vérifier automatiquement que ce groupe d'exclusion ne présente aucun contre-exemple.

5.7 Application concrète au sein d'Asterios

Nous avons mis en œuvre en section 5.6 une preuve de concept générique permettant de vérifier des propriétés de non-simultanéité sur des automates arbitraires. Nous allons ici détailler son intégration à Asterios sur un exemple complet.

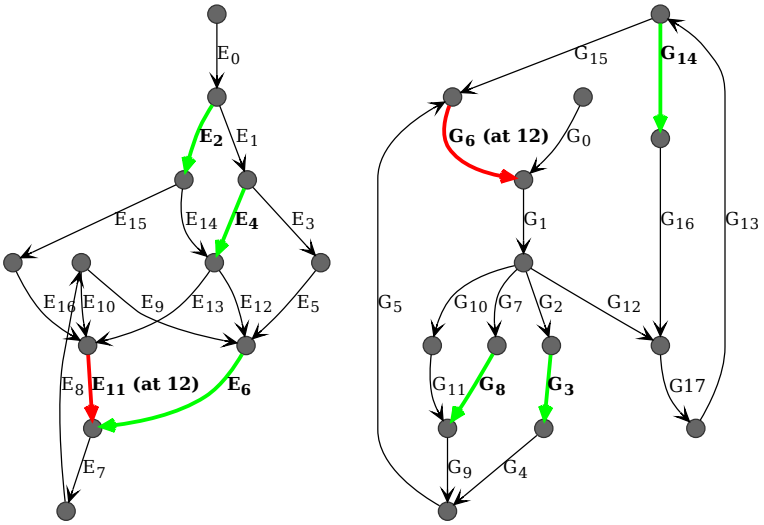


Figure 5.6 – Conception avec interférences, ne garantissant pas un partage de ressources sûr. Un contre-exemple est montré en tableau 5.1. Les arcs verts indiquent qu'au terme de l'analyse, aucun recouvrement temporel avec cet arc n'a été révélé. À l'inverse, les arcs rouges présentent un recouvrement temporel entre eux.

5.7.1 Intégration à la chaîne de compilation

Nous avons détaillé au chapitre 3 qu'Asterios s'appuie sur une chaîne de compilation spéciale à laquelle l'utilisateur fournit les descriptions temporelles des tâches *via* le langage PsyC. Nous avons étendu ce langage pour permettre l'annotation des instructions `advance`, ce qui permet à l'utilisateur de les nommer. L'identification de deux `advance` consécutifs permet ainsi de nommer les EA. Ainsi, le code PsyC en figure 5.8 permet de décrire deux agents exprimant les mêmes contraintes temporelles que les TCA vus en figure 5.2. Les `advance` sont nommés par le texte qui suit l'identifiant « `@` » et précède le mot-clef `advance` : `A0`, `A1`, `A2`, `B0`, `B1`, `B2` et `B3`.

Le nommage des EA est réalisé par une description au format JSON comme montré en figure 5.9. À noter que le premier nœud des TCA est implicite en PsyC : il n'est jamais explicitement écrit par l'utilisateur. La date à laquelle ce nœud est activé est modifiable grâce à l'instruction `starttime` (initialisée à 1 si non spécifiée). Aussi, ce nœud implicite est nommé `start` par convention, afin de permettre à l'utilisateur de nommer les EA que ce nœud contraint.

5.7.2 Description de l'application de test

Nous mettons en place une application produisant une image en niveaux de gris avec huit bits de profondeur ainsi que son CRC (Cyclic Redundancy Check) *via* une

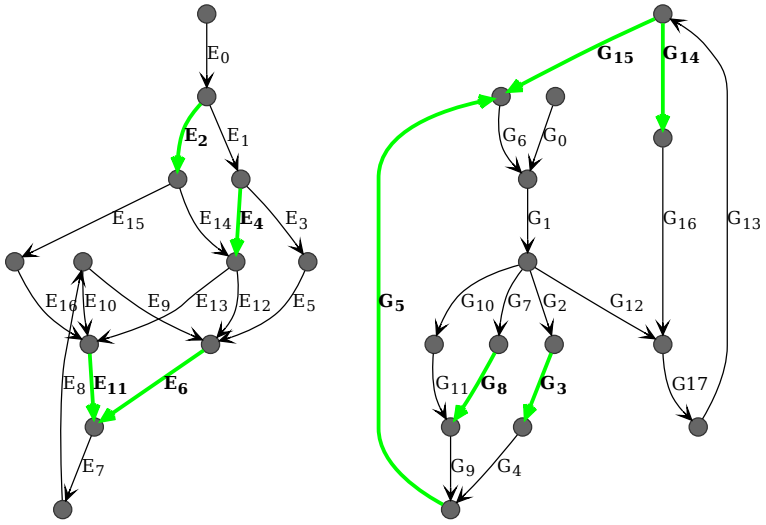


Figure 5.7 – Conception sans interférences, garantissant pas un partage de ressources sûr. La légende est identique à celle de la figure 5.6.

```
source realtime;
clock clk = 1 * realtime;
agent task_A (uses realtime)
{
  body start {
    @A0 advance 1 with clk;
    /* Accès */
    @A1 advance 1 with clk;
  }
}
```

```
source realtime;
clock clk = 1 * realtime;
agent task_B (uses realtime)
{
  body start {
    @B0 advance 1 with clk;
    @B1 advance 1 with clk;
    /* Accès */
    @B2 advance 1 with clk;
    @B3 advance 1 with clk;
    /* Accès */
  }
}
```

Figure 5.8 – Code PsyC décrivant les TCA vus en figure 5.2.

ligne série. L'image est générée de manière procédurale avant qu'un filtre de traitement d'image y soit appliqué. Ces deux fonctions peuvent être configurées à l'exécution : les paramètres de génération et de filtrage sont modifiables *via* la lecture d'une ligne série. Ces spécifications sont récapitulées en figure 5.10.

L'objectif dans cet exemple est de tirer parti des fonctionnalités mises à disposition par la plateforme matérielle présentée en section 3.5 en utilisant les trois cœurs, tout en évitant les accès simultanés aux ressources matérielles partagées. On utilise pour cela le principe de non-simultanéité développé dans ce chapitre. Une manière d'atteindre cet

```
{
  "eas": {
      "task_A": [
          { "name": "A0", "from": "start", "to": "A0" }
          { "name": "A1", "from": "A0",    "to": "A1" }
          { "name": "A2", "from": "A1",    "to": "A0" }
      ]
      "task_B": [
          { "name": "B0", "from": "start",  "to": "B0" }
          { "name": "B1", "from": "B0",     "to": "B1" }
          { "name": "B2", "from": "B1",     "to": "B2" }
          { "name": "B3", "from": "B2",     "to": "B3" }
          { "name": "B4", "from": "B3",     "to": "B0" }
      ]
  ]
  "exclusion_groups": [[ "A1", "B2", "B4" ]]
}
```

Figure 5.9 – Description de l'association entre paires d'advance et EA, ainsi que de leur constitution en groupes d'exclusion (configuration Asterios).

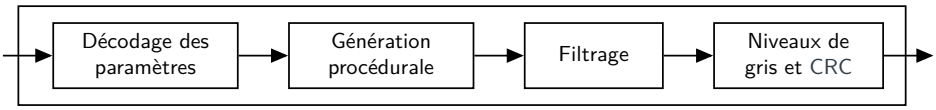


Figure 5.10 – Description du système en chaîne de blocs fonctionnels.

objectif consiste à mettre en place une architecture logicielle basée sur un pipeline où chaque cœur effectue des opérations locales et où au plus un cœur accède simultanément aux mémoires partagées. On découpe ainsi le logiciel en phases d'acquisition et de publication mémoire et en phases de calcul, comme montré en figure 5.11 (ce qui rappelle le modèle d'exécution PREM évoqué en section 2.4.3).

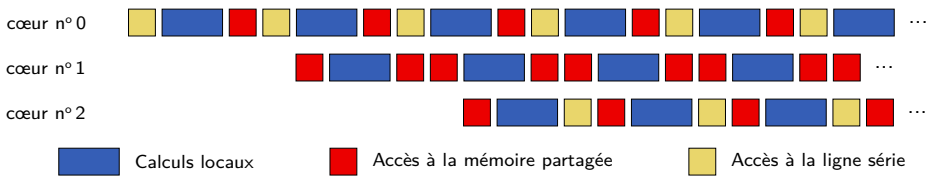


Figure 5.11 – Implémentation des spécifications en figure 5.10 en pipeline parallèle.

5.7.3 Implémentation dans Asterios

L'application décrite en section 5.7.2 est implémentée dans Asterios en écrivant plusieurs agents PsyC. Ici, nous choisissons de disposer d'un agent par cœur. Les agents des deux premiers cœurs peuvent être écrits comme montré en figure 5.12 (l'agent implanté sur le cœur n° 2 est très similaire à celui sur le cœur n° 0). Dans cette

implémentation, nous avons choisi d'allouer dix unités de temps pour les calculs locaux, et une seule pour chaque forme de communication (accès à la mémoire partagée et à la ligne série). En effet, les calculs locaux sont plus consommateurs de temps physique que les communications.

```
source realtime;
clock clk = 1 * realtime;
agent core0
( uses realtime, starttime 1 )
{
  body start {
    // Lecture série
    @c00 advance 1 with clk;
    // Génération procédurale
    @c01 advance 10 with clk;
    // Écriture mémoire partagée
    @c02 advance 1 with clk;
  }
}
```

```
source realtime;
clock clk = 1 * realtime;
agent core1
( uses realtime, starttime 14 )
{
  body start {
    // Lecture mémoire partagée
    @c10 advance 1 with clk;
    // Filtrage */
    @c11 advance 10 with clk;
    // Écriture mémoire partagée
    @c12 advance 1 with clk;
  }
}
```

Figure 5.12 – Code PsyC décrivant les agents occupant les cœurs n° 0 et n° 1.

La chaîne de compilation Asterios permet de vérifier, grâce aux groupes d'exclusion, que les EA accédant à la mémoire partagée ou à la ligne série ne peuvent jamais s'exécuter simultanément. Neuf EA sont ainsi produites :

1. acquisition de paramètres *via* une ligne série (cœur n° 0) ;
2. génération procédurale [Per02] (cœur n° 0) ;
3. publication de l'image générée en mémoire partagée (cœur n° 0) ;
4. consultation de l'image générée depuis la mémoire partagée (cœur n° 1) ;
5. filtrage de l'image (cœur n° 1) ;
6. publication de l'image filtrée en mémoire partagée (cœur n° 1) ;
7. consultation de l'image filtrée depuis la mémoire partagée (cœur n° 2) ;
8. conversion de l'image en niveaux de gris et calcul de son CRC (cœur n° 2) ;
9. envoi de ces informations *via* une ligne série (cœur n° 2).

Cette application ne peut respecter ce contrat que si son exécution reflète les intentions des groupes d'exclusion. Il est de la responsabilité de l'utilisateur de veiller à ce que le code et les données sollicitées durant les phases locales soient placées dans des régions mémoires propres aux cœurs concernés. Pour cela, nous tirons parti de la méthode de placement mémoire décrite au chapitre 4 pour profiter des mémoires locales aux cœurs de cette plateforme.

Pour permettre à la chaîne de compilation de générer une application implémentant un partitionnement temporel (comme vu en section 3.4.2), il est nécessaire de fournir des *budgets de temps* traduisant les WCET de chaque EA. Bien sûr, comme nous n'avons pas encore exécuté le programme (car une RSF est requise), nous ne pouvons fournir de mesures en conditions opérationnelles. Il est toutefois possible d'avoir recours à d'autres techniques, comme de l'analyse statique, un mode de *profiling*, ou une

heuristique d'allocation de temps d'exécution. Ici, étant donné que nous avons un seul agent par cœur, chaque EA dispose de son propre intervalle dans la RSF finale. Nous attribuons ainsi comme budget de temps 80 % de la taille de l'intervalle hébergeant l'EA — les 20 % restants sont donc du temps *idle*. Il est très probable que cette provision de temps surestime considérablement les temps d'exécution. Nous pourrons affiner ce résultat après avoir effectué des mesures de temps d'exécution sur la cible matérielle. La compilation permet la génération des RSF pour chaque cœur, qui sont montrées en figure 5.13 avec une correspondance entre EA et fenêtres temporelles préallouées.

EA	Cœur	Rôle	Budget (μs)
(1)	0	Lecture ligne série	4000
(2)	0	Génération procédurale	40000
(3)	0	Publication	4000
(4)	1	Consultation	4000
(5)	1	Filtrage	40000
(6)	1	Publication	4000
(7)	2	Consultation	4000
(8)	2	Finalisation	40000
(9)	2	Écriture ligne série	4000

Table 5.2 – Association de budgets (WCET) aux EA. Chaque EA occupe 80 % du temps de l'intervalle dans lequel elle réside (il n'y a qu'une EA par intervalle).

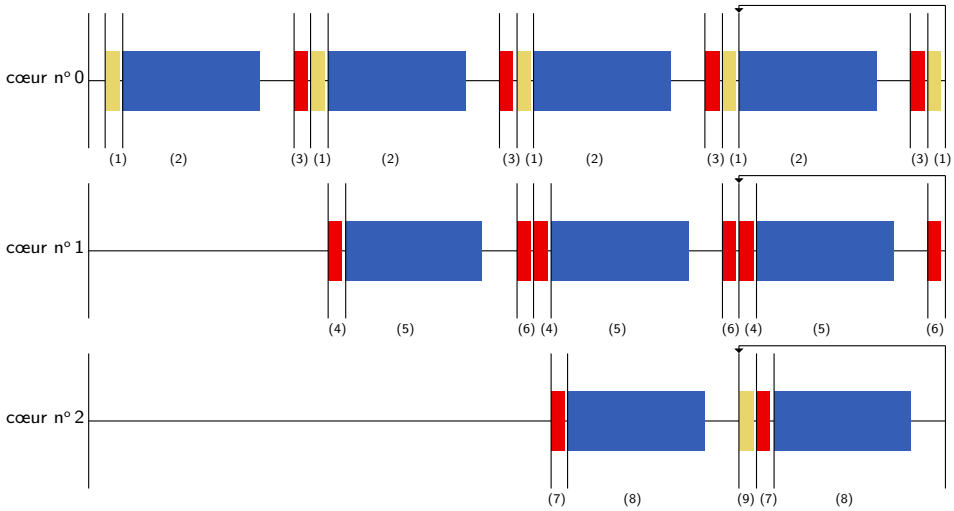


Figure 5.13 – Plans d'ordonnancement statiques (RSF) du système dans lesquels 80 % des intervalles sont réservés. La couleur utilisée pour représenter les EA est la même qu'en figure 5.11.

5.7.4 Résultats expérimentaux

Nous exécutons l'application précédemment décrite dans deux conditions expérimentales.

A. **Données partagées** : toutes les données manipulées sont placées dans la SRAM partagée : les mémoires locales D-MEM ne sont pas utilisées pour stocker les variables globales des agents (sauf leurs piles, qui résident en D-MEM). Le code de chaque agent est placé dans les mémoires locales I-MEM, évitant ainsi les interférences sur la FLASH partagée. Cette stratégie de placement mémoire ne permet pas le respect effectif du principe de non-simultanéité, mais permet au contraire d'estimer le comportement d'une application soumise à interférences causées par des accès simultanés à la SRAM.

B. **Données locales** : afin de respecter le principe de non-simultanéité, nous exploitons les mémoires locales aux cœurs comme précédemment expliqué en section 5.7.3. Les données utilisées par les EA en charge d'effectuer uniquement des calculs locaux sont placées dans les D-MEM locales.

L'objectif est de comparer les temps d'exécution observés, pour chacune des EA, entre une exécution avec et sans isolation des ressources matérielles. Ainsi, nous désactivons dans les deux cas tous les caches matériels de la plateforme afin d'isoler les effets des accès aux ressources matérielles partagées que sont la SRAM et la FLASH, et ici la SRAM en particulier.

Il est important de noter que ce test présente un facteur de confusion possible : les temps d'exécution mesurés avec des mémoires locales sont forcément biaisés par rapport à l'exécution utilisant exclusivement des mémoires partagées, car la vitesse d'accès des mémoires locales est supérieure à celles des mémoires partagées. On peut donc s'attendre à ce que les temps d'exécution des EA n'accédant plus aux mémoires partagées soient systématiquement améliorées, indépendamment du problème des accès simultanés. Toutefois, l'étude des EA de communication, donc accédant aux mêmes régions mémoires entre ces deux expériences, devrait montrer une amélioration sensible de leurs temps d'exécution, liés uniquement aux ressources partagées.

Les temps d'exécution de chaque EA sont mesurés sur cible grâce à un minuteur matériel contrôlant le code applicatif de chaque tâche. Quand une tâche est élue pour démarrer une EA, la valeur du compteur associée au minuteur est stockée dans le contexte d'exécution de l'EA. Lorsque cette EA s'achève, le temps mesuré est stocké dans un tampon de taille fixe en tant que nombre d'incréments du compteur. Après que chaque EA ait effectué 2000 itérations (ce nombre est contraint par la taille des D-MEM), soit 120 secondes par cœur, l'expérience prend fin. Le tampon contenant les mesures effectuées (par cœur) est extrait depuis la cible par le biais d'une sonde JTAG. Comme la fréquence du minuteur est connue (5 MHz) et n'est pas modifiée au cours de l'exécution du système, il est possible de calculer les temps d'exécution effectifs hors ligne, avec une précision de plus ou moins 200 nanosecondes. Nous comparons les temps d'exécution moyens et maximaux entre chaque expérience, ainsi que la variabilité de ces temps.

Les résultats expérimentaux sont montrés en figure 5.14 et certaines statistiques remarquables sont détaillées dans le tableau 5.3. On nomme respectivement **(A)** et **(B)** les deux expériences mentionnées plus haut ; **(A)** indique un placement où les données

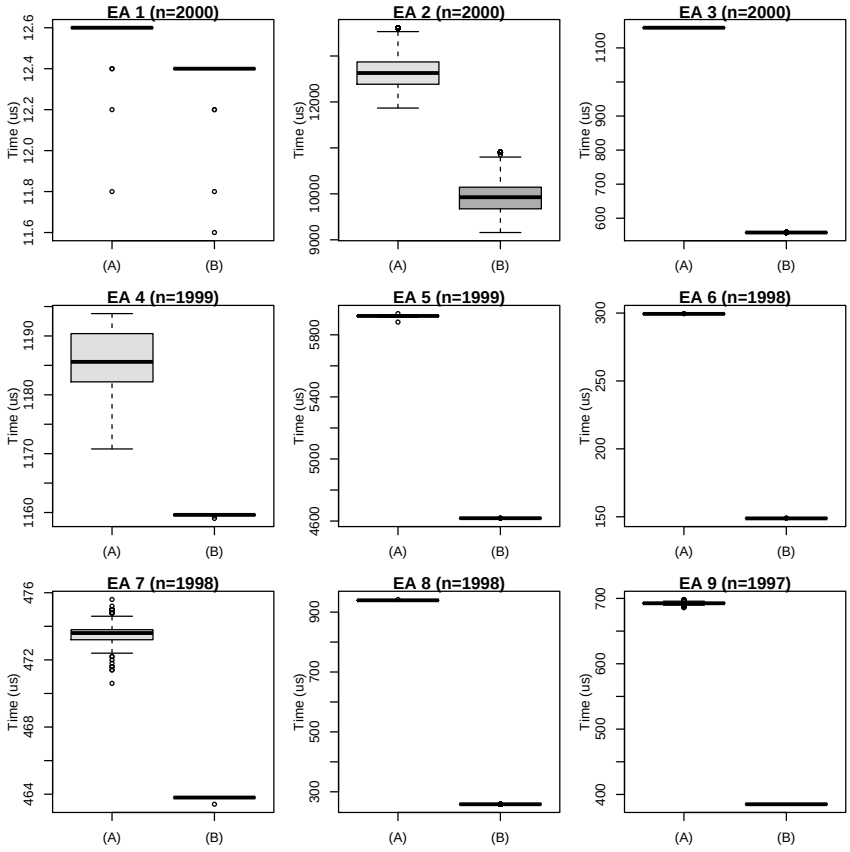


Figure 5.14 – Distributions statistiques des temps d'exécution de l'application, pour chaque EA.

sont systématiquement partagées et **(B)** un placement pour lequel les données non partagées sont affectées à des mémoires locales. Rappelons que dans les deux cas, les instructions sont placées dans les mémoires locales. Notons que le RTOS est placé globalement en FLASH. À ce titre, il est susceptible d'être la cause d'interférences, car partagé entre tous les cœurs.

Pour chacune des EA, on constate que les WCET mesurés sont systématiquement réduits dans le groupe de mesure **(B)**, ce qui est attendu, puisque d'une part l'utilisation des D-MEM permet des temps de réponse plus faibles, et d'autre part, cela permet d'éviter les interférences multicœurs causées par l'application. Comme précédemment évoqué, le changement de stratégie de placement ne permet pas d'isoler ces deux facteurs. La contribution des interférences aux temps d'exécution peut toutefois s'observer dans la variabilité des temps d'exécution sur les EA dédiées à la publication ou consultation des données globales. Elles sont particulièrement visibles pour les EA (4) et (7) où l'on observe une réduction très importante de la variabilité. Cela se confirme par la consultation du plan d'ordonnancement en figure 5.13 : ces deux EA s'exécutent

	(EA 1)		(EA 2)		(EA 3)	
	(A)	**(B)**	**(A)**	**(B)**	**(A)**	**(B)**
moyenne (μs)	12,60	12,40	12711,63	10006,84	1159,20	558,05
médiane (μs)	12,60	12,40	12629,40	9926,80	1159,20	558,00
min. (μs)	11,80	11,60	11867,00	9158,20	1159,20	558,00
max. (μs)	12,60	12,40	13614,60	10909,80	1159,20	558,20

	(EA 4)		(EA 5)		(EA 6)	
	(A)	**(B)**	**(A)**	**(B)**	**(A)**	**(B)**
moyenne (μs)	1185,26	1159,60	5921,78	4618,40	299,40	148,80
médiane (μs)	1185,60	1159,60	5921,40	4618,40	299,40	148,80
min. (μs)	1170,80	1159,00	5882,00	4617,20	299,40	148,80
max. (μs)	1193,80	1159,60	5936,60	4619,20	299,60	149,00

	(EA 7)		(EA 8)		(EA 9)	
	(A)	**(B)**	**(A)**	**(B)**	**(A)**	**(B)**
moyenne (μs)	473,61	463,80	939,54	258,44	692,74	385,00
médiane (μs)	473,60	463,80	939,40	258,40	692,60	385,00
min. (μs)	470,60	463,40	937,40	258,40	685,80	385,00
max. (μs)	475,60	463,80	942,20	259,20	698,40	385,00

Table 5.3 – Statistiques remarquables sur les temps d'exécution observés en figure 5.14.

simultanément avec l'EA (2), qui réalise un nombre important d'accès mémoires. Ainsi, avec la stratégie de placement **(B)**, cette EA n'interfère plus avec la mémoire partagée, ce qui réduit la dispersion des temps d'exécution.

Ces observations expérimentales, en particulier sur les EA (4) et (7), permettent de valider sur une preuve de concept modeste notre raisonnement sur l'exploitation des mémoires locales et sur l'intérêt du principe de non-simultanéité. Notre application est construite dessus, pour valider une conception favorisant la non-interférence des agents entre eux (*via* des accès à la SRAM partagée). Nous avons observé qu'un placement ne permettant pas de satisfaire ce principe à l'exécution est suboptimal en comparaison d'un placement conforme. En nous basant sur les mesures des temps d'exécution montrées en tableau 5.3, nous pouvons proposer de nouvelles RSF pour les cas **(A)** et **(B)**. Elles nécessitent une provision de temps moindre, comme montré en figures 5.15 et 5.16. On constate que notre système est ici — comme attendu — largement surdimensionné. L'application passe davantage de temps en *idle*, donc à ne pas effectuer de calculs utiles — certaines *frames* sont même trop petites pour être distinguées. Notons toutefois que la dernière RSF, tirant pleinement parti du principe de non-simultaénité (figure 5.16) permet de réduire encore davantage le temps alloué à chaque EA.

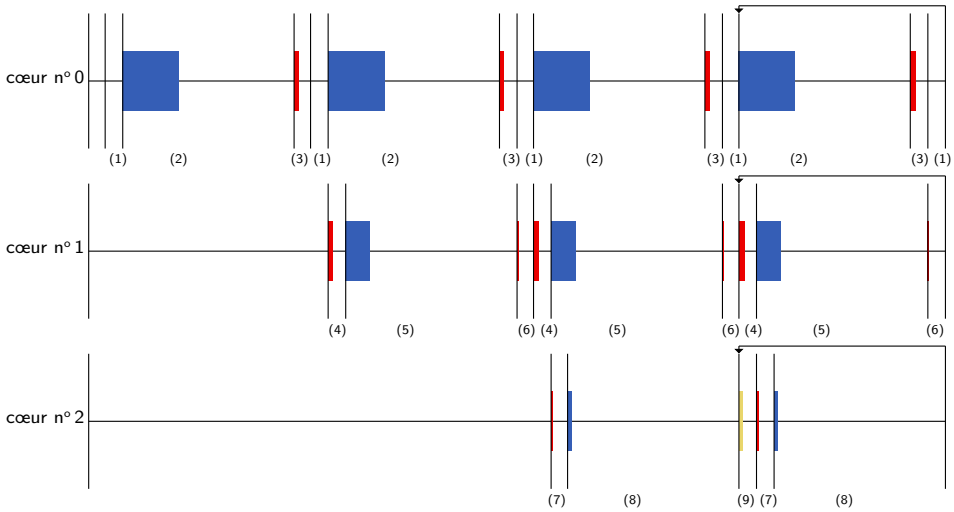


Figure 5.15 – Plans d'ordonnancement statiques du système en utilisant comme budgets de temps les WCET mesurés **(A)** (tableau 5.3), avec une surestimation (par sûreté) de 20 %.

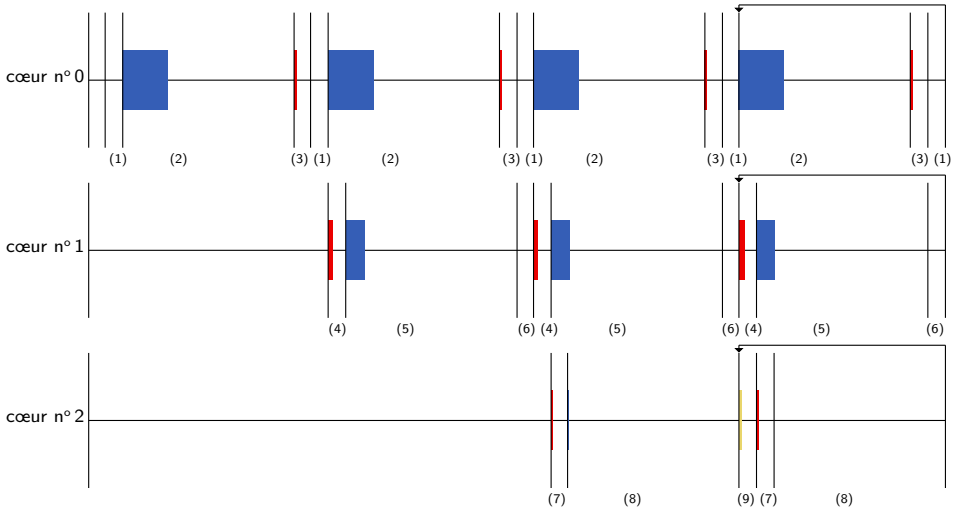


Figure 5.16 – Plans d'ordonnancement statiques du système en utilisant comme budgets de temps les WCET mesurés **(B)** (tableau 5.3), avec une surestimation (par sûreté) de 20 %.

5.8 Conclusion

Dans ce chapitre, nous avons insisté sur les impacts importants des accès simultanés aux ressources matérielles partagées sur les propriétés remarquables des temps d'exé-

cution, notamment le WCET. Nous avons proposé une approche visant à mitiger les interférences temporelles induites par ces accès simultanés en fournissant une méthode de construction d'applications temps-réelles, basée sur le formalisme TCA. Elle permet de garantir hors ligne le non-recouvrement de fenêtres temporelles à l'exécution. Ce principe de non-simultanéité, qui peut être vérifié sur des applications de tailles arbitraires, permet de construire des applications robustes à ces interférences. Nous préconisons que l'application soit conçue suivant ce principe. Si sa conception peut être validée en amont du développement, il est important de veiller à ce que son implémentation n'y déroge pas, en vérifiant au fur et à mesure de son évolution que la non-simultanéité est bien effective.

Une limitation pratique à l'approche présentée ici réside dans sa mise en œuvre sur des systèmes industriels. Sans assistance supplémentaire, l'utilisateur doit identifier manuellement la liste des accès mémoires réalisés par chaque EA, ce qui est un procédé laborieux et coûteux. En effet, le principe de non-simultanéité ne peut être effectif que si les groupes d'exclusion fournis sont complets et corrects. Nous développons dans la suite de ces travaux des techniques permettant de simplifier l'écriture des groupes d'exclusion pour favoriser l'utilisation du principe de non-simultanéité à la base de la conception des applications temps-réelles de sûreté dans un contexte multicœur.

Chapitre 6

Méthode d'analyse par co-exécution

Sommaire

6.1 Motivations

Les accès simultanés aux ressources matérielles partagées pouvant survenir durant l'exécution d'applications sur des plateformes multicœurs engendrent des interférences temporelles. Celles-ci dégradent certaines propriétés temporelles d'intérêt, comme vu au chapitre 2.

Nous proposons ici une méthode visant à construire des applications temps-réelles dans des environnements multicœurs tout en facilitant leur analyse temporelle, dans l'objectif de contribuer à la certification de ces applications, conformément aux attentes du CAST-32A. Pour cela, nous nous appuyons sur le principe de non-simultanéité que nous avons proposé au chapitre 5, qui permet d'exprimer des contraintes garantissant que des fenêtres temporelles d'intérêt ne puissent jamais se recouvrir dans le temps. La vérification de cette propriété de sûreté permet la conception de systèmes robustes aux interférences induites par le multicœur puisque leurs occurrences ne peuvent plus survenir. Nous l'associons à une analyse expérimentale de la cible matérielle, afin de capturer son comportement temporel dans des conditions d'utilisation nominales.

Ces travaux n'ont pas encore été publiés [Guy+20b]. Ils ont été soumis à un journal dont nous sommes dans l'attente.

6.2 Détail d'une méthode d'analyse

Afin d'exploiter le principe de non-simultanéité tout en nous conformant aux contraintes industrielles de ces travaux, notre méthode considère des applications basées sur les principes régissant Asterios décrits au chapitre 3, et particulièrement :

— le modèle de calcul TCA ;

— l'utilisation d'un modèle d'exécution différenciant temps logique et temps physique ;

— un découpage en agents indépendants, communiquant *via* des canaux de communication dédiés régis par le principe de visibilité.

6.2.1 Vue d'ensemble

La description générale de notre méthode est montrée en figure 6.1. Elle débute par une analyse rigoureuse du matériel utilisé, dont la bonne compréhension conditionne l'écriture de *co-runners*, détaillés en section 6.2.3. Ils permettent de caractériser la plateforme matérielle ainsi que le logiciel embarqué grâce à l'observation de distributions statistiques de temps d'exécution mesurées sur cible. Les informations qualitatives et quantitatives obtenues par analyse de ces temps permettent la mise en place de stratégies de mitigation des interférences. L'objectif étant de faciliter l'application de techniques d'analyse temporelles « classiques », dont l'application est maîtrisée sur des systèmes monocœurs, mais que les interférences temporelles induites par le multicœur rendent difficile à exploiter dans la pratique.

Notons qu'une approche similaire est développée de manière indépendante par l'entreprise Rapita Systems [1]. Fortement axée sur le principe du cycle en « V », elle sert à caractériser la plateforme logicielle et matérielle grâce à des procédés analogues [VE20 ; VT20]. Toutefois, en nous attachant aux spécificités du modèle Asterios, notre méthode nous permet en plus de proposer des mitigations aux interférences temporelles.

1. https://www.rapitasystems.com/

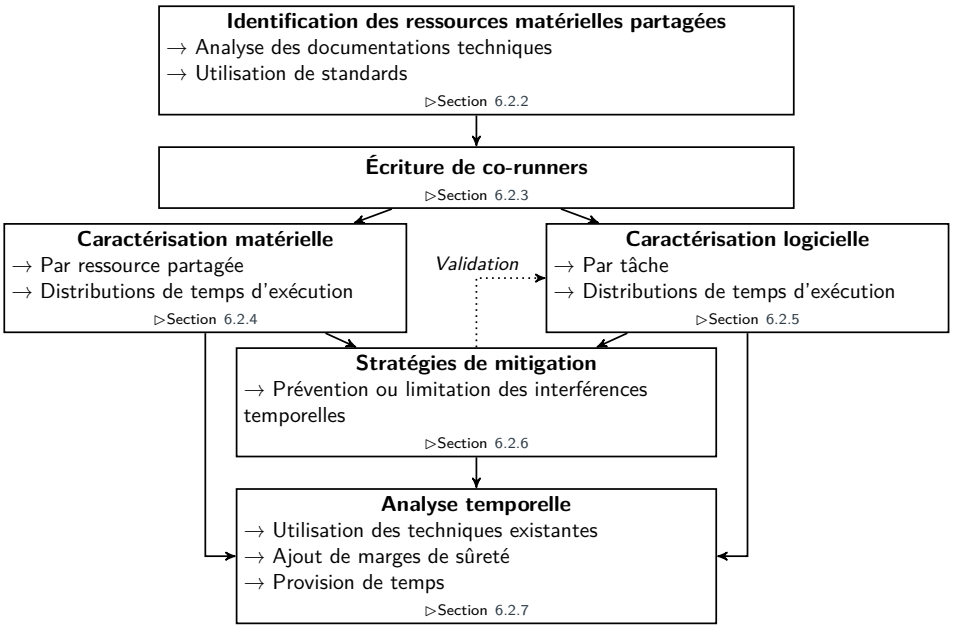


Figure 6.1 – Méthode d'analyse permettant l'élaboration de stratégies de mitigation des interférences temporelles visant à faciliter l'analyse temporelle des applications temps-réelles multicœurs.

6.2.2 Identification des canaux d'interférences

Les canaux d'interférences sont des structures matérielles susceptibles d'affecter les temps d'exécution d'applications indépendantes quand celles-ci sont exécutées simultanément. Les accès simultanés aux ressources matérielles partagées (p. ex. les mémoires partagées, les périphériques) en sont les exemples les plus évidents. Il convient toutefois de noter que l'implémentation matérielle d'une plateforme est susceptible de contenir des canaux d'interférences *cachés*, c'est-à-dire qu'ils sont manipulés indirectement et que leur existence n'est pas nécessairement exposée à l'utilisateur dans les documentations techniques [Bin+14a]. Il est ainsi difficile de lister exhaustivement tous les canaux d'interférences sans faire d'expérience éprouvant le matériel. Nous faisons ainsi la distinction entre *ressources partagées* qui sont identifiables en amont et *canaux d'interférences cachés* que seule une exécution sur cible peut mettre en évidence avec certitude.

La première étape de notre méthode consiste ainsi à explorer la documentation technique mise à disposition par le fabricant du matériel. Une analyse experte permet d'identifier les ressources matérielles pouvant être partagées entre au moins deux cœurs, et en réaliser la liste exhaustive (autant que possible). Cette analyse est demandée par le CAST-32A, ce qui la rend incontournable. Ces ressources pourront être, en fonction des cibles et de manière non exhaustive : les mémoires partagées, les caches partagées (p. ex. L2, L3), les bus d'interconnexion. Même si cette analyse n'est pas exhaustive vis-à-vis des canaux d'interférences, elle permet une première collecte d'informations

qui orienteront les prochaines étapes de notre méthode.

Cette analyse peut être assistée ou complétée par des standards émergents, comme le SHIM (Software-Hardware Interface for Multi-many-core), un standard IEEE permettant aux constructeurs de matériel de fournir à ses utilisateurs une description formalisée de la plateforme matérielle [IEE19]. Toutefois, comme ces standards sont récents, leur utilisation ne semble pas être encore répandue.

6.2.3 Conception de co-runners

Définition 10 (Co-runner)**.** *Séquence infiniment répétée d'instructions consistant à stresser une ressource matérielle partagée, de sorte à maximiser l'utilisation de sa bande passante. Elle n'est pas soumise à des contraintes temporelles particulières.*

Le concept de « co-runner » a notamment été développé dans les travaux de Bin et al. afin de quantifier la disponibilité des ressources matérielles partagées et de mettre à jour certaines caractéristiques non documentées du matériel [Bin+14a]. Un cœur processeur est ainsi programmé pour héberger un seul co-runner qui dispose ainsi de la totalité de ses ressources calculatoires. Sa seule activité consiste à stresser de manière intelligente les ressources matérielles considérées. Toutefois, plus la plateforme matérielle est riche, plus ce nombre croît, augmentant de manière conséquente l'espace d'exploration. Il est alors possible de construire des co-runners hybrides stressant plusieurs ressources. L'élaboration de ces fragments de code doit ainsi être soigneusement considérée par les architectes systèmes qui devront chercher à maximiser la consommation de bande passante tout en gérant un espace d'exploration pouvant être conséquent. Celui-ci aura été déterminé durant la première étape de notre méthode, qui consiste à identifier les ressources matérielles partagées, comme vu en section 6.2.2.

Notons que notre utilisation des co-runners consiste uniquement à stresser les ressources matérielles partagées ; nous ne comptons pas utiliser cette technique pour capturer des WCET. En effet, si d'autres travaux s'y sont essayés [Bin+14b ; Fer+15], il nous semble ardu de fournir des garanties fortes que les co-runners permettent de capturer avec certitude un temps d'exécution maximal représentatif d'une exécution réelle. Nous préférons nous servir de cet outil dans un but purement instrumental, nous aidant à élaborer des stratégies de mitigation des interférences temporelles, appliquées à l'application finale.

Des travaux menés par différents industriels ont abouti à des concepts très similaires aux co-runners. Rapita Systems développe ainsi les RapiDaemons [VE20], Maspa Technologies [2] les micro-benchmarks [Mas21]. On retrouve d'autres travaux académiques dans la même veine, afin de caractériser les plateformes matérielles [Pal+20 ; Mas+21].

6.2.4 Caractérisation de la plateforme matérielle

Après avoir identifié l'ensemble des ressources matérielles partagées (section 6.2.2) et construit les co-runners permettant un stress efficace (section 6.2.3), la plateforme matérielle doit être caractérisée. On cherche ainsi à mettre en évidence l'impact de l'utilisation des ressources matérielles sur les temps d'exécution d'un logiciel de contrôle,

2. https://maspatechnologies.com

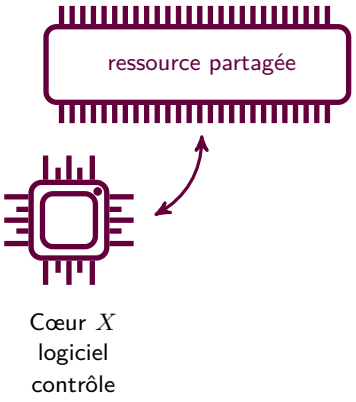
et en particulier à savoir si cet impact est *significatif*. C'est-à-dire si le partage de la ressource matérielle entre plusieurs cœurs a des effets jugés négatifs sur les temps d'exécution. On effectuera des prises de mesure de temps d'exécution d'un logiciel accédant à une ressource matérielle dans plusieurs environnements et pour chaque ressource partagée identifiée.

Ce processus de caractérisation du matériel est montré en figure 6.2 et se déroule comme suit.

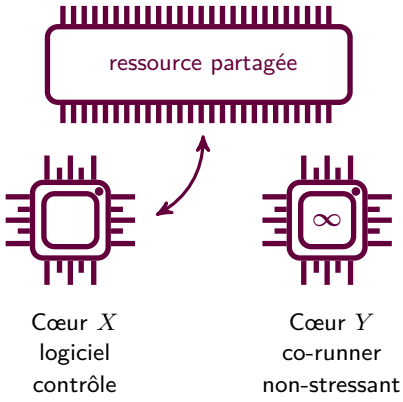
1. Le logiciel contrôle est implanté sur un cœur processeur X et aucun co-runner n'est encore actif. Les temps d'exécution mesurés durant l'exécution de cette première configuration reflètent les temps d'accès à la ressource matérielle considérée dans le cas monocœur le plus simple, comme montré en figure 6.2a.
2. Le logiciel contrôle est implanté sur un cœur processeur X, comme dans le cas précédent. À la différence près qu'un deuxième cœur $Y \neq X$ est activé et exécute un co-runner. Ce cœur Y ne doit effectuer aucun accès à la ressource partagée : cette expérience cherche à évaluer les interférences liées uniquement à l'activation d'un nouveau cœur, comme montré en figure 6.2b. Une telle implémentation n'est pas toujours réalisable en fonction de la plateforme matérielle considérée. Dans le cas où des mémoires locales aux cœurs ne seraient pas disponibles, les caches d'instructions et de données de premier niveau (L1) du cœur Y peuvent être mobilisés pour garantir cet effet.
3. Le logiciel contrôle est implanté sur un cœur processeur X, et est en concurrence avec le co-runner implanté sur un deuxième cœur Y. Dans ce cas, en figure 6.2c, les deux cœurs accèdent simultanément à la même ressource matérielle partagée, dont le co-runner cherche à maximiser le stress. Cette expérience permet de mettre en évidence les impacts sur les temps d'exécution des accès simultanés à une ressource matérielle partagée identifiée.

Cette méthode de caractérisation du matériel peut se décliner en fonction du nombre de cœurs disponibles sur le matériel : des combinaisons de plusieurs co-runners stressants peuvent être conçues pour évaluer les différences entre les cœurs. Par exemple, des architectures comprenant des cœurs hétérogènes peuvent montrer des sensibilités variables aux accès simultanés aux ressources matérielles partagées. Il est également recommandé d'exécuter un logiciel de contrôle n'accédant pas à la ressource matérielle partagée, afin de rechercher à montrer l'effet inverse : le stress d'une ressource matérielle par un autre cœur a-t-il un impact sur les temps d'exécution du logiciel contrôle ? Cette procédure permet de collecter des informations *quantitatives* et *qualitatives* permettant d'évaluer la signature d'interférences temporelles liées aux accès simultanés aux ressources matérielles partagées ainsi qu'à certains canaux d'interférences cachés. Les observations suivantes sont attendues.

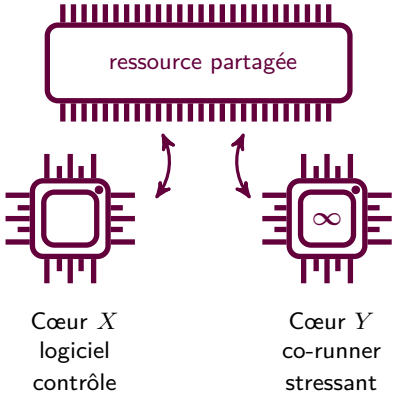
— Les différences observées entre les distributions de temps d'exécution entre les expériences (1) et (2) de la liste plus haut permettent de révéler des interférences temporelles causées par des canaux d'interférences cachés. Les co-runners non stressants devraient avoir un impact mineur sur les temps d'exécution. Le contraire pourrait impliquer un couplage fort entre cœurs, suggérant que le matériel choisi n'est pas adéquat pour une analyse temporelle robuste en multicœur.



(a) Exécution monocœur du logiciel contrôle avec accès exclusif à la ressource partagée.



(b) Exécution du logiciel contrôle avec cœur auxiliaire actif n'accédant pas à la ressource partagée.



(c) Exécution du logiciel contrôle en concurrence avec un co-runner stressant la ressource partagée.

Figure 6.2 – Caractérisation d'une ressource matérielle partagée en trois étapes, où $\leftrightarrow$ désigne les accès d'un cœur à la ressource considérée. Ici, seulement deux cœurs sont représentés, mais ce procédé se décline lorsque davantage de cœurs sont disponibles.

— Le stress de la ressource matérielle partagée par le co-runner devrait avoir un impact significatif sur les temps d'exécution du logiciel contrôle accédant à cette ressource. Le contraire peut indiquer que le matériel choisi est particulièrement robuste à ce type d'interférences temporelles, en supposant que le co-runner ait été correctement conçu et qu'il remplisse bien sa fonction.

— Le stress de la ressource matérielle partagée par le co-runner devrait avoir un impact mineur sur les temps d'exécution du logiciel contrôle n'accédant pas à cette ressource. Le contraire suggère une erreur de programmation du co-runner

ou du logiciel contrôle, ou met en évidence un canal d'interférence caché ayant un impact significatif. Ce dernier cas montrerait que le matériel choisi n'est pas robuste aux interférences temporelles et les mitigations que nous proposons ne peuvent s'appliquer : l'application du principe de non-simultanéité pour ségréguer les ressources matérielles partagées ne permet pas de limiter les interférences temporelles.

6.2.5 Caractérisation de la plateforme logicielle

L'étape de caractérisation de la plateforme logicielle peut s'effectuer parallèlement à la caractérisation du matériel vue en section 6.2.4. Elle requiert toutefois d'avoir préalablement identifié les ressources matérielles partagées et d'avoir conçu les co-runners permettant de les éprouver.

Notre positionnement privilégié vis-à-vis d'Asterios nous permet de tirer parti de la découpe en agents pour faciliter cette opération. En effet, les travaux originels des co-runners ne permettent pas d'être intégrés à côté d'une application existante : celle-ci doit être découpée manuellement pour que des fonctions clés puissent être exécutées en parallèle des co-runners. De manière générale, une application n'étant pas conçue de manière modulaire requiert des ajustements, qui ne sont pas toujours possibles dans la pratique. Notre contribution permet d'exploiter le principe de visibilité, et plus généralement le paradigme LET implémenté par Asterios pour exécuter chaque agent dans un environnement contrôlé, sans avoir à exécuter les autres tâches. Cette approche permet de dédier un cœur à l'agent testé tandis que les autres cœurs peuvent être occupés par des co-runners. Ceci est possible, car le modèle d'exécution d'Asterios n'autorise pas de migration des tâches entre différents cœurs à l'exécution.

Les applications embarquées sont souvent en charge d'interagir avec leur environnement (p. ex. par le biais de périphériques), et les tâches qui les constituent échangent souvent des données entre elles. Selon le paradigme Asterios, ces communications s'effectuent par le biais de canaux de communication implémentant le principe de visibilité. Les données transitant dans ces canaux de communication peuvent influer sur le flot de contrôle des tâches ; il est donc nécessaire de fournir des entrées aux tâches s'exécutant seules pour espérer explorer chaque branchement logique à l'exécution. Nous avons ainsi conçu des tâches particulières, appelées *stubs*[3], qui sont chargées de fournir aux agents des données d'entrées et de consommer les données qu'ils produisent. Ce flot de données artificielles est décrit par des *scénarios*, que l'utilisateur est chargé d'élaborer. Cette combinaison de *stubs* associée à un agent donné permet l'exécution d'exactement un agent de l'application sur un cœur donné, en autonomie complète. Pour faciliter les étapes de vérification et validation du logiciel, les *stubs* et leurs scénarios devraient être écrits de manière à être reproductibles et déterministes. Les exécutions successives des mêmes scénarios de test doivent conduire à l'observation du même comportement de l'agent testé.

Notons que les *stubs* s'exécutent « hors du temps », c'est-à-dire que le temps physique ne doit virtuellement être consommé que par l'agent. Rappelons que le temps physique et le temps logique sont décorrélés, comme illustré en figure 6.3. Aussi, les applications Asterios suivent un plan d'ordonnancement basé sur une provision de

3. Nous utilisons par la suite cet anglicisme, issu des méthodes de test unitaire du logiciel.

Figure 6.3 – Schéma d'exécution en temps logique d'un agent et d'un stub. Les tics $\lambda_{i|i\in\mathbb{N}}$ d'une horloge logique contraignent leur cadencement.

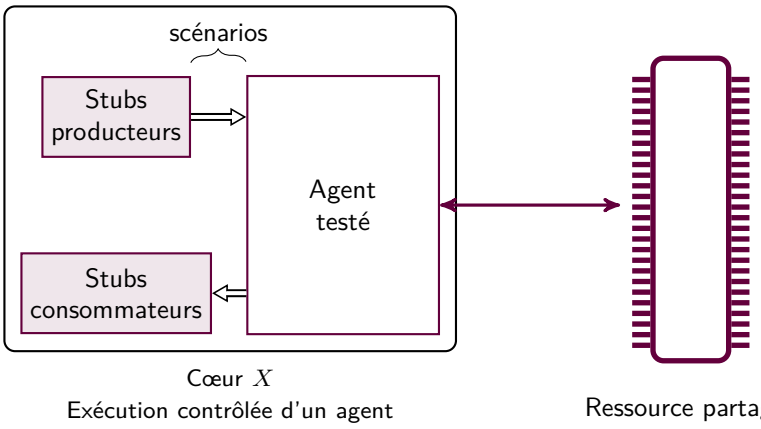
Figure 6.4 – Exemple d'utilisation du temps physique par le stub et l'agent, tout en respectant les contraintes logiques de la figure 6.3.

temps hors ligne, garantissant que leur exécution est toujours contrainte par des bornes temporelles connues. Toutefois, la caractérisation logicielle force l'exécution de l'agent en parallèle de co-runners qui ont pour but de stresser les ressources matérielles partagées, ce qui présente un impact fort sur les temps d'exécution de l'agent (si celui-ci accède à ces ressources). Il est alors possible que cette exécution en environnement hostile conduise l'agent à consommer plus de temps que nécessaire, empêchant ainsi le respect de ses contraintes temporelles. C'est pourquoi, lors de cette étape, le temps physique consommé par l'agent n'est pas vérifié. Afin de rester cohérent vis-à-vis des *stubs* qui produisent et consomment des données, l'agent doit rester conforme à ses contraintes de temps logique, ce qui implique de distinguer les deux. Plus concrètement, comme un seul cœur est mobilisé par l'exécution de l'agent, son exécution reste conforme aux spécifications du PsyC tant que ses contraintes logiques sont respectées, ce qui est toujours vrai en monocœur. Aussi, l'agent peut consommer une quantité arbitraire de temps physique tout en restant conforme de sa spécification, comme le montre l'exemple donné en figure 6.4.

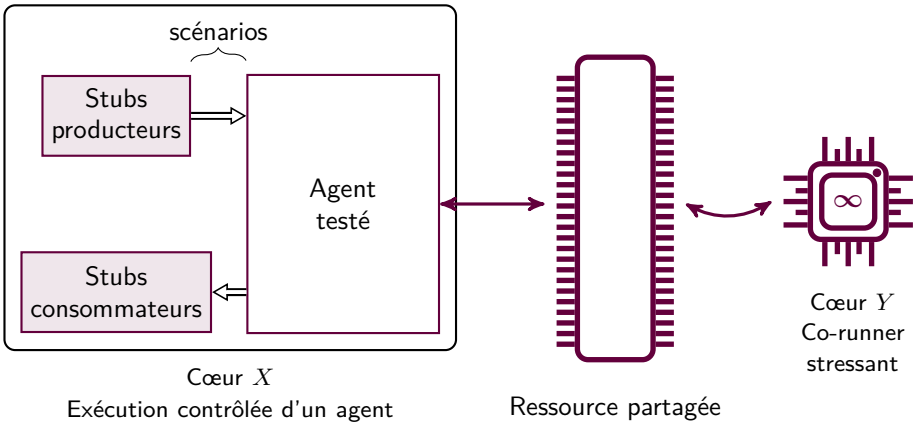
Notre méthode de caractérisation logicielle tire ainsi pleinement partie du concept de *stubs* et de scénarios, et se déroule en deux phases comme montré en figure 6.5.

1. Une première exécution est réalisée en environnement monocœur : l'agent déroule son exécution dans un environnement contrôlé qui lui fournit les entrées nécessaires à l'exploration de son code fonctionnel et éventuels branchements temporels. La complétion de chaque EA de l'agent résulte en la collecte d'un temps d'exécution associé à cette EA. Cette étape peut être répétée pour s'assurer de la faible variabilité des distributions de temps d'exécution obtenues.
2. La deuxième exécution implique l'utilisation d'une combinaison de co-runners stressant la ressource matérielle d'intérêt. De manière analogue au cas précédent l'agent est exécuté dans un environnement contrôlé, à la différence que la ressource matérielle considérée est stressée par le co-runner.

Comparer les distributions statistiques des temps d'exécution mesurés avec et sans



(a) Exécution d'un agent dans un environnement contrôlé avec accès réservé à une ressource matérielle partagée (cas monocœur).



(b) Exécution d'un agent dans un environnement contrôlé en concurrence avec un co-runner stressant ressource matérielle partagée (cas multicœur).

Figure 6.5 – Caractérisation logicielle du code d'un agent particulier. Il doit s'exécuter dans un environnement contrôlé, dans deux conditions expérimentales : avec et sans co-runner stressant la ressource matérielle partagée d'intérêt.

co-runner, pour chacune des EA de l'agent, fournit au développeur ou à l'intégrateur des informations quantitatives et qualitatives permettant d'identifier les EA accédant de manière significative à la ressource matérielle sélectionnée. Ainsi, notre méthode d'utilisation des co-runners permet d'identifier quelles sont les EA « critiques », c'est-à-dire celles pouvant être sensibles aux interférences temporelles si elles s'exécutent simultanément avec une autre EA accédant à la même ressource matérielle. Notons toutefois que nous ne préconisons aucun critère de décision permettant d'identifier formellement ces EA. Ce critère serait en effet fortement dépendant de l'application, et

nous préférons laisser à ses concepteurs le soin de ce choix.

Cette méthode nécessite de couvrir tous les branchements fonctionnels et temporels de l'agent, donc des scénarios pleinement couvrants. Cette contrainte est attendue dans le développement d'applications aéronautiques critiques, aussi les utilisateurs de cette méthode pourront réutiliser leur base de tests existante, qu'ils doivent développer pour certifier leur système.

Rappelons que si cet environnement de contrôle est suffisant pour permettre de tester un agent, il ne peut garantir des mesures de temps d'exécution pouvant être utilisées pour une analyse temporelle sérieuse. Les temps collectés sont fournis à titre indicatif, avec pour principal objectif d'effectuer des comparaisons entre distributions statistiques afin de mettre en œuvre des mitigations aux interférences temporelles causées par l'utilisation simultanée de ressources matérielles partagées.

6.2.6 Mise en place de stratégies de mitigation

Nous proposons d'utiliser le principe de non-simultanéité décrit au chapitre 5 en tant que contrainte de conception des applications, afin de mettre en œuvre des stratégies de mitigation efficaces. Il permet en effet de vérifier statiquement, à la compilation de l'application, que les EA identifiées comme accédant à une ressource matérielle partagée ne peuvent s'exécuter simultanément. Cela permet l'implémentation d'un modèle d'exécution pouvant garantir l'absence d'accès simultanés aux ressources matérielles partagées considérées. L'efficacité de ces mesures peut être confirmée en effectuant une nouvelle caractérisation logicielle des agents, une fois reconfigurés avec une stratégie de placement alternative. La méthode de partitionnement spatial que nous avons décrite au chapitre 4 peut être utilisée pour cela.

Notons que les contraintes industrielles sur l'application peuvent être localement incompatibles avec nos stratégies de mitigations. En effet, en fonction de la complexité de l'application et de la capacité à privilégier une conception prenant en compte les accès simultanés, il reste possible que toutes les contraintes d'exclusion temporelle ne puissent être mises en œuvre. C'est aussi pour cela que nous ne proposons aucun critère de sélection quant aux EA sensibles aux interférences : c'est l'utilisateur qui en fonction de ses exigences sera le plus à même d'écrire des groupes d'exclusion respectant ses contraintes. Les informations obtenues par la caractérisation matérielle et logicielle lui permettent ainsi de privilégier certaines combinaisons de groupes d'exclusion plutôt que d'autres, dans l'optique de minimiser les interférences temporelles.

6.2.7 Impact sur l'analyse temporelle

Un système construit avec le principe de non-simultanéité comme contrainte de conception dès sa genèse offre la possibilité aux EA de ne pas engendrer d'interférences temporelles lors de leur exécution. Un corollaire à cette propriété est que ces EA ne sont pas soumises aux interférences. Quand ces mitigations peuvent être déployées, leur WCET peut être estimé grâce aux techniques classiques d'analyse temporelle, qui ont fait leur preuve sur des systèmes monocœurs. En effet, si la caractérisation logicielle montre que les interférences dues à l'utilisation du multicœur sont négligeables, alors il devient possible de justifier les résultats obtenus sur la base d'un modèle monocœur. En

fonction des interférences résiduelles observées, l'intégrateur pourra choisir d'appliquer des marges de sûreté correspondantes, pour minimiser les risques.

6.3 Application à une preuve de concept

Dans cette section, nous déroulons la méthode que nous avons détaillée en section 6.2. Nous nous arrêtons après l'application des mitigations : nous n'effectuons pas d'analyse temporelle, mais nous questionnons leur application.

6.3.1 Environnement matériel

Nous utilisons la plateforme matérielle MPC5777M, qui a été présentée en section 3.5. Pour cet exercice, nous retenons deux ressources matérielles partagées après analyse de la documentation technique :

1. la FLASH, qui contient les instructions et les données en lecture seule ;
2. la SRAM, qui contient les données inscriptibles utilisées par l'application.

Nous considérons que les caches usuels sont désactivés (c.-à-d. les caches L1 — rappelons qu'il n'y a aucun cache L2 ou L3), et que seuls les cœurs n° 1 et n° 2 seront utilisés (le cœur n° 0 est donc toujours désactivé). Cette approche nous permet d'expérimenter une architecture hétérogène, où les accès à la SRAM et à la FLASH sont asymétriques entre chaque processeur.

6.3.2 Application de test

Nous définissons une application composée de deux agents PsyC avec un cadencement complexe. On les nomme G et H, et leurs graphes d'états temporels (c.-à-d. leurs TCA) sont montrés en figure 6.6. L'agent G est le même qu'en section 5.6.2. À ce stade, les ressources matérielles partagées accédées par chacune des EA qui composent ces agents sont considérées comme inconnues. Ce sont les étapes de caractérisation logicielle, détaillées en sections 6.3.7 et 6.3.8, qui permettent d'obtenir cette information. Le code complet de l'application de test n'est pas reproduit ici, par souci de concision. Il a toutefois été publié en open source et est consultable en ligne [4].

Le code et les données des agents ont un placement par défaut : certaines EA exploitent les mémoires locales aux cœurs. Les caractérisations logicielles qui suivent se reposent sur l'exécution des agents dans un environnement de contrôle qui permet d'explorer l'ensemble des branchements temporels et fonctionnels des agents testés, comme décrit en section 6.2.5. On pourra alors déterminer avec certitude quels sont les accès *significatifs* aux ressources matérielles partagées. Nous nous intéressons particulièrement à l'utilisation de la mémoire FLASH.

6.3.3 Écriture du co-runner stressant la FLASH

Comme mentionné précédemment, la FLASH est une mémoire partagée contenant typiquement le code et les données non modifiables des agents. Rappelons que tous

4. `https://github.com/krono-safe/corunners-example`, commit `078d2b6`

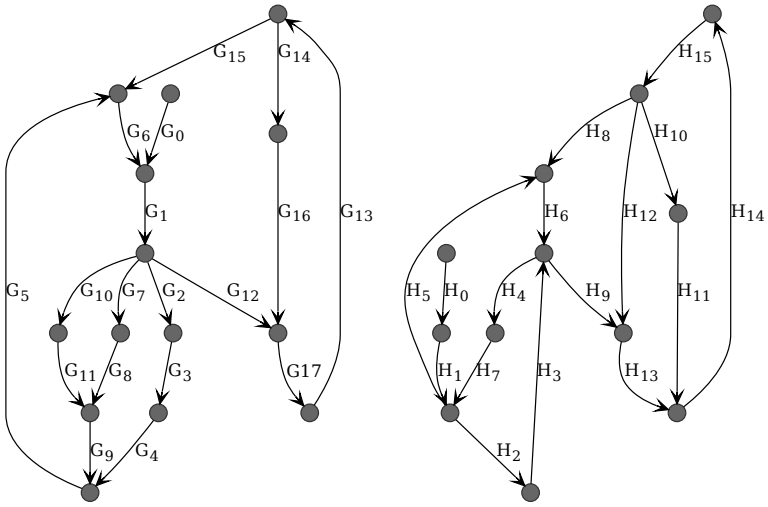


Figure 6.6 – Représentation des TCA des agents G et H testés. Les EA accédant aux ressources matérielles partagées ne sont pas encore identifiées.

les caches ont été désactivés. Nous concevons alors un co-runner pouvant stresser la FLASH, sans toutefois chercher à le rendre optimal. Afin d'éviter les accès à la pile pouvant être provoqués par l'exécution de code C, nous l'avons écrit en assembleur PowerPC. Un générateur de code nous permet de construire une séquence contrôlée de branchements. On génère ainsi un code composé uniquement d'instructions de branchement, n'effectuant aucun accès aux données, ni aucun calcul. Ceci permet de consommer davantage de bande passante. De plus, la génération d'un grand nombre d'instructions de branchements permet de limiter les effets du minicache présent au niveau du bus d'accès à la FLASH. Ainsi, le co-runner utilisé est contenu dans un seul fichier objet, comportant une unique section `.text` contenant son code.

6.3.4 Écriture du co-runner stressant la SRAM

La SRAM est une mémoire partagée entre tous les cœurs contenant les données inscriptibles à l'exécution de l'application. Les caches de données étant désactivés, le contrôleur de SRAM est accédé frontalement par les cœurs, *via* le crossbar de calcul. Nous avons réalisé un co-runner en assembleur PowerPC effectuant une suite répétée d'instructions `stw` permettant l'écriture du contenu d'un registre à des adresses précises de la SRAM. Ces adresses résident dans une zone de 512 octets, de sorte à accéder à une plage de la SRAM sans recouvrement possible avec les données que les agents G et H (et le RTOS Asterios) pourraient utiliser. Le code du co-runner est placé dans la mémoire locale au cœur sur lequel il est implanté, afin qu'il n'accède pas à la FLASH, mais uniquement à la SRAM.

6.3.5 Caractérisation matérielle de la FLASH

La caractérisation matérielle de la FLASH est réalisée suivant la méthode précédemment énoncée en section 6.2.4. Ici, nous choisissons d'écrire un agent de test composé de deux EA : l'une (F_1) n'effectuant pas d'accès à la FLASH, et l'autre (F_2) y réalisant de nombreux accès (instructions et données constantes). Les instructions exécutées entre F_1 et F_2 sont identiques. Cela est possible en dupliquant le code exécutable utilisé par chaque EA et en le plaçant à des endroits différents (c.-à-d. en I-MEM pour F_1 et en FLASH pour F_2). Cette capacité est permise grâce à notre technique de partitionnement mémoire, décrite en chapitre 4.

Afin d'éprouver des stratégies d'implantation sur des cœurs différents, nous réalisons chaque prise de mesure dans deux cas de figure. Dans le premier, l'agent testé est implanté sur le cœur n° 1 ; dans le second, il est implanté sur le cœur n° 2. Les cas suivants sont ainsi étudiés :

1. l'agent s'exécute seul sur le cœur n° 1 ;
2. l'agent s'exécute sur le cœur n° 1, et un co-runner non stressant s'exécute sur le cœur n° 2 ;
3. l'agent s'exécute sur le cœur n° 1, et un co-runner stressant la FLASH s'exécute sur le cœur n° 2 ;
4. l'agent s'exécute seul sur le cœur n° 2 ;
5. l'agent s'exécute sur le cœur n° 2, et un co-runner non stressant s'exécute sur le cœur n° 1 ;
6. l'agent s'exécute sur le cœur n° 2, et un co-runner stressant la FLASH s'exécute sur le cœur n° 1.

Les résultats montrant la distribution des temps d'exécution dans chacun des cas écrits plus haut sont présentés en figure 6.7. Les données sont représentées par des diagrammes en violon [HN98]. Ils indiquent les temps d'exécution minimum et maximum, ainsi qu'une estimation par noyau montrant la distribution de ces temps. Notons que le nombre d'échantillons de F_2 ($n = 511$) est différent de F_1 ($n = 512$). Cette différence s'explique par notre implémentation : le RTOS utilise un tampon permettant 1024 mesures d'EA. La structure du PsyC nous force à rajouter une EA surnuméraire (F_0, non renseignée ici), qui est exécutée une seule fois au démarrage de l'agent, avant de basculer sur l'enchaînement F_1 puis F_2. Nous « perdons » ainsi une mesure, qui n'est pas utilisée par la caractérisation matérielle.

On observe dans chaque cas que l'activation du co-runner non stressant présente un impact peu marqué entre les cas (1) et (2). Les données récoltées permettent de montrer qu'il s'agit d'un canal d'interférence caché : l'activation d'un nouveau cœur effectuant des accès séparés du premier dégrade les temps d'exécution mesurés. Toutefois, cet impact reste négligeable et cette caractérisation matérielle permet d'apporter un élément de preuve dans ce sens. L'activation du co-runner stressant la FLASH (3) semble renforcer ce canal d'interférence caché dans le cas de F_1, qui pourtant n'accède pas à cette ressource matérielle. Celui-ci reste toutefois très modéré, à l'inverse de F_2 qui y accède. Dans ce cas de figure, on constate que le stress de la FLASH dégrade de manière importante les temps d'exécution du code y accédant normalement. Ces observations nous permettent de confirmer que la FLASH est une ressource matérielle

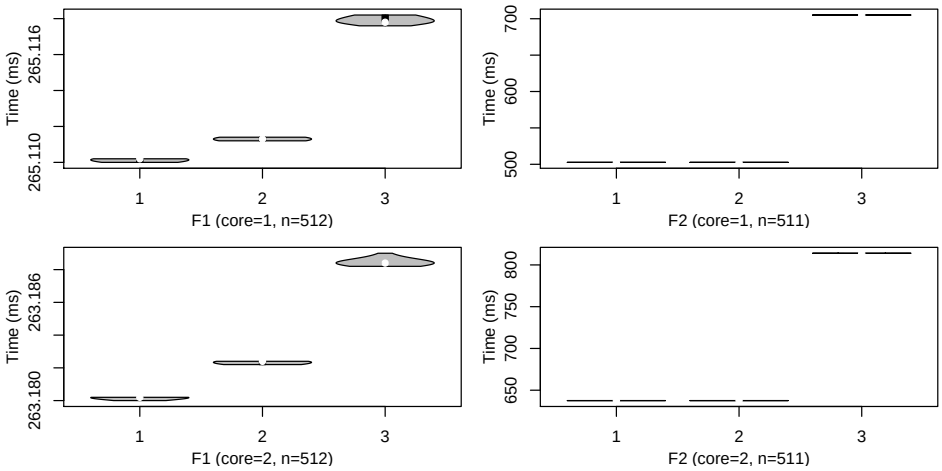


Figure 6.7 – Caractérisation matérielle sommaire de la FLASH du MPC5777M. F_1 n'accède pas à la FLASH alors que F_2 y accède. Trois mesures sont effectuées dans chaque cas : (1) sans co-runner, (2) avec co-runner non stressant et (3) avec co-runner stressant la FLASH. Le détail numérique est montré en tableau 6.1. Notons la différence entre les échelles de F_1 et F_2.

	EA	max(1) *(ms)*	max(2) *(ms)*	max(3) *(ms)*	$R(1,2)$ *(%)*	$R(1,3)$ *(%)*
Cœur 1	F_1	265,110	265,111	265,118	0,000	0,003
	F_2	502,629	502,630	705,075	0,000	40,277
Cœur 2	F_1	263,180	263,182	263,189	0,001	0,003
	F_2	637,564	637,566	813,976	0,000	27,670

Table 6.1 – Données numériques issues de la figure 6.7. Ce tableau montre, pour chaque cœur, les temps d'exécution minimaux et maximaux mesurés pour chaque EA, ainsi que des différences relatives R (comme détaillé en section 6.3.6).

susceptible de dégrader les temps d'exécution des tâches y accédant lorsque plusieurs sont en concurrence. D'autre part, si ces observations nous ont permis de déceler un canal d'interférence caché, elles permettent également d'estimer son impact comme étant négligeable.

6.3.6 Proposition de critère statistique

Avant de détailler la caractérisation logicielle pour la FLASH et la SRAM, nous proposons un critère statistique permettant d'évaluer *a priori* si les EA peuvent être classées comme interférentes après étude de leurs distributions statistiques de temps d'exécution. Ici, nous utilisons comme mesure quantitative la différence relative notée $R(x_{ref}, x)$ entre les WCET mesurés des distributions avec exécution du co-runner (représentés par x) et ceux sans présence de co-runner (représentés par x_{ref}). Cette

valeur est exprimée sous la forme d'un pourcentage, comme montré en équation (6.1).

$$R(x_{ref}, x) = 100\frac{x - x_{ref}}{x_{ref}} \tag{6.1}$$

La comparaison du résultat de ce calcul avec une valeur de seuil h déterminée en fonction des caractérisations matérielles offre un filtre quantitatif pour discriminer les EA interférentes des autres. Nous pouvons ensuite affiner ce choix en fonction des données qualitatives mises à notre disposition. En nous aidant des résultats montrés en section 6.3.5, nous sélectionnons comme valeur de h *a priori* une grandeur correspondant à un ordre de magnitude supérieur aux différences relatives calculées sans interférence. On fixe ainsi $h = 0.01\%$.

6.3.7 Caractérisation logicielle pour la FLASH

Le système étant composé de deux tâches (G et H), on considère qu'elles peuvent être implantées sur le cœur n° 1 ou n° 2, comme vu en section 6.3.2 (le cœur n° 0 est désactivé). On cherche donc à implanter chaque tâche sur ces différents cœurs, afin d'évaluer comment exploiter au mieux l'architecture hétérogène de la plateforme matérielle. On effectue ainsi les tests suivants, pour chaque agent, en utilisant le co-runner décrit en section 6.3.3.

a. L'agent testé est exécuté dans un environnement contrôlé permettant d'explorer ses branchements temporels et fonctionnels. Les temps d'exécution de chaque EA sont enregistrés, pour son implantation sur un cœur n° 1, et aucun co-runner n'est actif sur l'autre cœur n° 2.

b. Le test (a) est répété, à la différence qu'un co-runner stressant la FLASH s'exécute sur le cœur n° 2.

c. On répète le test (a), mais en changeant l'implantation : l'agent testé est exécuté sur le cœur n° 2, et aucun co-runner n'est actif sur le cœur n° 1.

d. Le test (c) est répété, à la différence qu'un co-runner stressant la FLASH s'exécute sur le cœur n° 1.

Ces expériences permettent de collecter quatre distributions de temps d'exécution, pour chaque EA composant les agents testés. Ainsi, (a) et (b) peuvent être comparés afin d'évaluer l'impact du co-runner stressant la FLASH pour des agents testés s'exécutant sur le cœur n° 1. De manière analogue, (c) et (d) peuvent être comparés pour évaluer l'impact du stress de la FLASH quand ces agents sont exécutés sur le cœur n° 2.

Les résultats observés, pour chacune de ces quatre expériences, sont présentés en figure 6.8 pour l'agent G et en figure 6.9 pour l'agent H. Chacune de ces figures est composée d'une sous-figure par EA ; ces dernières présentent chacune quatre boîtes à moustaches (une pour chaque expérience). Rappelons que ces diagrammes permettent de visualiser des distributions statistiques, notamment ses valeurs extrêmes, les premier et troisième quartiles, ainsi que la médiane. Les données clés de ces expériences sont retranscrites en tableau 6.2.

Les temps d'exécution mesurés sont représentés par des nuages de points superposés aux boîtes à moustaches, afin de visualiser les données brutes. Leur écartement par

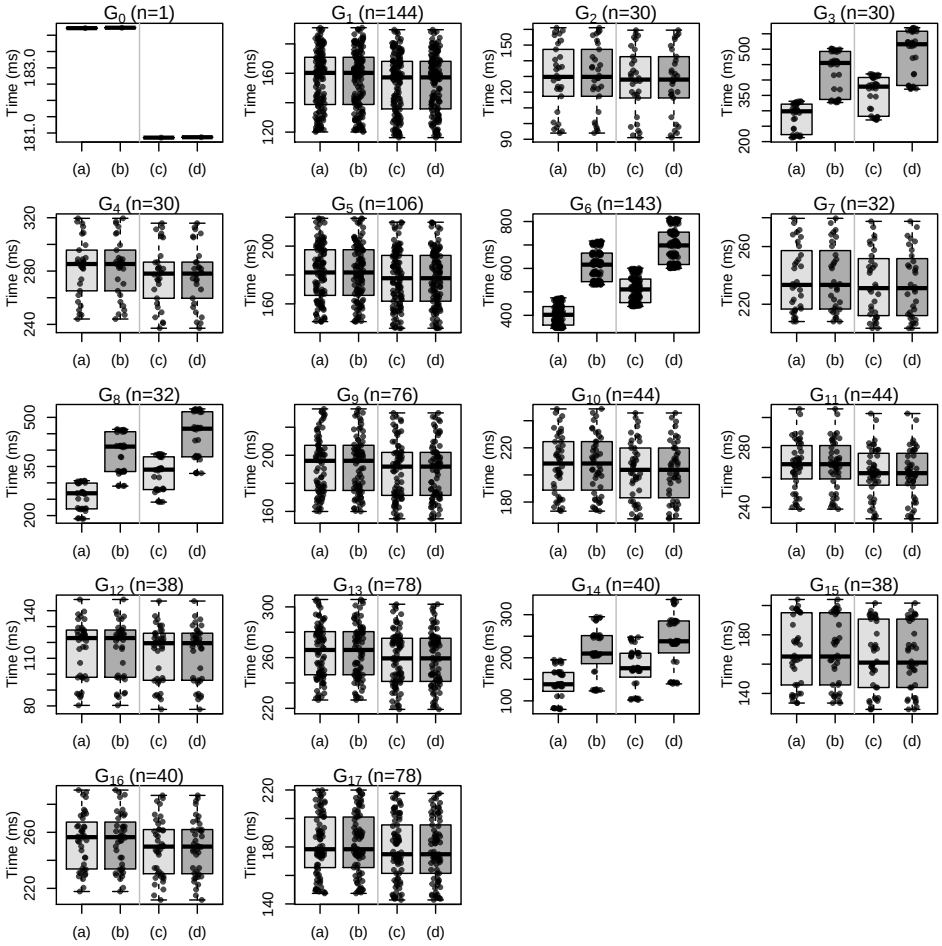


Figure 6.8 – Caractérisation logicielle de l'agent G avec stress de la FLASH. n représente le nombre de mesures de temps d'exécution par boîte à moustaches, et $G_{i|i\in\mathbb{N}}$ les EA constituant l'agent G.

rapport à l'axe des abscisses permet une meilleure clarté. Notons que les EA G_0, H_0 et H_1 n'ont qu'une seule valeur ($n = 1$) car ces résultats ne couvrent qu'une seule exécution des agents, et que ces EA ne sont accessibles qu'une seule fois au cours de la durée de vie des agents, comme montré en figure 6.6.

Les résultats quantitatifs obtenus permettent d'identifier quelles EA accèdent suffisamment à la FLASH pour exhiber des variations de temps d'exécution importantes quand cette ressource matérielle partagée est stressée. Il s'agit de G_3, G_6, G_8 et G_{14} pour l'agent G, et de H_0, H_4, H_{12} et H_{14} pour l'agent H. Ces résultats sont identiques quelle que soit la stratégie d'implantation des cœurs utilisée. Il est intéressant de noter que l'EA H_0 de l'agent H a été classifiée par notre critère statistique comme « interférente ». Toutefois, sa sensibilité est bien moindre que celles des autres EA (de trois

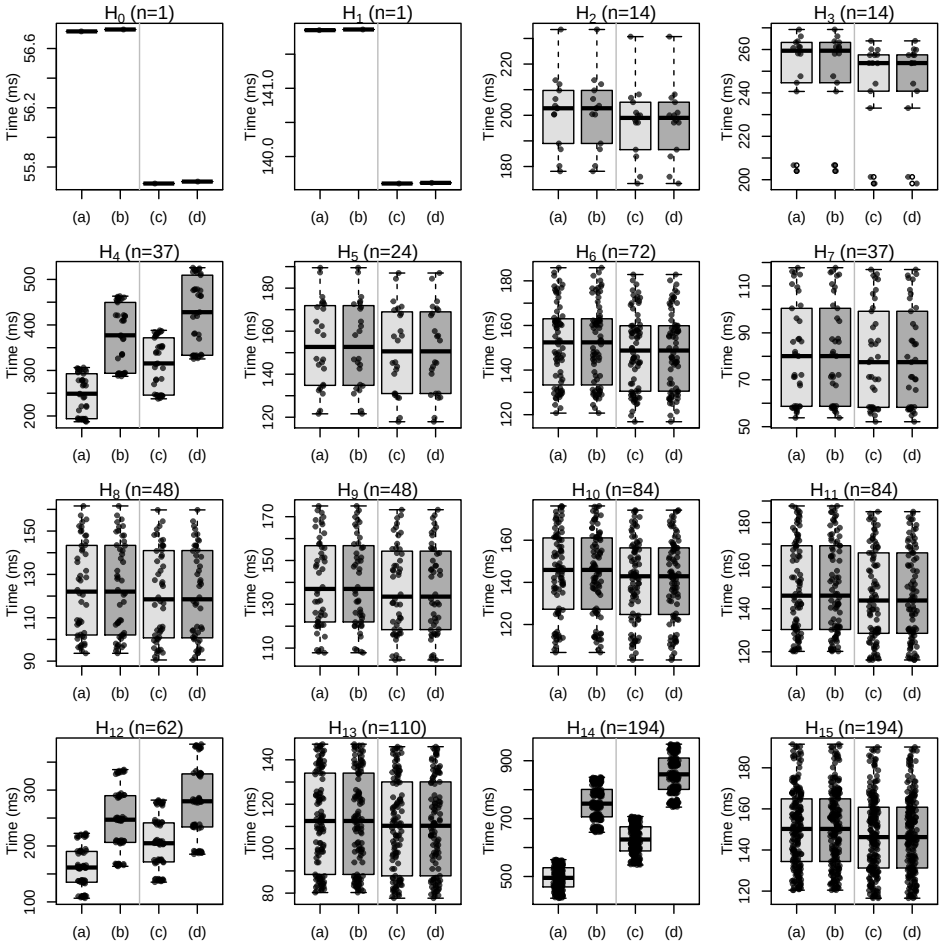


Figure 6.9 – Caractérisation logicielle de l'agent H avec stress de la FLASH. n représente le nombre de mesures de temps d'exécution par boîte à moustaches et $H_{i|i\in\mathbb{N}}$ les EA constituant l'agent H.

ordres de grandeur). On pourra ainsi la considérer comme modérément interférente, et au regard de la différence d'impact, on pourra raisonnablement l'ignorer des stratégies de mitigation.

Ces résultats permettent également de mettre en évidence l'hétérogénéité de la plateforme choisie : la stratégie d'implantation des cœurs a un impact visible sur les temps d'exécution observés. On peut noter les comportements suivants.

1. Les EA non-interférentes ont un WCET mesuré moindre quand elles sont exécutées sur le cœur n° 2 par rapport au cœur n° 1. Cette observation est à première vue contre-intuitive car le cœur n° 1 est cadencé à une fréquence supérieure au cœur n° 2, et que les accès aux mémoires pour le cœur n° 2 peuvent transiter *via* un crossbar additionnel. Ces différences sont expliquées en section 6.3.9.

	Cœur n° 1			Cœur n° 2		
EA	**max(a)** *(ms)*	**max(b)** *(ms)*	$R(a,b)$ *(%)*	**max(c)** *(ms)*	**max(d)** *(ms)*	$R(c,d)$ *(%)*
G_0	184,217	184,230	0,007	180,857	180,870	0,007
G_1	190,981	190,992	0,006	189,639	189,649	0,006
G_2	161,039	161,049	0,006	159,511	159,522	0,007
G_3	331,060	502,167	**51,684**	419,580	569,639	**35,764**
G_4	319,557	319,568	0,003	315,902	315,912	0,003
G_5	219,241	219,251	0,004	216,546	216,556	0,005
G_6	474,479	716,620	**51,033**	601,022	813,646	**35,377**
G_7	279,576	279,586	0,004	277,474	277,484	0,004
G_8	306,564	463,404	**51,161**	388,349	525,899	**35,419**
G_9	233,026	233,037	0,005	230,172	230,182	0,004
G_{10}	248,713	248,723	0,004	245,702	245,712	0,004
G_{11}	305,734	305,744	0,003	302,646	302,656	0,003
G_{12}	147,060	147,069	0,007	146,063	146,073	0,007
G_{13}	305,834	305,844	0,003	302,088	302,097	0,003
G_{14}	195,298	295,131	**51,118**	247,730	335,280	**35,341**
G_{15}	204,100	204,110	0,005	201,639	201,649	0,005
G_{16}	290,148	290,157	0,003	286,340	286,350	0,004
G_{17}	219,936	219,946	0,005	217,564	217,574	0,005
H_0	56,716	56,728	**0,023**	55,688	55,702	**0,024**
H_1	141,852	141,861	0,007	139,605	139,614	0,007
H_2	233,485	233,495	0,004	230,708	230,718	0,004
H_3	269,218	269,227	0,004	263,977	263,988	0,004
H_4	306,409	463,260	**51,190**	388,163	525,697	**35,432**
H_5	189,481	189,491	0,005	187,025	187,035	0,005
H_6	185,871	185,880	0,005	182,831	182,841	0,006
H_7	117,766	117,777	0,009	116,999	117,009	0,009
H_8	161,568	161,578	0,006	159,746	159,756	0,006
H_9	174,894	174,905	0,006	173,170	173,181	0,006
H_{10}	176,252	176,262	0,006	174,265	174,275	0,006
H_{11}	187,700	187,710	0,005	185,041	185,051	0,005
H_{12}	222,633	336,583	**51,183**	282,047	382,103	**35,475**
H_{13}	147,023	147,033	0,007	145,888	145,898	0,007
H_{14}	558,199	842,991	**51,020**	707,068	957,216	**35,378**
H_{15}	191,485	191,494	0,005	190,093	190,103	0,005

Table 6.2 – Données collectées après exécution des agents G et H, dont les distributions sont visibles en figures 6.8 et 6.9. L'étude de l'écart relatif R indique quelles EA peuvent être classées comme interférentes par rapport à la FLASH.

2. Inversement, les EA interférentes ont un WCET accru quand elles sont exécutées sur le cœur n° 2 par rapport au cœur n° 1. Cette observation est asymétrique vis-à-vis de la précédente, mais peut s'expliquer par une priorité plus grande du cœur n° 1 sur le cœur n° 2. Ainsi, un stress sur le cœur n° 2 semble plus marqué que sur le cœur n° 1.
3. La variabilité du WCET est moindre quand le cœur n° 2 est utilisé, par rapport au cœur n° 1. Cette observation est particulièrement intéressante par rapport à la précédente : sur le cœur n° 2, le WCET est accru, mais sa variabilité diminue. En fonction de ce que l'utilisateur cherche à montrer, il peut ainsi privilégier une stratégie plutôt que l'autre. Il peut privilégier le cœur n° 1 pour réduire le WCET mesuré, mais peut préférer le cœur n° 2 pour accroître sa confiance en l'analyse

du WCET.

6.3.8 Caractérisation logicielle conjointe FLASH-SRAM

Nous illustrons dans cette section une caractérisation logicielle des agents G et H en utilisant une combinaison de deux co-runners afin de stresser de manière conjointe la FLASH et la SRAM. On reprend un protocole de test similaire à celui établi en section 6.3.7, mais en utilisant le co-runner décrit en section 6.3.4 sur le cœur n° 0. On effectue ainsi les tests suivants.

a. L'agent testé est exécuté dans un environnement contrôlé permettant d'explorer ses branchements temporels et fonctionnels. Les temps d'exécution de chaque EA sont enregistrés pour son implantation sur le cœur n° 1, et aucun co-runner n'est actif (les cœurs n° 0 et n° 2 sont désactivés).

b. Le test précédent est répété, à la différence qu'un co-runner stressant la FLASH s'exécute sur le cœur n° 2, et qu'un autre stressant la SRAM s'exécute sur le cœur n° 0.

c. On répète le test (a), mais en changeant l'implantation : l'agent testé est exécuté sur le cœur n° 2, et aucun co-runner n'est actif (les cœurs n° 0 et n° 1 sont désactivés).

d. Le test (c) est répété, à la différence qu'un co-runner stressant la FLASH s'exécute sur le cœur n° 1, et qu'un autre stressant la SRAM s'exécute sur le cœur n° 0.

Les résultats sont présentés de manière analogue à la caractérisation logicielle vue en section 6.3.7. Les figures 6.10 et 6.11 montrent ainsi les distributions statistiques des temps d'exécution des agents G et H quand la FLASH et la SRAM sont stressées de manière conjointe. Les données quantitatives associées sont résumées en tableau 6.3.

On constate que les comparaisons des distributions statistiques n'ont que très peu évolué par rapport à la caractérisation logicielle précédente. Cela suggère que la ressource matérielle partagée qu'est la SRAM n'est que très peu utilisée. Toutefois, ce changement de co-runner a bien un impact sur les temps d'exécution mesurés : on observe dans le tableau 6.3 que davantage d'EA sont classées comme interférentes, et cette classification varie en fonction de la stratégie d'implantation des cœurs. Il semble en effet que l'activation du cœur n° 0 soit source d'interférences mineures et cette légère augmentation des WCET mesurés permet à certaines EA de dépasser la valeur de seuil h fixée précédemment. Toutefois, ces EA ont un WCET trois ordres de grandeur inférieur à celles précédemment observées. On peut donc les considérer comme non-interférentes au regard de la globalité de l'application.

6.3.9 Analyse des résultats et mitigations

L'étude des résultats de la caractérisation logicielle en sections 6.3.7 et 6.3.8 impliquant les deux ressources matérielles partagées nous permet d'identifier quelles EA accèdent à quelles ressources matérielles partagées, et ainsi d'établir des stratégies de mitigation des interférences temporelles.

Les données quantitatives montrées en tableaux 6.2 et 6.3 permettent de relever que les EA G_3, G_6, G_8, G_{14}, H_4, H_{12} et H_{14} sont fortement interférentes, quelle que soit

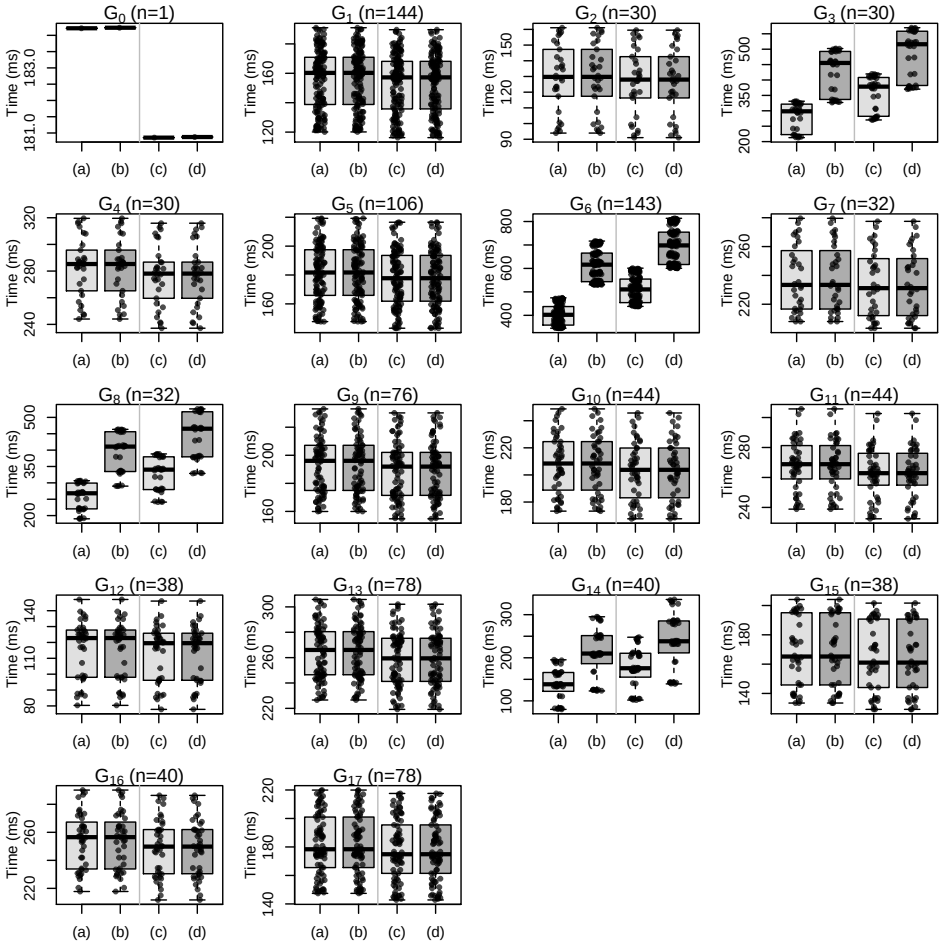


Figure 6.10 – Caractérisation logicielle de l'agent G avec stress conjoint de la FLASH et de la SRAM. n représente le nombre de mesures de temps d'exécution par boîte à moustaches et $G_{i|i\in\mathbb{N}}$ les EA constituant l'agent G.

la politique d'implantation sur les cœurs. On retrouve ces résultats, que la FLASH soit stressée seule, ou conjointement avec la SRAM. On observe que les temps d'exécution des autres EA sont dégradés, mais trois ordres de grandeur moindre, ce qui nous permet de considérer que ces EA sont suffisamment robustes aux interférences pour ne pas être la cible de mitigations.

Certaines mesures peuvent être difficiles à interpréter, mais s'expliquent à la lumière de ces deux caractérisations logicielles et grâce au schéma d'architecture matérielle du MPC5777M montré en figure 3.11. Ainsi, les EA non-interférentes ont un WCET mesuré moindre sur le cœur n° 2 que sur le cœur n° 1. Cela peut être surprenant au regard de la différence de cadencement des processeurs et aux mécanismes d'interconnexion du cœur n° 2 pour accéder aux ressources matérielles partagées. Cela peut s'expliquer d'une

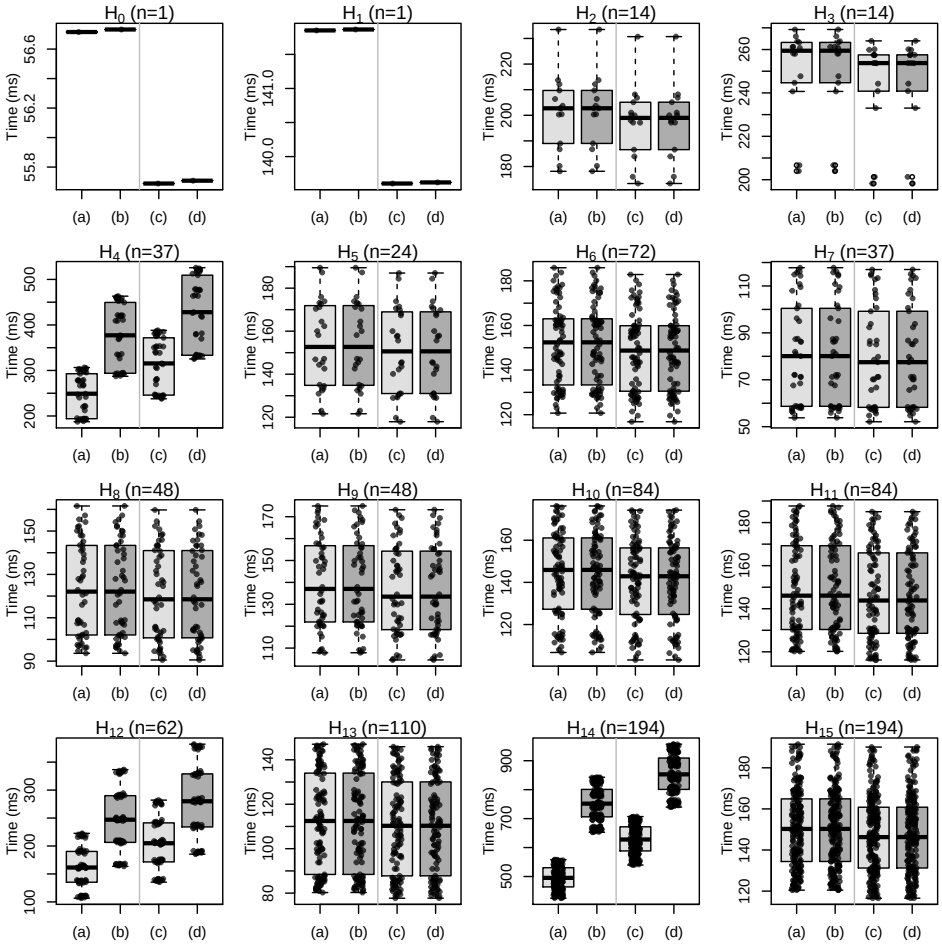


Figure 6.11 – Caractérisation logicielle de l'agent H avec stress conjoint de la FLASH et de la SRAM. n représente le nombre de mesures de temps d'exécution par boîte à moustaches et $H_{i|i\in\mathbb{N}}$ les EA constituant l'agent H.

part par le fait que le cœur n° 2 soit *dual-issue* alors que le cœur n° 1 est *single-issue*. D'autre part, la caractérisation logicielle stressant conjointement la SRAM et la FLASH montrent que la FLASH et la SRAM sont très peu utilisées par ces EA (trop peu pour observer des variations significatives de temps d'exécution). Ainsi, ces EA semblent davantage tirer parti des mémoires locales aux cœurs que des ressources matérielles partagées.

Cette analyse des interférences grâce à la caractérisation logicielle nous permet donc de décrire un groupe d'exclusion $\mathcal{G}$ regroupant les EA que nous avons jugées fortement interférentes : $\mathcal{G} = \{G_3, G_6, G_8, G_{14}, H_4, H_{12}, H_{14}\}$. On peut les représenter comme en figure 6.12, et mettre en application le principe de non-simultanéité décrit au chapitre 5. En l'occurrence, un recouvrement temporel entre les EA G_6 et H_4 est

	Cœur n° 1			Cœur n° 2		
EA	**max(a)** *(ms)*	**max(b)** *(ms)*	$R(a,b)$ *(%)*	**max(c)** *(ms)*	**max(d)** *(ms)*	$R(c,d)$ *(%)*
G_0	184,217	184,234	0,009	180,857	180,876	**0,010**
G_1	190,981	190,995	0,007	189,639	189,654	0,008
G_2	161,039	161,053	0,008	159,511	159,527	0,010
G_3	331,060	502,172	**51,686**	419,580	569,652	**35,767**
G_4	319,557	319,571	0,004	315,902	315,918	0,005
G_5	219,241	219,254	0,006	216,546	216,562	0,007
G_6	474,479	716,623	**51,034**	601,022	813,650	**35,378**
G_7	279,576	279,590	0,005	277,474	277,490	0,006
G_8	306,564	463,397	**51,159**	388,349	525,913	**35,423**
G_9	233,026	233,040	0,006	230,172	230,188	0,007
G_{10}	248,713	248,727	0,006	245,702	245,717	0,006
G_{11}	305,734	305,748	0,005	302,646	302,661	0,005
G_{12}	147,060	147,074	0,010	146,063	146,079	**0,011**
G_{13}	305,834	305,848	0,005	302,088	302,103	0,005
G_{14}	195,298	295,135	**51,121**	247,730	335,290	**35,345**
G_{15}	204,100	204,114	0,007	201,639	201,655	0,008
G_{16}	290,148	290,161	0,005	286,340	286,356	0,006
G_{17}	219,936	219,949	0,006	217,564	217,580	0,007
H_0	56.716	56.733	**0.030**	55.688	55.708	**0.034**
H_1	141.852	141.865	0.010	139.605	139.621	**0.012**
H_2	233.485	233.500	0.006	230.708	230.726	0.008
H_3	269.218	269.232	0.005	263.977	263.994	0.006
H_4	306.409	463.262	**51.191**	388.163	525.702	**35.433**
H_5	189.481	189.496	0.008	187.025	187.040	0.008
H_6	185.871	185.884	0.007	182.831	182.848	0.009
H_7	117.766	117.780	**0.012**	116.999	117.015	**0.014**
H_8	161.568	161.583	0.009	159.746	159.762	**0.010**
H_9	174.894	174.908	0.008	173.170	173.187	0.009
H_{10}	176.252	176.266	0.008	174.265	174.281	0.009
H_{11}	187.700	187.714	0.007	185.041	185.057	0.009
H_{12}	222.633	336.592	**51.187**	282.047	382.109	**35.477**
H_{13}	147.023	147.037	0.009	145.888	145.904	**0.011**
H_{14}	558.199	843.005	**51.022**	707.068	957.229	**35.380**
H_{15}	191.485	191.499	0.007	190.093	190.110	0.009

Table 6.3 – Données collectées après exécution des agents G et H, dont les distributions sont visibles en figures 6.10 et 6.11. L'étude de l'écart relatif R indique quelles EA peuvent être classées comme interférentes vis-à-vis de la FLASH et de la SRAM.

possible : le groupe d'exclusion $\mathcal{G}$ ne valide pas la propriété de sûreté que les EA qu'il contient ne peuvent s'exécuter simultanément. L'application que nous avons étudiée est donc sensible aux interférences temporelles causées par des accès simultanés aux ressources matérielles partagées identifiées. Le tableau 6.4 montre une trace d'exécution conduisant à une potentielle exécution simultanément, donnant un contre-exemple qui invalide le groupe d'exclusion.

À ce stade, on peut conserver la structure de l'application existante, mais à condition de provisionner suffisamment de temps pour l'exécution des EA accédant à la FLASH en prenant en compte les interférences inhérentes. Le but étant toutefois de mettre en œuvre une conception logicielle permettant une certaine robustesse aux interférences induites par les accès aux ressources matérielles partagées, nous pouvons essayer de

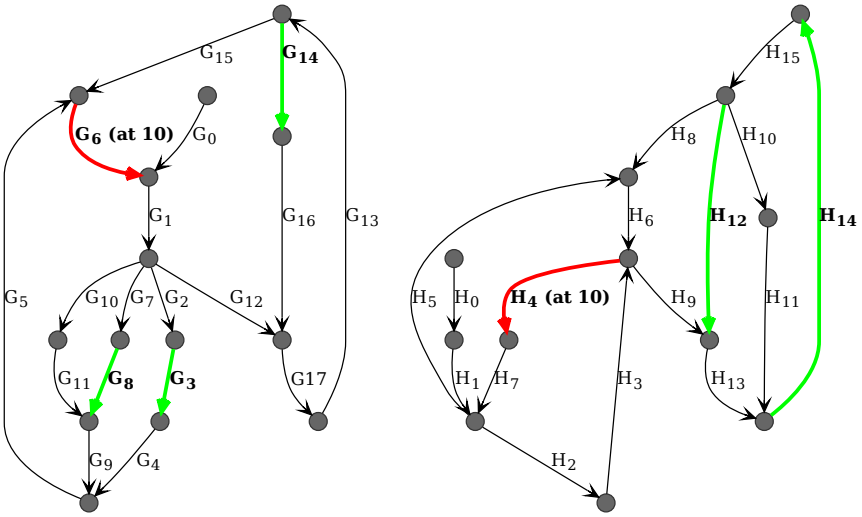


Figure 6.12 – Identification des EA interférentes du groupe d'exclusion $\mathcal{G}$ et vérification de leur non-recouvrement temporel. L'analyse montre que G_6 et H_4 peuvent être exécutées simultanément. La légende est identique à celle de la figure 5.6.

Tic	0	1	2	3	4	5	6	7	8	9	**10**
Agent G	G_0	G_1	G_{12}	G_{17}	G_{13}	G_{14}	G_{16}	G_{17}	G_{13}	G_{15}	$\boldsymbol{G_6}$
Agent H	H_0	H_1	H_2	H_3	H_9	H_{13}	H_{14}	H_{15}	H_8	H_6	$\boldsymbol{H_4}$

Table 6.4 – Contre-exemple montrant une trace d'exécution conduisant au recouvrement temporel des EA G_6 et H_4 au dixième tic d'horloge logique.

modifier l'application pour ce faire. Une première piste de recherche consiste à modifier le cadencement de l'application pour introduire davantage d'EA, afin d'affiner les contraintes d'exclusion. Une autre consiste à ne pas altérer le cadencement existant et chercher à réorganiser le code fonctionnel pour éviter les accès simultanés. Par exemple, on peut remarquer que l'EA G_6 est précédée par G_5 et G_{15}. Si l'on peut effectuer l'accès à la ressource matérielle partagée au cours de ces deux EA plutôt qu'en G_6, on obtient un nouveau groupe d'exclusion $\mathcal{G}' = \{G_3, G_5, G_8, G_{14}, G_{15}, H_4, H_{12}, H_{14}\}$, qui s'avère être valide, comme montré en figure 6.13. Ce type d'approche par tâtonnement cherche, par itérations successives, à mettre en œuvre une application qui soit la moins sensible possible aux interférences temporelles causées par les accès simultanés à des ressources matérielles partagées.

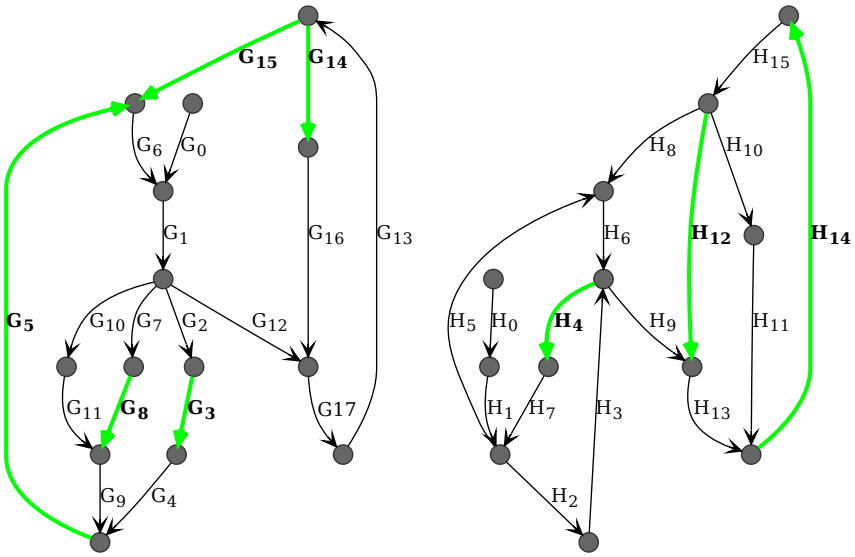


Figure 6.13 – Identification des EA du groupe d'exclusion $\mathcal{G}'$ et vérification de leur non-recouvrement temporel. L'analyse permet de montrer l'absence de recouvrement. La légende est identique à celle de la figure 5.6.

6.4 Conclusion

Nous avons présenté dans ce chapitre une méthode originale permettant de caractériser la plateforme matérielle ainsi que le logiciel qui y est implanté. En exploitant le principe des co-runners, adapté aux applications PsyC d'Asterios, nous pouvons identifier la sensibilité des EA aux ressources matérielles partagées par l'étude de distributions statistiques de temps d'exécution mesurés. Leur analyse quantitative et qualitative permet aux concepteurs d'applications de classifier certaines EA comme sensibles aux interférences vis-à-vis d'une ressource matérielle partagée identifiée. Ces données permettent de construire des groupes d'exclusion, pouvant bénéficier du principe de non-simultanéité que nous avons détaillé au chapitre 5. Celui-ci permet de vérifier hors ligne une propriété de sûreté garantissant que les EA d'un même groupe d'exclusion ne peuvent s'exécuter simultanément. Ces informations permettent de construire une application robuste à ce type d'interférences temporelles, en cherchant à minimiser l'occurrence des interférences. De plus, cette méthode permet aux architectes de mieux appréhender le comportement effectif du matériel, qui peut être parfois contre-intuitif.

Cette méthode présente toutefois quelques inconvénients et limitations. Elle nécessite l'écriture de tests pleinement couvrants pour que tous les chemins fonctionnels et temporels puissent être explorés. Si cette exigence semble raisonnable pour des systèmes critiques soumis à de fortes contraintes de certification, elle n'est reste pas moins

laborieuse à mettre en place. D'autre part, elle ne permet pas d'adresser les ressources matérielles partagées de manière automatique, telles que les caches partagés entre plusieurs cœurs (p. ex. L2, L3). Aussi, dans l'état actuel de nos travaux, ceux-ci doivent être désactivés ou configurés de telle sorte à ne pas être sources d'interférences cœur à cœur (p. ex. partitionnés ou utilisés comme mémoires annexes). Les interférences habituelles du monde monocœur telles que celles causées par les caches de premier niveaux doivent être gérées en amont par les utilisateurs.

Malgré ses limitations, nous pensons que cette méthode permet à l'utilisateur d'élaborer une stratégie d'analyse temporelle dans un environnement multicœur permettant de répondre, au moins partiellement, aux objectifs de certification attendus. Sa prochaine application à des projets industriels plus ambitieux que notre preuve de concept devrait éprouver sa viabilité.

Ce chapitre conclut nos travaux s'attaquant spécifiquement au problème de l'utilisation de cœurs d'exécution simultanés pour des applications temps-réelles de sûreté. La partie III qui suit propose quelques pistes de recherche plus générales, applicables également au modèle monocœur, dont peuvent bénéficier les travaux de cette partie.

III

Troisième partie

Vers la mise en place systématique de mitigations

Chapitre 7

Analyse par exécution concolique

Sommaire

7.1 Motivations

Nous avons rappelé dans les chapitres précédents — en particulier en partie I — l'importance du WCET au sein des systèmes temps-réels de sûreté. L'utilisation des mémoires mises à disposition par le matériel influe considérablement sur cette métrique.

Dans la partie II, nous nous sommes intéressés à leur usage dans un contexte multicœur. Ainsi, les accès simultanés aux mémoires partagées dégradent de manière importante les WCET des tâches temps-réelles. Nous avons présenté une méthode (chapitre 5) visant à concevoir des applications temps-réelles pour minimiser l'occurrence de ces accès simultanés, permettant même de garantir leur absence. Si cette méthode s'appuie sur des outils formalisés pour produire des résultats corrects, il est indispensable que l'utilisateur fournisse des données d'entrées exactes et complètes. Dans le cas contraire, l'analyse que nous effectuons pourrait ne pas être en lien avec le système réel. Afin d'aider l'utilisateur dans cette tâche, nous avons mis au point une méthode d'analyse expérimentale (chapitre 6). Toutefois, elle ne peut être mise en œuvre que tardivement dans le cycle de développement logiciel. Aussi, une analyse hors-ligne équivalente, permettant d'identifier les accès mémoires réalisés par les tâches temps-réelles, serait avantageuse.

L'analyse des accès mémoires effectués par une application a donné lieu à profusion de travaux dans divers domaines de l'informatique. Les avantages retirés sont multiples : une connaissance fine de l'utilisation de la mémoire d'un logiciel permet notamment une meilleure exploitation des caches matériels. Les technologies de compilation, d'une manière générale, visent ainsi à réorganiser le code et les données afin de maximiser les performances du logiciel. Du point de vue des systèmes temps-réels de sûreté, il s'agit particulièrement de réduire le WCET ainsi que la variabilité des temps d'exécution. Deux problèmes distincts peuvent être identifiés.

1. Comment mettre en œuvre une stratégie de placement du code et des données efficace (voire optimale) ?
2. Comment collecter les informations nécessaires et suffisantes permettant l'utilisation de ces stratégies de placement mémoire ?

Dans nos travaux, nous nous concentrons particulièrement sur ce second point : nous mettons en œuvre une technique d'analyse des accès mémoires effectués. Il est difficile d'estimer avant l'exécution d'un programme quelles instructions seront exécutées, et dans quel ordre. De même, déterminer les séquences de lecture et d'écriture des données hors ligne est complexe [PR02]. Le premier point n'est pas traité ici, mais de futurs travaux pourront y revenir. Aussi, nous ne nous attardons pas sur les multiples techniques de placement dont la littérature foisonne. Nous notons toutefois que ce problème n'a pas de solution optimale dans le cas général, à moins que $P = NP$ ne soit démontré [PR02]. Il est toutefois possible d'utiliser des heuristiques, l'objectif étant d'obtenir un *meilleur* résultat par rapport à un placement naïf.

Ces travaux n'ont pas encore été publiés [GV21]. Ils ont été soumis à une conférence et nous sommes dans l'attente.

7.2 Revue de stratégies de mitigation

Comme décrit en section 7.1, la méthode d'analyse que nous détaillons dans ce chapitre vise à déterminer un modèle des accès mémoires effectués par une application temps-réelle. Nous laissons le développement de stratégies de mitigation exploitant notre analyse pour de futurs travaux. Nous effectuons ici une brève revue de techniques existantes, qui pourraient être améliorées grâce aux données que notre analyse collecte.

On retrouve ainsi les techniques de mitigation développées pour le multicœur, déjà décrites en section 2.4.3, comme le *Single Core Equivalence framework*, le modèle d'exécution PREM ou encore les points d'intérêt temporels. Une connaissance fine des accès mémoires effectués permettrait ainsi de faciliter leur mise en œuvre ou d'améliorer leur efficacité. Nos propres travaux sur le principe de non-simultanéité, décrits au chapitre 5 pourraient également tirer parti de cette technique d'analyse. En effet, comme tous les accès mémoires peuvent être identifiés, les mémoires matérielles sollicitées pour chaque EA sont détectées automatiquement. Cette information permet de générer automatiquement des groupes d'exclusion (section 5.2.3) permettant d'éprouver facilement le principe de non-simultanéité sans nécessairement recourir à des approches expérimentales, comme celle des co-runners décrite au chapitre 6.

WCC, une infrastructure de compilation sensible au problème du WCET, cherche à intégrer des outils d'analyse du WCET pour intégrer à son processus de compilation des informations-clés permettant d'améliorer le WCET des programmes que cette infrastructure compile [FLT06 ; FL10]. Cet outil est notamment compatible avec aiT, déjà mentionné en section 2.3.2. Les binaires générés après une première compilation sont inspectés et les résultats de cette analyse servent ainsi d'entrées à des compilations successives. Compléter les informations obtenues avec le modèle des accès mémoires que nos travaux permettent d'extraire hors ligne pourrait avantageusement compléter ce modèle existant.

Finalement, des stratégies *ad hoc* peuvent être mises en place en jouant sur le positionnement mémoire, afin de tirer parti au mieux des différents caches matériels. Nos précédents travaux sur le partitionnement mémoire, décrits en chapitre 4 proposent une implémentation d'un tel mécanisme de placement. Des approches de partitionnement de caches peuvent également être suivies, comme celles déjà présentées en section 6.2.6. Si le partitionnement de caches a souvent été évoqué dans la littérature, sa mise en œuvre est délicate à cause du manque d'informations sur les accès mémoires effectués à l'exécution. Nous espérons que notre technique d'analyse permettra d'élaborer des stratégies validant les approches de partitionnement de caches à plus grande échelle.

7.3 Techniques d'analyse de programmes

Cette section passe en revue quelques techniques d'analyse de programme. Nous posons ainsi les bases de la technique retenue dans nos travaux, qui s'apparente à du DSE (Dynamic Symbolic Execution) binaire.

7.3.1 Choix de l'analyse statique

L'objectif principal des travaux décrits dans cette partie consiste à capturer tous les accès mémoires possibles réalisés par les tâches composant un système étudié. Nous considérons qu'une tâche peut s'exprimer comme une suite de calculs préemptifs, qui s'exécutent sur un RTOS effectuant des opérations de changement de contexte pour assurer l'élection des différentes tâches. Cette définition est compatible avec le cadre d'Asterios qui a été décrit au chapitre 3. Nous utilisons ainsi une stratégie que l'on pourrait qualifier de « RTOS-in-the-loop » pour faire un parallèle avec les approches MIL (Model-In-the-Loop), HIL (Hardware-In-the-Loop) et SIL (Software-In-the-Loop) bien

connues de l'industrie automobile [NHC12]. Ainsi, notre collecte de données prend en compte le RTOS sur lequel les tâches étudiées s'exécutent. Ceci devrait permettre aux algorithmes tiers exploitant les données produites de générer des stratégies de placement prenant non seulement en compte les accès du code utilisateur, mais également des services du RTOS, ce qui leur confère un potentiel d'efficacité plus important.

Le comportement du logiciel peut être analysé par l'observation de son exécution dans des conditions réelles de son utilisation, c'est-à-dire quand il s'exécute sur du matériel. Cette méthode empirique très répandue est facilitée par les moniteurs matériels, dont des abstractions logicielles simplifient l'utilisation [Ter+10]. Des méthodes telles que le PGO (Profile-Guided Optimization) tirent notamment parti de ce genre de techniques afin de générer des exécutables plus adaptés à leur architecture d'implantation [PH90]. On peut noter que cette stratégie est toujours utilisée et se perfectionne. Des outils comme Bolt, par exemple, sont développés par des géants du numérique, afin d'améliorer les performances à grande échelle des programmes s'exécutant dans des *data center*, où tous les gains s'additionnent pour devenir considérables [Pan+19]. Toutefois, nous pensons que ces approches basées sur l'observation de l'exécution d'un système se prêtent peu aux systèmes embarqués, qui nécessitent une exécution sur du matériel déporté. En particulier pour la collecte exhaustive des accès mémoires effectués. En effet, certains composants micro-architecturaux interfèrent avec les accès mémoires réalisés. Par exemple, les caches masquent les accès à des plages d'adresses résidant déjà dans ces caches. Ils peuvent également engendrer des accès mémoires inopinés lors de l'éviction des lignes de cache quand ils sont configurés en *write-back*. De plus, explorer l'entièreté des accès mémoires réalisables pour chaque branchement du code requiert un effort de couverture bien supérieur à celui habituellement attendu des systèmes temps-réel de sûreté. Aussi, notre approche se concentre sur l'analyse statique des programmes, permettant de s'affranchir du matériel et de tests de couverture fastidieux à concevoir.

7.3.2 Interprétation abstraite

L'*interprétation abstraite* est une méthode formelle visant à déterminer une surapproximation de la sémantique d'un programme afin d'élaborer des preuves correctes (*sound*) sur les propriétés dynamiques de ce programme [CC77]. Rappelons que la sémantique désigne ici l'ensemble de tous les états possibles du programme, quelles que soient ses entrées [CC92].

Cette approche a reçu un grand intérêt de la communauté des systèmes temps-réels étudiant le WCET [FW98]. La plupart des outils d'analyse statique du WCET évoqués en section 2.3.2 se basent ainsi sur cette technique [Wil+08]. Ils disposent souvent d'un composant clef de leur infrastructure nommé *value analysis*, permettant de déterminer les adresses physiques atteignables par le code [CBS13 ; FH05]. Associé avec une analyse du flot de contrôle, il offre une vue d'ensemble des accès mémoires effectués par le programme. Grâce à l'interprétation abstraite, cette analyse est *sound* : aucun accès mémoire n'en est exclu. Toutefois, cette approche manque généralement de précision, la *soundness* étant recherchée. Nous ne cherchons cette propriété : comme évoqué en section 7.1, les mitigations pouvant être mises en œuvre à partir des données collectées ne peuvent généralement pas être optimales. Aussi, nous pensons qu'il vaut

mieux privilégier la précision des données, au détriment de la *soundness*, quitte à sous-approximer les accès mémoires. De plus, à notre connaissance, les outils existants excluent les accès effectués par le RTOS sous-jacent, ce qui ne permet que d'avoir une vision partielle du système final.

7.3.3 Exécution symbolique

L'*exécution symbolique* est une autre technique d'analyse de programmes [Kin76]. Elle présente des similarités avec l'interprétation abstraite présentée en section 7.3.2, mais leurs objectifs sont différents. En effet, si l'interprétation abstraite recherche la *soundness*, l'exécution symbolique s'intéresse à la *reachability* — c'est-à-dire trouver les conditions nécessaires à l'activation d'un branchement particulier [Ama+20].

Cette technique a été utilisée à l'origine dans l'automatisation de tests de couverture [CS13]. Des outils comme KLEE [CDE08] permettent de s'intégrer à des infrastructures de compilation comme LLVM [LA04] pour générer des cas de tests permettant de tester la quasi-totalité des branchements. Ces principes sont réutilisés dans le cadre de la sécurité informatique avec des outils comme BINSEC/SE [Dav+16], Mayhem [Cha+12] ou Triton [SS15]. Ceux-ci permettent, par exemple, d'inspecter des programmes malveillants ou de recueillir les prérequis permettant l'activation de fonctions-clés.

Le cœur de l'exécution symbolique consiste à maintenir un contexte symbolique [1] décrivant les états des variables durant l'exécution, ainsi qu'à explorer séquentiellement le programme testé jusqu'à rencontrer un branchement. Chaque branche est explorée, souvent grâce à un mécanisme de *fork*. Une implémentation naïve mène inexorablement au problème de l'explosion des états du système, limitant considérablement l'utilisation de l'exécution symbolique « pure ». Des stratégies de fusion ont été développées afin de limiter le nombre d'états à explorer [Kuz+12]. Elles peuvent être couplées avec des mécanismes de *concrétisation* [2], permettant aux valeurs symboliques d'être distinguées des valeurs dites *concrètes* — c'est-à-dire que ces valeurs sont connues. Une exécution combinant valeurs concrètes et valeurs symboliques est qualifiée de **concolique** [3]. Nos travaux s'appuient fortement sur ce principe. L'exécution symbolique — et par extension, l'exécution concolique — n'apporte pas de propriété de *soundness* contrairement à l'interprétation abstraite. Toutefois, cette technique permet une plus grande précision dans la collecte de données, ce qui motive son utilisation pour la conception de stratégies de placement mémoire.

7.3.4 Nature du programme analysé

Les techniques d'analyse statique vues plus haut peuvent opérer au niveau du code source (p. ex. typiquement du C, C++ ou de l'ADA dans les systèmes temps-réels de sûreté) ou au niveau du binaire compilé, obtenu après complétion des opérations de compilation et d'édition de liens. L'analyse du binaire final présente certains avantages par rapport à une analyse basée uniquement sur le code source. En effet, les compilateurs et éditeurs de liens sont susceptibles d'ajouter des modifications structurelles au programme,

1. *Symbolic store* en anglais.
2. On utilise ici cet anglicisme, venant de l'anglais *concretization*.
3. Ce terme étrange est un anglicisme du mot-valise *concolic* où « conc- » et « -olic » sont issus respectivement des mots anglais *concrete* et *symbolic*.

modifiant le comportement décrit au niveau du code source. Si les optimisations y contribuent pour une grande part [Dar+19], des modifications restent observables quand les optimisations sont désactivées. Par exemple, les opérations arithmétiques entières sur 64-bits exécutées sur des plateformes 32-bits peuvent nécessiter l'ajout de branchements additionnels par le compilateur, qu'une analyse seule du code source ne peut anticiper. À moins de disposer d'un modèle précis de génération du code objet à partir du code source, une analyse basée exclusivement sur le code source n'est pas garantie de capturer le comportement exact du programme. Toutefois, des avancées récentes dans les technologies de compilation certifiées, comme CompCert [Käs+18], devraient permettre de répondre à ces problèmes. Ces approches sont toutefois peu répandues et semblent être encore dépendantes de l'éditeur de liens. Or, comme vu en section 4.3.4, les éditeurs de liens sont susceptibles de modifier le code objet généré par le compilateur.

De plus, les processus industriels évoqués en section 2.1 peuvent être incompatibles avec des analyses basées uniquement sur le code source. En effet, le système logiciel d'une application industrielle est généralement découpé en sous-composants, confiés à des sous-traitants. Ce logiciel est souvent fourni sous la forme d'objets compilés, qui sont agrégés durant la phase d'intégration. Ainsi, dans la pratique, il est peu aisé de généraliser les analyses de sources à l'ensemble des entités impliquées dans la création du logiciel final.

Pour toutes les raisons décrites plus haut, nous nous écartons de l'analyse de code source, au profit de l'analyse du binaire final. Il est produit par l'intégration, au sein de l'entité maîtresse d'œuvre, ce qui facilite son utilisation. De plus, le binaire final décrit le plus précisément les instructions devant être exécutées sur le matériel — sous réserve qu'il soit correctement implanté et ne soit pas corrompu ou réécrit à l'exécution. C'est donc l'objet nécessitant le moins d'hypothèses pour justifier de sa représentativité de l'exécution. Toutefois, le code objet analysé perd grandement de la sémantique originalement présente dans le code source, ce qui complexifie le travail d'analyse. De plus, si l'analyse binaire garantit une indépendance vis-à-vis des langages de programmation de plus haut niveau, elle requiert d'implémenter un mécanisme de décodage des formats binaires et des instructions. Ces dernières, plus généralement désignées comme ISA (Instruction Set Architecture), dépendent de l'architecture matérielle choisie. Toutefois, comme notre approche cible particulièrement l'industrie des systèmes temps-réels de sûreté, nous estimons que l'analyse binaire offre plus d'avantages pratiques que l'analyse basée sur les sources.

7.4 Modèle du système

7.4.1 Hypothèses sur le logiciel

Extraire le modèle des accès mémoires d'une application composée de multiples tâches est un problème complexe, probablement irréaliste dans le cas de logiciel non spécialisé. Cela est dû à l'écrasante présence d'exécution de code non contrôlée. Ce code est généralement trop permissif, parfois mal formé (son comportement est non défini) et quelques fois malicieux. De plus, comme ce type d'analyse requiert une forme d'exploration du flot de contrôle, celles-ci sont indécidables, comme nous le voyons en

section 7.5.6.

Toutefois, comme nos travaux visent spécifiquement les systèmes temps-réels de sûreté, nous pouvons restreindre considérablement le domaine d'applications à analyser, en nous reposant sur certaines hypothèses. Ces hypothèses sont probablement déraisonnables dans le cas de logiciel grand public, mais en nous appuyant sur nos contraintes de sûreté de fonctionnement, nous estimons qu'elles sont acceptables. Seules quelques exceptions pourront être tolérées. Celles-ci devront être dûment justifiées et gérées spécifiquement, au cas par cas.

Hypothèse 1 (Comportements indéfinis)**.** *L'exécution symbolique d'un programme construit sur la base de comportements indéfinis peut conduire à l'obtention de résultats non exploitables. La figure 7.1 propose un exemple de programme écrit en C comportant des comportements indéfinis. On fait ainsi l'hypothèse que les programmes que nous examinons ne présentent pas de comportements indéfinis, car des outils d'analyse statique tels qu'Astrée permettent de s'en assurer [Käs+10].*

```
1 signed int function(signed int value) {
2   signed int shift_count;
3   return value >> shift_count;
4 }
```

Figure 7.1 – Exemple de programme C invalide vis-à-vis de hypothèse 1. D'une part la variable `shift_count` (ligne 2) n'est pas initialisée : sa valeur à l'exécution n'est pas connue. D'autre part, l'opération de décalage logique (ligne 3) est indéfinie sur des entiers signés. Le résultat de l'exécution de ce code est donc indéfini.

Hypothèse 2 (Accès mémoires bornés)**.** *Un accès mémoire invalide peut causer une faute matérielle si une forme de protection mémoire est activée à l'exécution (auquel cas, il est probable que la tâche fautive soit terminée). Cet accès mémoire invalide peut aussi engendrer un comportement indéfini (ce qui contrevient à l'hypothèse 1). Les accès mémoires sont souvent minutieusement étudiés afin de prévenir l'occurrence des accès invalides. Nous estimons que leur survenue est liée à une défaillance du logiciel et ne représente pas un comportement nominal. Le figure 7.2 montre un exemple de cette pratique.*

Hypothèse 3 (Protection mémoire $W \oplus X$ persistante)**.** *Le $W \oplus X$ (ou write xor execute) est une pratique standard de sécurité permettant d'assurer que des zones mémoires inscriptibles ne puissent pas être exécutées. Cela permet de limiter l'exécution de code arbitraire. Nous faisons l'hypothèse que ce mécanisme est systématiquement présent, en particulier pour interdire l'utilisation de code automodifiant. Ce dernier pourrait en effet nuire à notre analyse.*

Hypothèse 4 (Boucles bornées)**.** *Les boucles* non *bornées posent souvent problème lors de l'analyse de programmes dans lesquels la connaissance du nombre maximal d'itérations est importante. Les programmes d'analyse du WCET se reposent souvent sur des mécanismes d'annotations quand celles-ci ne peuvent être déterminées automatiquement*

```
extern int data[MAX_SAMPLES];
// Lecture d'une donnée inconnue à la compilation, dont la
// plage fonctionnelle réside dans [int_min, int_max]
const int value = read_sensor();

// Code non conforme de l'hypothèse 2:
/* (1) */ data[value] += 1;
// Code conforme de l' hypothèse 2 :
/* (2) */ data[value % MAX_SAMPLES] += 1;
```

Figure 7.2 – Exemple de programme C illustrant l'hypothèse 2. En (1), un accès illégal à la mémoire peut être produit si `value` $\notin [0, \text{MAX\_SAMPLES}[$. En (2), le code est écrit de sorte à prévenir ces erreurs.

[HRP17]. Dans le cadre de l'exécution concolique présentée dans ce chapitre, une boucle est considérée comme non bornée quand sa condition d'arrêt est susceptible d'être inconnue (quand sa valeur est symbolique). Dit autrement, les boucles non bornées désignent des boucles dont la condition d'arrêt ne peut être rendue concrète. Le figure 7.3 illustre ce type de boucles. Nous faisons l'hypothèse que le programme est constitué de boucles bornées.

```
// Lecture d'une donnée inconnue à la compilation, dont la
// plage fonctionnelle réside dans [int_min, int_max]
const int data = read_sensor();
// Lecture d'une donnée dont la valeur est forcément connue
const int concrete_data = get_concrete();

// (1) Boucle avec valeur symbolique comme condition d'arrêt
for (int i = 0; i < data & 0xF; i++) { /* ... */ }
// (2) Boucle avec valeur concrète comme condition d'arrêt
for (int i = 0; i < concrete_data; i++) { /* ... */ }
```

Figure 7.3 – Exemple de programme C illustrant l'hypothèse 4. En (1), la condition d'arrêt de la boucle est composée d'une valeur symbolique (quatre bits symboliques). Le programme d'analyse ne peut déterminer le nombre d'itérations au moment de l'exécution : la boucle est considérée comme non bornée. En (2), la condition d'arrêt est toujours connue (même si ce n'est pas une constante). Elle est donc considérée comme bornée.

Hypothèse 5 (Appels symboliques)**.** *On fait l'hypothèse qu'à tout moment de l'exécution, les instructions sont chargées depuis des adresses mémoires concrètes (non symboliques). Dans le cas contraire, il serait impossible d'en déduire la nature de l'instruction à exécuter. Le figure 7.4 montre un exemple de violation de cette hypothèse.*

```
// Tableau de NB_FUNCTIONS pointeurs sur fonction
typedef void( *f_ptr )(int);
extern f_ptr functions[NB_FUNCTIONS];

// Lecture d'une donnée inconnue à la compilation, dont la
// plage fonctionnelle réside dans [int_min, int_max]
const int data = read_sensor();

// (1) Récupération d'un pointeur de fonction
const f_ptr func = functions[data % NB_FUNCTIONS]
// (2) Exécution du pointeur de fonction symbolique
func(data);
```

Figure 7.4 – Exemple de programme C illustrant l'hypothèse 5. En (1), un pointeur de fonction est récupéré, mais son adresse n'est pas concrète. En effet, n'importe quel pointeur de fonction du tableau `functions` est candidat potentiel. L'appel réalisé en (2) ne peut donc aboutir à un appel de fonction concret.

7.4.2 Modèle d'exécution des tâches

Notre méthode d'analyse cible particulièrement les applications Asterios, dont les principes fondateurs sont rappelés au chapitre 3. Ces applications sont donc constituées d'agents, développés selon le modèle de programmation PsyC, qui s'exécutent sur un RTOS particulier. On considère que les agents communiquent entre eux grâce aux variables temporelles, détaillées en section 3.3.3.

Nous illustrons la gestion des agents PsyC par notre technique d'analyse en partant de la figure 7.5. Celle-ci montre un agent PsyC consultant une VT et utilisant son contenu comme paramètre de son code fonctionnel. Comme montré plus tôt en figure 3.1, les fichiers PsyC sont compilés par l'outil `psyko` de la chaîne de compilation Asterios. Ceux-ci sont translittérés en fichiers C, ce qui permet leur transformation en fichiers objets par des compilateurs sur étagère choisis par les utilisateurs. La figure 7.6 montre une possible transformation de l'agent. Deux appels de fonction sont insérés dans le code C généré :

— `ast_tv_consult()` permet de consulter la VT indiquée avec une profondeur d'historique donnée. Le contenu de la VT est retourné par cette fonction.

— `ast_advance()` rend la main à l'ordonnanceur du RTOS. Ceci indique que le code utilisateur a terminé son EA en cours et respecte ainsi son contrat temporel (c'est-à-dire que le code a complété avant son échéance).

Ce code, à cause de la boucle infinie qu'il présente, semble montrer une contradiction flagrante avec l'hypothèse 4. Ce cas particulier est détaillé plus tard, en section 7.5.6.

Afin de faciliter notre analyse des applications PsyC, nous restreignons les capacités du modèle d'exécution, c'est-à-dire des fonctionnalités dynamiques dont le RTOS pourrait disposer. Notons qu'Asterios les respecte déjà.

Hypothèse 6 (Ensemble figé de tâches)**.** *Chaque agent PsyC décrit exactement une tâche temps-réel. Le RTOS ne doit exécuter que des tâches parmi un ensemble fixe d'agents. Cela implique que les tâches ne peuvent être supprimées et créées dynamiquement, en dehors les conditions d'erreur et du démarrage.*

```
// Déclaration de la source et des horloges logiques utilisées
source realtime;
clock clk_command = 2 * realtime;
clock clk_sensor = 1 * realtime;

// Déclaration d'une variable temporelle produite à chaque tic de
// l'horloge logique clk_sensor
temporal double plane_attitude with clk_sensor;

// Agent PsyC lisant la variable temporelle et utilisant son contenu
// pour exécuter du code fonctionnel.
agent command(uses realtime, starttime 2 with clk_sensor)
{
  consult 1 $ plane_attitude;
  body start {
    functional_code($[0]plane_attitude);
    advance 1 with clk_command;
  }
}
```

Figure 7.5 – Définition d'un agent PsyC consultant une VT et utilisant son contenu comme entrée de son code fonctionnel.

```
// Point d'entrée de l'agent compilé en C
void psyc_command(void) {
  // Le « body start » est transformé en une boucle infinie
  for (;;) {
    // La lecture de la variable temporelle et l'instruction advance
    // sont traduites en des appels à des fonctions C.
    // La nature exacte de leurs paramètres n'est pas renseignée car
    // dépend de l'implémentation d'Asterios.
    // Ces paramètres sont garantis exempts d'effet de bord.
    functional_code(ast_tv_consult(/* ... */));
    ast_advance(/* ... */);
  }
}
```

Figure 7.6 – Transformation du code PsyC de l'agent montré en figure 7.5 par le compilateur PsyC.

Hypothèse 7 (Démarrage des tâches)**.** *Les tâches sont exécutées grâce à des contextes d'exécution. Avant qu'une tâche ne soit ordonnancée pour la première fois, le RTOS doit préparer un contexte d'exécution permettant le démarrage de la tâche concernée. L'adresse à laquelle ce code de démarrage réside doit être connue de notre technique d'analyse.*

Hypothèse 8 (Fin des tâches)**.** *Une tâche peut être terminée uniquement en cas d'erreur, c'est-à-dire lorsqu'une faute est détectée. Le déclenchement de ce mécanisme d'arrêt, fourni par le RTOS, doit être identifiable par notre technique d'analyse afin de détecter les branchements menant aux cas d'erreurs. Une tâche ne peut être restaurée après son arrêt.*

Hypothèse 9 (Clouage des cœurs)**.** *Chaque tâche est assignée statiquement à un unique cœur processeur à la compilation. Ce mécanisme interdit l'utilisation de toutes stratégies de migration de tâche, généralement utilisées par des noyaux grand public (p. ex. Linux) [Alm+10].*

Hypothèse 10 (Initialisation)**.** *Le RTOS exécute un certain nombre de routines permettant d'initialiser le système. Elles doivent être exhaustivement documentées afin que notre technique d'analyse puisse effectuer des opérations similaires sans nécessiter un modèle fin du matériel. Par exemple, la configuration de contrôleurs d'interruptions n'est pas indispensable pour l'analyse du programme.*

Hypothèse 11 (Allocations dynamiques)**.** *Les programmes grand public utilisent fréquemment une zone mémoire de taille variable, utilisée dynamiquement et au gré du programme : le* tas*. Si des allocations dynamiques sont parfois utilisées dans les systèmes de sûreté, celles-ci sont réalisées depuis des zones mémoires bien identifiées dont la taille est fixée à la compilation. Aussi, nous n'autorisons que cette forme d'allocation dynamique.*

7.5 Méthode d'exécution concolique

7.5.1 Complémenter l'émulation par une exécution symbolique

En section 7.3.3, nous mentionnons que l'exécution concolique est grossièrement une forme d'exploration symbolique souffrant dans une moindre mesure du problème de l'explosion des états. L'utilisation de l'opération de *concrétisation* le permet en transformant certaines variables inconnues (symboliques) en valeurs fixes et connues (concrètes). Notre approche utilise un angle d'attaque légèrement différent : nous nous reposons d'abord sur l'*émulation* (donc des valeurs concrètes) avant de nous orienter vers une exécution symbolique dans certains cas.

L'émulation est une technique permettant d'interpréter les instructions d'un ISA non natif (c.-à-d. quand l'ISA du programme analysé est différent de celui de la machine exécutant le logiciel analyseur). De très nombreux outils comme QEMU [Bel05], Valgrind [NS07], gem5 [Bin+11], UNISIM [Aug+07] — et bien d'autres — réalisent cette fonction. Ils permettent ainsi d'exécuter nativement des programmes écrits pour des architectures très hétérogènes. Ce processus souffre généralement de faibles performances dûes au surcoût des opérations à effectuer. Des systèmes de caches d'instructions et de compilation *just-in-time* (comme le fait QEMU) permettent généralement d'offrir de meilleures performances. Grâce à cette technique, nous pouvons décoder et exécuter des instructions à la volée, tout en maintenant les états du processeur émulé — ou Vcpu (Virtual CPU) — en charge de l'exécution du code des tâches analysées. L'exécution symbolique n'est invoquée que lorsque l'émulation rencontre des valeurs « inconnues ». Cela rapproche nos travaux des approches choisies par QSYM [Yun+18] et SymQEMU [PF21], qui ne partagent toutefois pas nos objectifs.

L'émulation est limitée par les valeurs récupérées depuis l'environnement externe, c'est-à-dire l'ensemble inconnu à la compilation du programme, comme les entrées-sorties. Seules les tailles (plages numériques) de ces variables sont parfaitement connues. Par exemple, la lecture d'un mot de 32-bits d'une entrée-sortie accessible *via* l'espace

d'adressage dispose de 2^{32} combinaisons. Cette information ne permet que de faiblement restreindre la plage des valeurs possibles pour une variable symbolique. Appliqué au modèle Asterios, les communications entre tâches sont considérées comme étant des lectures et écritures de l'environnement externe et ne peuvent être concrètes. En effet, comme les tâches sont exécutées indépendamment les unes des autres, les valeurs produites par un agent T_1 ne peuvent être transmis de manière sûre vers une tâche T_2.

La lecture de l'environnement externe pourrait être simulée par des émulateurs matériels. Ce sont des fragments de code particuliers fournissant des valeurs concrètes quand certaines plages mémoires sont accédées. Toutefois, ces résultats sont concrets par essence. Aussi, utiliser ce mécanisme pour explorer exhaustivement tous les branchements et accès mémoires qui en découlent requerrait un effort de test considérable. L'espace d'exploration résultant rend en pratique cette approche difficile à mettre en œuvre.

L'exécution symbolique permet de pallier cette limitation de l'émulation. Elle est sollicitée quand l'émulation accède à l'environnement externe, ce qui est déterminé par le moteur concolique qui supervise l'exécution du programme. Elle repose sur les concepts suivants, détaillés plus tard :

— la gestion des contextes symboliques (section 7.5.3) ;

— la résolution des adresses symboliques (section 7.5.4) ;

— l'exploration des branchements symboliques (section 7.5.5).

L'idée générale derrière notre mécanisme d'exécution concolique peut être résumée comme suit. Un émulateur binaire spécialisé permet l'exécution du programme à analyser tout en suivant son flot de contrôle (section 7.5.2) au fur et à mesure du décodage et de l'interprétation des instructions qui le composent. Grâce à l'exécution symbolique, nous pouvons collecter tous les accès mémoires réalisables par le programme, pour chaque branche atteignable. Après l'arrêt de l'exécution concolique, détaillée en section 7.5.6, les traces d'exécution produites par notre technique d'analyse décrivent le modèle des accès mémoires réalisés par le programme. Le principe général de ce mécanisme d'exécution est résumé dans la figure 7.7.

Afin d'illustrer ce concept par un exemple concret, nous introduisons la figure 7.8, qui complète le code C montré en figure 7.6. Ce fragment de code consiste en une fonction C acceptant un paramètre, celui-ci étant symbolique puisque sa valeur provient de la lecture d'une variable temporelle. Par souci de simplicité, nous faisons ici l'approximation qu'à une instruction C correspond une instruction assembleur. La figure 7.9 montre ainsi les pseudo traces d'exécution de cet exemple, délimitées par les tics de l'horloge logique cadençant l'agent dans lequel s'exécute le code fonctionnel, ainsi que les relations d'ordre entre les accès mémoires effectués.

7.5.2 Suivi du flot de contrôle

Le flot de contrôle de l'exécution d'une tâche (qui dans notre approche inclue le RTOS) est souvent représenté par un graphe de flot de contrôle, ou CFG (Control Flow Graph). On le note $\mathcal{G} = (\mathcal{V}, \mathcal{E}, i)$ où $\mathcal{V}$ est l'ensemble des nœuds correspondant à des adresses mémoires spécifiques du programme. Celles-ci contiennent des instructions influant sur le flot de contrôle (comme des branchements conditionnels, des appels ou retours de fonctions) ou étant atteignables après l'exécution de ces instructions de flot

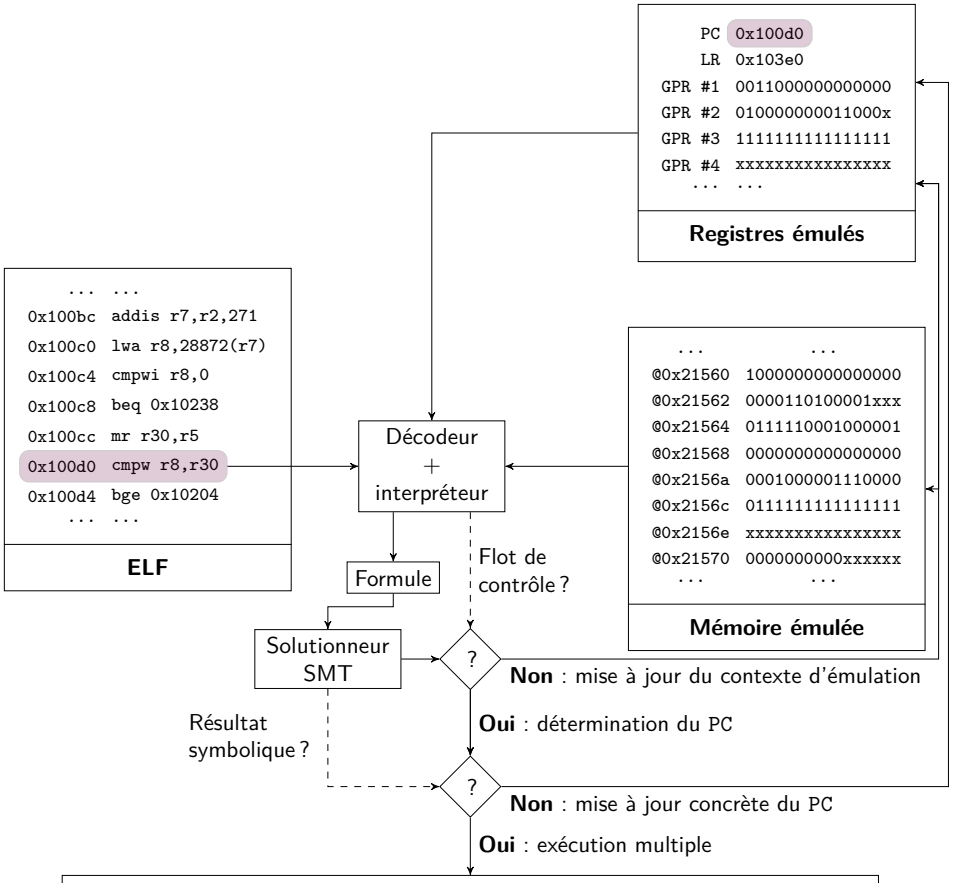


Figure 7.7 – Schéma explicatif de la méthode d'exécution employée.

```
1  // Données globales représentant des zones mémoires mutables
2  static unsigned int count_below_threshold = 0u;
3  static double buffer[2];
4
5  // Code fonctionnel accédant aux données globales définies plus haut,
6  // en fonction d'une variable symbolique (value). On considère que les
7  // variables internes à cette fonction ne sollicitent pas la mémoire
8  // et sont stockées dans des registres (ce que montre l'usage abusif
9  // du mot-clef register ici).
10 void functional_code(register double value) {
11   if (value < THRESHOLD) {
12     count_below_threshold += 1u;
13   } else {
14     for (register int i = 0; i < 2; i++)
15     { buffer[i] = value; }
16   }
17 }
```

Figure 7.8 – Code exécuté au sein du corps fonctionnel de l'agent montré en figure 7.6.

de contrôle (par exemple le contenu des branches issues d'une condition). $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ est l'ensemble des arcs, correspondant au flot de contrôle entre ces adresses. i est le nœud ininitial du CFG, c'est-à-dire le point d'entrée unique de la tâche. Comme expliqué en section 7.3.4, notre seule entrée pour construire ce CFG est constituée d'un programme binaire compilé. Ainsi, l'exécution concolique doit maintenir un CFG partiel (c.-à-d. une sous-approximation) qui décrit le flot de contrôle observé du binaire. On le note $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', i)$ où $\mathcal{V}' \subseteq \mathcal{V}$ et $\mathcal{E}' \subseteq \mathcal{E}$.

Comme le point d'entrée de la tâche i est statiquement connu, il est possible de créer une première version du CFG $\mathcal{G}'$: les instructions sont scannées de manière séquentielle depuis le point d'entrée i et les branches directes sont explorées récursivement jusqu'à ce que toutes les instructions aient été atteintes. Les branches directes sont matérialisées par des instructions de branchement embarquant la prochaine adresse mémoire à exécuter. Ce premier CFG est toutefois incomplet car il ignore les branchements indirects, c'est-à-dire quand la prochaine adresse à exécuter dépend d'instructions contextuelles. Il s'agit typiquement d'instructions lisant ces adresses depuis des registres, dont le contenu est construit et évalué à l'exécution.

Pour les exécutables générés à partir de sources écrites en C, les sources principales de branchements indirects sont les pointeurs de fonction et les appels distants [4]. Même si les pointeurs de fonctions ont tendance à être peu répandus dans les systèmes temps-réel de sûreté — car ils en complexifient l'analyse — les appels distants peuvent foisonner. En effet, ceux-ci sont générés par le compilateur lorsque la fonction à appeler est placée en mémoire « trop loin » du site d'appel, c'est-à-dire que son adresse ne peut être contenue à l'intérieur de l'instruction de branchement. Dans ce cas, un registre intermédiaire est utilisé, ce qui correspond à un branchement indirect. Les appels de fonctions inter-binaires sont résolus grâce à ce même mécanisme, puisque le placement final de chaque exécutable n'est pas nécessairement connu lors de leur compilation. Il est donc indispensable pour notre outil d'analyse de gérer correctement les branchements

4. *Far calls* en anglais.

Pseudo-instructions traitées	(Adresse, taille)	Pseudo trace d'exécution
Démarrage	—	⏲ Temps logique passe à 2
`void psyc_command(void) {`	$(a_1, 4)$	Lecture concrète et exécution sur 4 octets a_1 (instruction)
`for (;;) {`	$(a_2, 4)$	Lecture concrète et exécution sur 4 octets a_2 (instruction)
`value := psyc_read_tv();`	$(a_3, 8)$	Lecture symbolique de 8 octets depuis a_3 (données)
`functional_code(value);`	$(a_4, 4)$	Lecture concrète et exécution sur 4 octets a_4 (instruction)
`if (value < THRESHOLD) {`	$(a_5, 4)$	Lecture concrète et exécution sur 4 octets a_5 (instruction)
		Branchement symbolique $\#1$ – *then*
`count_below_threshold += 1u`	$(a_6, 4)$	Lecture et écriture concrètes de 4 octets à a_6 (données)
`} else {`	—	Branchement symbolique $\#1$ – *else*
`for (register int i = 0;`	$(a_7, 4)$	Lecture concrète et exécution sur 4 octets a_7 (instruction)
`i < 2`	$(a_8, 4)$	Lecture concrète et exécution sur 4 octets a_8 (instruction)
`buffer[i] = value;`	$(a_9, 8)$	Écriture symbolique de 8 octets à a_9 (données)
`i++)`	$(a_{10}, 4)$	Lecture concrète et exécution sur 4 octets a_{10} (instruction)
`i < 2;`	$(a_8, 4)$	Lecture concrète et exécution sur 4 octets a_8 (instruction)
`buffer[i] = value;`	$(a_9 + 8, 8)$	Écriture symbolique de 8 octets à $a_9 + 8$ (données)
`i++)`	$(a_{10}, 4)$	Lecture concrète et exécution sur 4 octets a_{10} (instruction)
`i < 2;`	$(a_8, 4)$	Lecture concrète et exécution sur 4 octets a_8 (instruction)
`}`	—	Branchement symbolique $\#1$ – *end*
`}`	$(a_{11}, 4)$	Lecture concrète et exécution sur 4 octets a_{11} (instruction)
`ast_advance(/*...*/);`	$(a_{12}, 4)$	Lecture concrète et exécution sur 4 octets a_4 (instruction)
		⏲ Temps logique passe à 4
`} for (;;) {`	$(a_2, 4)$	Lecture concrète et exécution sur 4 octets a_2 (instruction)
`value := psyc_read_tv();`	$(a_3, 8)$	Lecture symbolique de 8 octets depuis a_3 (données)
`...`	...	...

Figure 7.9 – Illustration du principe de l'exécution concolique. Une pseudo-exécution du code montré en figures 7.6 et 7.8 conduit à l'obtention de pseudo traces d'exécution. On fait l'hypothèse que les instructions et entiers sont stockés sur 32-bits et que les variables de type `double` le sont sur 64-bits. L'exécution continue jusqu'à un tic d'horloge logique précis, comme discuté en section 7.5.6.

indirects afin de générer un CFG représentatif. En se reposant sur l'hypothèse 5, tous les branchements indirects peuvent être résolus lors de l'exploration du code.

La figure 7.10 représente la construction statique du CFG du code montré en figure 7.6, où il est admis que les appels de fonctions sont traduits par des branchements

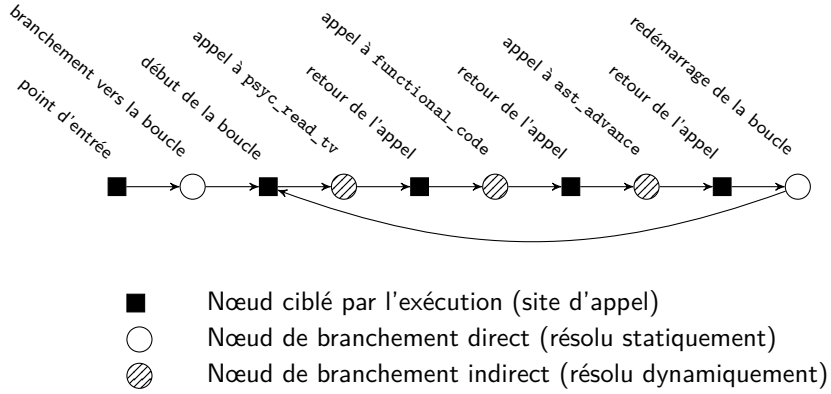


Figure 7.10 – CFG statique construit depuis l'exécutable binaire généré depuis la figure 7.6. Les appels de fonctions étant considérés comme des branchements indirects ici, ce CFG statique ne dispose pas de plus de détails.

indirects. Par souci de simplicité, les routines de démarrage exécutées par le RTOS ne sont pas représentées. Au démarrage, le mode d'émulation est actif et aucune valeur symbolique (comme par exemple le contenu d'une VT) n'a encore été rencontrée. L'appel indirect à `psyc_read_tv()` est résolu grâce au contenu des registres du Vcpu, peuplés et inspectés par les instructions rencontrées. Comme le premier appel à cette fonction atteint une adresse $a \notin \mathcal{V}$, le moteur concolique lance une nouvelle résolution statique du CFG à partir de a. Cela permet d'obtenir un CFG augmenté $\mathcal{G}'$ de telle sorte que $a \in \mathcal{V}'$. Ainsi, le CFG est construit à la volée, au fur et à mesure que l'exploration du programme progresse.

7.5.3 Gestion des contextes symboliques

Le moteur concolique maintient un contexte symbolique au fil de son exécution. Celui-ci est connecté à un solutionneur SMT [BT18]. Étant donné que nous effectuons du DSE, nous pouvons modéliser les cellules mémoires ainsi que les registres par des bit-vecteurs. L'ensemble des registres mobilisés comprend généralement les GPR (General-Purpose Registers), le PC (Program Counter) et le LR (Link Register). Les ISA peuvent requérir davantage de registres, mais nous pouvons les omettre ici. Le PC courant a la propriété d'être toujours une valeur concrète puisqu'il est impératif de connaître l'adresse des instructions à exécuter (hypothèse 5). À l'inverse, la prochaine instruction à exécuter doit être gérée comme un bit-vecteur, car elle peut être symbolique (p. ex. quand une condition symbolique est rencontrée et qu'au moins deux branchements sont possibles).

Nous faisons l'hypothèse que les valeurs concrètes dominent dans l'espace d'adressage du programme (dans lequel le RTOS réside). En effet, celui-ci contient le code exécutable non modifiable (hypothèse 3), les données constantes ainsi que les données modifiables pas encore symboliques. Ainsi, l'espace d'adressage occupé par des valeurs symboliques est représenté par une structure de données creuse. Les valeurs concrètes sont initialisées avec le contenu des segments mémoires du binaire exécutable. Les sections en lecture

seule sont toujours stockées dans des régions mémoires concrètes car ne peuvent devenir symboliques durant l'exécution du programme.

Quand le programme effectue une lecture depuis une adresse mémoire concrète, les opérations suivantes sont effectuées par le moteur concolique.

— Le segment mémoire contenant cette adresse est recherché. L'échec de cette recherche révèle un accès à une zone mémoire associée à des entrées-sorties (auquel cas une donnée symbolique est retournée) ou indique un accès mémoire illégal.

— Si la section est inscriptible (c'est-à-dire qu'elle peut contenir des données symboliques), une valeur symbolique est retournée si elle est présente dans la représentation creuse de l'espace d'adressage symbolique. Dans le cas contraire, une lecture concrète est effectuée.

— Si la section est en lecture seule, une lecture concrète est systématiquement effectuée car cette section ne peut contenir de données symboliques (hypothèse 3).

De manière analogue, l'écriture de l'espace d'adressage à des adresses concrètes déclenche les opérations suivantes.

— Le segment mémoire contenant cette adresse est recherché. L'échec de cette recherche révèle un accès à une zone mémoire associée à des entrées-sorties (auquel cas une écriture fictive est effectuée) ou indique un accès mémoire illégal.

— Si la section trouvée n'est pas inscriptible, une erreur est levée car cela indique une violation des hypothèses 1 à 3.

— Si les adresses devant être écrites sont présentes dans la représentation creuse de l'espace d'adressage, alors elles sont modifiées en conséquence. Une entrée dédiée est créée au moment de la première écriture symbolique. Si la donnée à écrire est concrète et qu'aucune entrée n'existe encore, alors une simple écriture concrète est effectuée.

Gérer les accès mémoire à partir d'adresses symboliques (et non plus concrètes) présente des différences. Pour effectuer des lectures et des écritures, une opération de « ciblage » [5] est réalisée, comme détaillé en section 7.5.4. Au moment des lectures, le moteur concolique retourne une valeur symbolique contenant autant de bits symboliques que nécessaire. Les opérations d'écritures demandent une gestion particulière car peuvent demander d'écrire des plages d'adresses potentiellement importantes, modifiant en profondeur le contenu de l'espace d'adressage. Le ciblage joue ainsi un rôle clef : si moins de λ bits symboliques interviennent dans la résolution d'une adresse (où $\lambda \in \mathbb{N}$ est une valeur de seuil arbitraire), des opérations régulières d'écriture sont exécutées. Dans le cas contraire, ces écritures sont jugées trop coûteuses et l'exécution concolique ne peut se poursuivre : l'analyse échoue.

7.5.4 Résolution des adresses symboliques

L'objectif principal de cette exécution concolique réside dans le traçage des accès mémoires effectués lors de l'exécution d'une tâche. Ainsi, les adresses inconnues (symboliques) doivent être résolues à un ensemble d'adresses plausibles. Les ignorer dégraderait

5. *Narrowing* en anglais.

significativement la pertinence de notre analyse, étant donné que les écritures influent sur le flot de contrôle. D'autre part, des écritures symboliques excessives peuvent mettre en péril les opérations de concrétisation.

Par exemple, il n'est pas inhabituel que des tampons mémoires soient accédés en fonction d'une valeur d'index dépendant du contenu d'un registre, lequel peut faire intervenir une variable symbolique dans son contenu. Nous nous appuyons sur l'hypothèse 2 pour restreindre l'occurrence de ce genre de scénarios. C'est en effet une pratique standard dans les systèmes temps-réel de sûreté que de contrôler que les accès mémoires sont toujours restreints à des zones mémoires valides (par exemple dans la règle 18.1 du MISRA-C [MIS19]). Le code montré précédemment en figure 7.2 pour illustrer l'hypothèse 2 représente bien ce problème. En effet, dans cet exemple, l'index utilisé pour accéder à la mémoire est borné par une opération modulo qui restreint la plage des adresses mémoires accessibles. L'absence de cette contrainte aurait permis une sélection d'une partie bien trop importante de l'espace d'adressage.

Ainsi, en pratique, les adresses symboliques devraient être exprimées comme des plages d'adresses concrètes, pouvant être significativement réduites grâce à certaines contraines embarquées dans le code. Cette opération de ciblage peut être implémentée dans des solutionneurs SMT en forçant la résolution des opérations appliquées aux bit-vecteurs. Cela permet d'identifier clairement quels sont les bits concrets et symboliques d'une variable, ceux-ci recevant des valeurs explicites pouvant être 0, 1 ou « x » (symbolique).

7.5.5 Exploration des branches symboliques

Comme mentionné en section 7.3.3, l'exécution symbolique est sensible au problème de l'explosion des états, qui empire avec le nombre de branches recontrées. Si l'exécution concolique permet d'éliminer les branchements superflus grâce à la technique de concrétisation, l'explosion exponentielle des états est ramenée à chaque branchement symbolique. C'est-à-dire lorsque la concrétisation ne permet pas d'identifier une branche à suivre en particulier. Comme notre objectif est de récupérer tous les accès mémoires effectués par le programme, nous devons explorer les branchements symboliques.

Nous réussissons à contenir le problème de l'explosion des états en nous appuyant sur le CFG expliqué en section 7.5.2. À chaque fois qu'une instruction est consommée par le moteur d'exécution concolique, celui-ci identifie la position du PC sur le CFG. Dans ces travaux, nous faisons l'hypothèse que le code s'articule autour de constructions telles que des fonctions C. Formellement, une branche conditionnelle est identifiée sur le CFG comme un nœud avec plus d'un successeur, comme illustré en figure 7.11 (nœud b). Quand le moteur concolique rencontre une branche dépendant d'une condition au nœud n, plusieurs choix sont possibles.

— Si la condition est concrète, alors une seule branche est atteignable car la condition peut être évaluée avec exactitude.

— Si la condition est symbolique, alors les deux branches doivent être explorées car il est impossible de déterminer avec certitude une seule branche candidate.

Ce dernier cas est bien sûr le plus intéressant des deux et celui posant le plus de problèmes. Comme toutes les instructions sont exécutées dans le contexte d'une fonction C (ou peut-être une structure similaire), le CFG peut temporairement être restreint aux fonctions

atteignables depuis celle-ci. Ceci inclut les appels systèmes et les fonctions récursives. Tout d'abord, cette portion du CFG est explorée afin de trouver tous les chemins commençant aux successeurs du nœud n. La construction de ces chemins s'interrompt lorsqu'ils contiennent plus d'une fois le même nœud. Dans l'exemple de la figure 7.11, deux chemins peuvent être construits depuis le nœud b : (c, d, e, h, i) et (f, g, h, i). Si tous les chemins partagent au moins un nœud n' en commun, alors celui-ci est désigné comme étant un nœud de jonction (dans notre exemple : $n' = h$[6]). La structure du code assembleur des fonctions rend cette stratégie viable car elles comportent un point de sortie unique et donc commun à toutes les branches à l'intérieur d'une fonction.

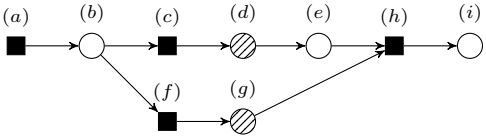


Figure 7.11 – Hypothétique CFG généré depuis une fonction C simple contenant une construction de type « if-then-else » dans laquelle chaque branche appelle une seule fonction (par branchement indirect). Les nœuds ont la même signification qu'en figure 7.10.

Si n' existe, le moteur concolique créé un nouveau contexte symbolique, qui est un clone superficiel du Vcpu effectuant l'interprétation des instructions. Ce clone inclut une représentation de la mémoire du système ainsi qu'une copie du PC. Ce nouveau contexte prend la main sur son parent et reprend l'exécution concolique en suivant l'une des branches en utilisant l'adresse du premier nœud d'un des chemins précédemment construits (dans notre exemple, c ou f). Son exécution s'interrompt lorsque l'adresse du nœud n' est rencontrée. Le contexte parent reprend alors son exécution jusqu'à atteindre ce même nœud, mais en suivant l'autre chemin. Ces contextes symboliques sont alors fusionnés, ce qui correspond aux valeurs des registres et aux cellules mémoires. Les règles de fusion sont les suivantes, et sont très similaires à celles de Lundqvist et Stenström [LS99b] :

— si deux bit-vecteurs sont identiques, n'importe lequel des deux peut être utilisé ;

— dans le cas contraire les bits différents entre eux deviennent symboliques et les bits identiques sont conservés.

Après le processus de fusion, le moteur concolique reprend son exécution et le contexte symbolique résultant contient la superposition des états des exécutions symboliques précédentes. Ce mécanisme est bien sûr appliqué de manière récursive afin de gérer les branchements imbriqués, pouvant même survenir au sein de fonctions différentes.

Les boucles, qui n'ont pas encore été abordées, présentent quelques subtilités additionnelles. Comme illustré en figure 7.12, elles complexifient la construction des chemins et la découverte de points de fusion uniques. Dans cet exemple, trois chemins sont possibles à partir du branchement symbolique du nœud b : (c, d, e, h, i), (f, g, j, k, h, i) et (f, g, j, k). Ce dernier chemin est produit par le corps de la boucle. À cause de

6. Notons que $n' = i$ est également possible. On priviliégiera toutefois une intersection minimisant le nombre de nœuds.

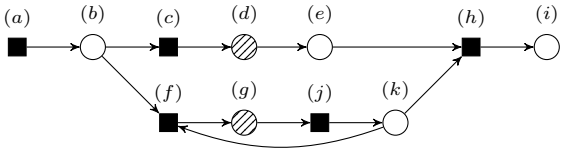


Figure 7.12 – Modification du CFG de la figure 7.11 pour que l'une des branches contienne une boucle. Les nœuds ont la même signification qu'en figure 7.10.

celui-ci, les trois chemins ne partagent pas de nœud de fusion unique. Toutefois, quand l'exécution atteint le nœud b — avant la boucle —, le chemin (f, g, j, k) peut être identifié comme étant une boucle et peut être temporairement ignoré. Deux chemins restent à l'étude et comme ils partagent au moins un nœud en commun, on trouve un nœud de fusion $n' = h$. Cette approche peut sembler surprenante, mais se repose sur l'hypothèse 4 qui sous-tend que les boucles sont concrètes et peuvent donc être entièrement résolue par l'émulation. Quand l'exécution concolique atteint le nœud k (qui contrôle la boucle) il est possible de déterminer si la condition assujettie au branchement génère une boucle. Dans ce cas, si la condition est concrète, l'exécution poursuit grâce à l'émulation. Si la condition est symbolique, l'hypothèse 4 est violée : la boucle n'est pas bornée et l'exécution concolique ne peut poursuivre, car n'est pas en mesure de déterminer un critère d'arrêt à cette boucle avec certitude.

Cette combinaison de boucles concrètes et de branchements symboliques permet à l'exécution concolique d'explorer toutes les branches atteignables du programme tout en réduisant grandement le problème de l'explosion des états. Comme cette approche a d'abord recours à l'émulation, on peut lui reprocher de manquer d'« élégance » sur sa gestion des boucles. En effet, d'autres approches plus formelles — comme l'interprétation abstraite — peuvent analyser des boucles de $m \in \mathbb{N}$ itérations sans effectuer m opérations. Notre approche nous force à toutes les effectuer. Toutefois, comme nous ciblons une catégorie très particulière de logiciels, il nous semble peu probable que m soit « grand » dans la pratique. De plus, cette technique nous permet de capturer les accès mémoires différentiés pour chaque itération, permettant une collecte de données plus fine.

7.5.6 Gestion du temps logique

Dans la section 7.4, le modèle de notre système requiert que les tâches soient créées une seule fois et que leurs durées de vies soient liées à celle du RTOS qui les exécute. Cela implique que les tâches aient une durée de vie virtuellement infinie, puisqu'elles ne terminent jamais (sauf cas d'erreur). Ce sont les horloges logiques qui rythment leur exécution. Des points de synchronisation temporels (c.-à-d. les instructions `advance`) font progresser le temps logique. Cela est capturé par le moteur d'exécution concolique, qui dispose d'une intime connaissance du RTOS sous-jacent. Ainsi, le temps logique est automatiquement mis à jour durant la progression de l'analyse hors-ligne du programme.

Comme notre méthode n'intègre pas les aspects sémantiques du programme analysé, le moteur d'exécution concolique ne peut déterminer de moment précis du programme pour lequel tous les accès mémoires auront été couverts. Afin de lever cette limitation,

nous nous reposons sur le système des horloges logiques. Le concepteur de l'application désirant réaliser l'exécution concolique du programme doit indiquer un tic d'horloge logique Λ marquant la fin de l'exploration. L'analyse s'arrête lorsque la date logique courante devient supérieure ou égale à Λ. Ce tic marque une « hyperpériode » de l'application : après l'occurrence de Λ, le programme ne peut exhiber de motifs d'accès mémoires déjà réalisés. Cette approche est assez similaire au *bounded model checking* [Bie+03].

Déterminer Λ avec exactitude est sans doute une tâche ardue qui est confrontée à des obstacles pratiques importants. Aussi est-il peu probable que ces « hyperpériodes » puissent être déterminées pour toutes les applications. Toutefois, à la différence des problèmes d'estimation du WCET, qui requièrent de trouver une borne supérieure stricte (*sound*) — dans notre cas à Λ — les résultats de notre analyse n'ont pas de considérations de sûreté. En effet, les stratégies de mitigation déployées suite à notre collecte d'information n'ont pas vocation à apporter des garanties de sûreté, mais uniquement à améliorer un existant (*best effort*). Aussi, leurs impacts sur les temps d'exécution seront capturés par les outils d'analyse de WCET qui font ainsi office de sentinelles. Même si le choix de Λ est quelque peu arbitraire (*unsound*), tant que les mitigations découlant des résultats de l'analyse sont pertinentes vis-à-vis des WCET, cette technique reste intéressante.

7.5.7 Prise en compte de la préemption et des interruptions

La méthode proposée dans ce chapitre nous permet de construire, grâce à des traces d'exécution, un modèle des accès mémoires effectués par une tâche ainsi que le RTOS sous-jacent. Toutefois, il ne peut renseigner les accès mémoires causés par les interruptions et la préemption, qui peuvent survenir durant l'exécution sur cible matérielle. Ces événements permettent aux cœurs processeur d'exécuter du code noyau virtuellement entre chaque instruction du domaine utilisateur de manière non déterministe et non prédictible. On pourrait objecter que comme notre modèle des accès mémoires ne capture pas ces informations, les stratégies de mitigations qui en découlent pourraient être invalides.

Comme cela est souvent le cas pour les systèmes implémentant le paradigme LET, les interruptions sont réduites à un strict minimum. Typiquement, au moins un minuteur contrôle la consommation de temps physique par les tâches (et le RTOS). L'expiration de ces minuteurs peut être utilisée pour contrôler les échéances des tâches ou pour provoquer l'élection d'une nouvelle tâche (conduisant à de la préemption). Ainsi, les instructions et les données pouvant être accédées par le biais des interruptions ou de la préemption doivent être gérées avec minutie pour qu'elles n'interfèrent pas ou peu. Par exemple, celles-ci peuvent être placées dans des régions mémoires non-cachables ou dans des mémoires locales aux cœurs.

Ainsi, une gestion particulière de ce type de code doit permettre que des accès mémoires surnuméraires puissent s'intercaler au milieu de ceux listés dans notre modèle des accès mémoires, sans que leur occurrence ne perturbe les mitigations déployées.

7.6 Résultats expérimentaux

Dans cette section, nous détaillons notre prototype d'un moteur d'exécution concolique, qui implémente les concepts avancés en section 7.5. Nous le nommons `kc3e` par souci de concision.

7.6.1 Choix d'implémentation

`kc3e` est un exécutable compilé avec le langage de programmation rust [7] (version 1.45.2). La cible principale du moteur est la plateforme QorIQ T1042 développée par NXP [Fre15]. Elle intègre quatre cœurs e5500 de l'architecture Power [NXP13]. Ainsi, `kc3e` supporte le décodage d'exécutables au format ELF [TIS95] ainsi qu'un sous-ensemble de l'ISA Power 2.06. Nous avons préféré cette cible au MPC5777M pourtant principalement utilisé dans ces travaux, car le MPC5777M implémente un ISA bien plus vaste. Par souci d'économie de développement, nous avons préféré un ISA plus réduit. L'exploration de graphes, le décodage des instructions et l'émulation de manière plus générale ont été implémentés de manière *ad hoc*, sans nous reposer sur des composants tiers. Comme `kc3e` est un prototype, il n'a pas subi de passe d'optimisation : il ne bénéficie ni du multithreads, ni des techniques de caches souvent utilisées par les émulateurs. À son stade actuel, il est donc peu compétitif en termes de performance.

L'exécution concolique repose sur Boolector [BB09] (version 3.2.1) comme solutionneur SMT. Ce choix est motivé par son efficacité à manipuler les bit-vecteurs [NPB19], son intégration au langage rust ainsi que sa licence (MIT). À la différence de certains outils concurrents (comme Z3 [DB08]), Boolector ne fournit pas de support pour les théories des nombres flottants. En pratique, ce type d'opérations intervient rarement dans la détermination des accès mémoires, aussi son intégration n'a pas été jugée nécessaire. De plus, cela permet de stresser le concept du moteur d'exécution concolique, puisque toutes les manipulations de nombres flottants contiennent des valeurs symboliques : les opérations flottantes sur f bits font systématiquement intervenir des bit-vecteurs de f bits symboliques.

Comme notre exécution concolique est particulièrement intégrée à Asterios, nous nous reposons sur le RTOS Asterios pour gérer les interruptions permettant le contrôle de l'écoulement du temps physique ainsi que les mécanismes de préemption. `kc3e` effectue une partie seulement des routines d'initialisation du RTOS : une partie de la configuration du matériel n'est pas nécessaire en mode émulation. Trois binaires sont chargés par le moteur concolique ; ce sont les mêmes que ceux représentés en figure 3.1 : le noyau, un exécutable de partition contenant les agents à analyser et la « runtime » qui contient les données pré-calculées nécessaires à la bonne exécution des agents par le noyau (section 3.1.2). Un fichier de configuration fournit à `kc3e` des informations supplémentaires quant à la configuration de l'espace d'adressage (où sont les mémoires matérielles, les entrées-sorties, *etc*). Des évolutions futures permettront d'y intégrer des mécanismes d'annotations pour affiner l'exécution concolique. `kc3e` détecte automatiquement, en fonction du contenu du binaire de « runtime » les tampons de communications utilisés par les VT. Il les marque comme étant « purement symboliques », c'est-à-dire que lire depuis ces zones mémoires retourne toujours une valeur symbolique.

7. https://www.rust-lang.org/

Au cours de son exécution, des traces d'exécution sont produites, où les accès mémoires sont répertoriés dans leur ordre d'occurrence, accompagnés du tic d'horloge logique de leur survenue. Ce sont ces traces qui pourront être exploitées par des outils tiers afin de mettre en œuvre des stratégies de mitigations.

7.6.2 Cas d'étude ROSACE

Les applications temps-réelles de sûreté sont en grande partie développées par des industriels, font la plupart du temps partie du secret professionnel et sont soumises à des politiques de propriété intellectuelle. Il est ainsi souvent ardu d'utiliser des applications représentatives de tels systèmes ou de communiquer dessus quand celles-ci sont accessibles sous conditions restreintes — sous NDA (*non-disclosure agreement*).

Le cas d'étude ROSACE [Pag+14] présente le grand intérêt d'être une étude avionique open-source, axée sur la modélisation de systèmes de contrôle. Son code source est public[8]. Il décrit un contrôleur utilisant des périodicités multiples, conçu dans un environnement Simulink. L'application est décomposée en une quinzaine de blocs fonctionnels qui produisent et consomment des données à différentes fréquences. Leur amalgame en un tout cohérent est réalisé grâce à une description écrite en langage Prelude [Cor+11].

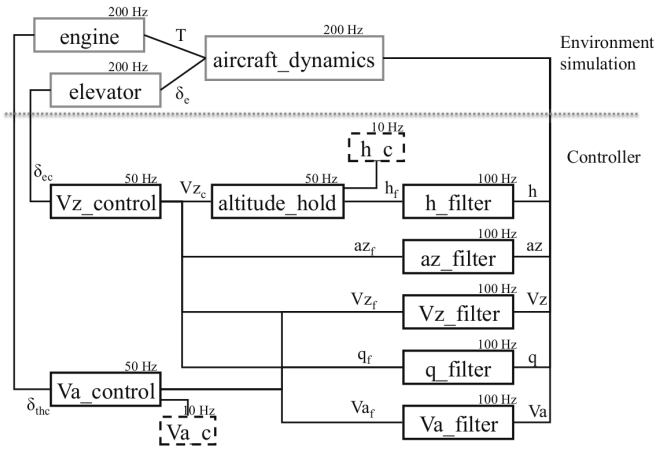


Figure 7.13 – Interaction entre les différents composants de l'application ROSACE. Figure extraite de Pagetti et al. [Pag+14, p. 4]. Chaque bloc est nommé et est accompagné de sa fréquence d'exécution. Les relations entre blocs indiquent les variables échangées, détaillées en figure 7.14.

Pour pouvoir être utilisée par notre moteur concolique, l'application ROSACE doit d'abord être convertie en agents PsyC. Pour cela, nous associons un agent à chaque bloc fonctionnel. Chaque agent est composé d'une EA principale qui exécute le code fonctionnel du bloc. Nous ne rappellerons pas ici le détail de ROSACE, mais nous rappelons le diagramme d'interaction entre les différents blocs fonctionnels en

8. Un lien vers un dépôt « subversion » est fourni dans le papier d'origine.

Outputs	V_z	vertical speed
	V_a	true airspeed
	h	altitude
	a_z	vertical acceleration
	q	pitch rate
Filtered outputs	V_{z_f}	vertical speed
	V_{a_f}	true airspeed
	h_f	altitude
	a_{z_f}	vertical acceleration
	q_f	pitch rate
Reference inputs	h_c	altitude command
	V_{a_c}	airspeed command
Commanded inputs	V_{z_c}	vertical speed command
	δ_{e_c}	elevator deflection command
	δ_{th_c}	throttle command
Aircraft inputs	δ_{e_c}	elevator deflection
	T	engine thrust

Figure 7.14 – Détail des données échangées par les blocs consistuant ROSACE. Figure extraite de PAGETTI et al. [Pag+14, p. 4].

figure 7.13. À titre d'exemple, le code de la figure 7.15 montre la transformation en agent PsyC du bloc fonctionnel *altitude_hold*. La fonction C `altitude_hold_50()` du code fonctionnel de ROSACE est ainsi appelée avec une fréquence de 50 Hz. Notons que le PsyC travaille en unités de temps et non en fréquence. Il est alors nécessaire de concevoir des horloges qui traduisent une fréquence en période. Par souci de clarté avec l'application d'origine ces horloges sont nommées `clk_50hz`, `clk_100hz`, *etc*, mais il s'agit bien d'horloges résultant d'une conversion de hertz en unité de temps.

Le code fonctionnel de ROSACE repose particulièrement sur des fonctions mathématiques, comme `sin()`, `cos()`, `sqrt()`, `atan()` et `pow()`, qui doivent être fournies par des bibliothèques tierces. Ces bibliothèques contiennent souvent un code complexe, motivé par des contraintes de performance ; toutes ne sont pas adaptées aux systèmes temps-réels de sûreté. Toutefois, des implémentations propres à ce genre de systèmes font généralement partie de la propriété intellectuelle des industriels et nous n'avons pas trouvé de bibliothèque mathématique adaptée et open-source. Aussi, nous avons opté pour l'utilisation d'une bibliothèque grand public, donc susceptible d'être incompatible avec les limitations que nous nous sommes fixées en section 7.4. Nous traitons ces impairs au cas par cas. Musl[9] (version 1.2.1) est le composant logiciel retenu. Open-source et sous licence MIT, son artefact `libm.a` s'intègre à notre application ROSACE réécrite en PsyC. Certaines des fonctions implémentées par musl exhibent trop de violations à nos hypothèses, en particulier `sin()` et `cos()`. Aussi, nous leur substituons des implémentations utilisant la technique CORDIC [Vol59]. Notons que le CFG produit par `kc3e` est trop large pour être représenté ici et comporte près d'un millier de nœuds.

Comme nos travaux proposent une technique d'analyse des accès mémoires, nous

9. `https://musl.libc.org/`

```
// Déclarations des variables temporelles par lequelles transitent les
// données produites et consommées par ce bloc fonctionnel.
temporal double TV_h_f[2] with clk_100hz;
temporal double TV_h_c with clk_10hz;
temporal double TV_V_z_c with clk_50hz;

agent altitude_hold
( uses realtime, starttime 1 with clk_50hz )
{
  // Variables temporelles consommées
  consult 1 $ TV_h_f; // h_f
  consult 1 $ TV_h_c; // h_c
  // Varibles temporelles produites
  display TV_V_z_c : Vz_control; // V_zc

  // Global variable, private to the agent
  global { int h_f_rcell = 0; }

  body start {
    // Appel au code fonctionnel de ROSACE
    const double Vz_c = altitude_hold_50(
        $[0]TV_h_f[h_f_rcell],
        $[0]TV_h_c);
    h_f_rcell = (h_f_rcell + 1) % 2;
    // Mise à jour de la VT contenant V_zc
    TV_V_z_c = Vz_c;
    // Fin d'une exécution du bloc fonctionnel
    advance 1 with clk_50hz;
  }
}
```

Figure 7.15 – Conversion du bloc fonctionnel *altitude_hold* en agent PsyC. Rappelons que la figure 7.14 décrit les variables échangées et la figure 7.13 leurs contraintes temporelles.

présentons ici quelques représentations spatiales et temporelles de leurs occurrences au sein du cas d'étude ROSACE. Étant donné la profusion des informations collectées, il est peu aisé de fournir une représentation statique qui soit exhaustive et compréhensible. Aussi, nous utilisons des représentations parcellaires pour montrer nos principaux résultats. Les figures 7.16 à 7.18 montrent des chronologies des accès mémoires entre des tics d'horloge logique, tandis que les figures 7.19 et 7.20 présentent des vues d'ensemble de l'espace d'adressage utilisé. Les résultats ont été obtenus en exécutant chaque agent PsyC constituant ROSACE jusqu'à ce que le tic logique 60 ait été atteint. Les données ont été collectées par une exécution de `kc3e` sur un cœur i9-9880 cadencé à 2,30 GHz, totalisant cinq minutes d'exécution.

Une chronologie des accès mémoires effectués par l'agent PsyC de ROSACE contrôlant la dynamique de l'avion (*aircraft dynamics*) entre les tics logiques 10 et 30 est affichée en figure 7.16. Les accès mémoires sont affichés le long de l'axe des abscisses tandis que les adresses accédées s'étendent le long de l'axe des ordonnées. Les accès aux instructions (en vert) et aux données lues (en bleu) et modifiées (en rouge) sont uniformément répartis entre deux tics logiques. Cette répartition ne tient pas compte

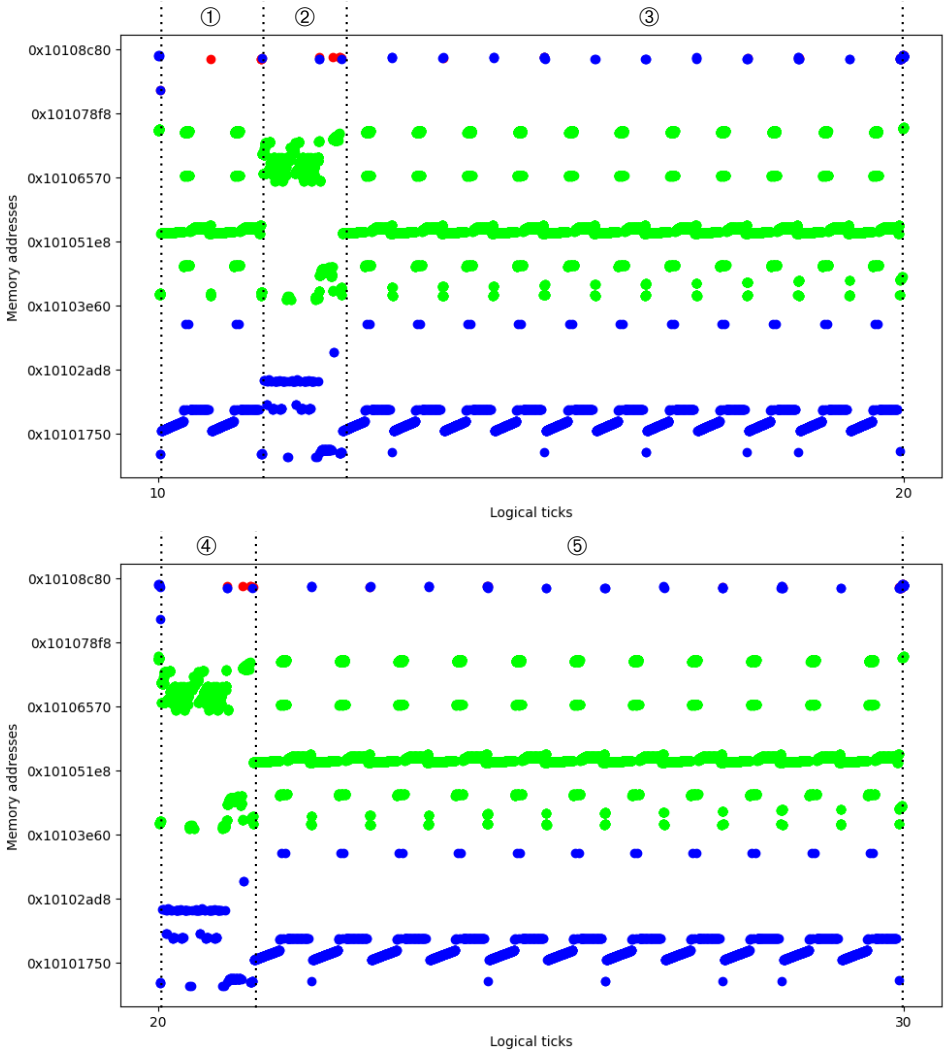


Figure 7.16 – Chronologie d'un sous-ensemble des accès mémoires effectués entre les tics logiques 10 et 30 d'un agent de ROSACE. Les accès en vert montrent les accès aux instructions, ceux en bleu aux données lues et en rouge aux données écrites.

du temps physique : étant donné qu'il s'agit d'une exécution hors ligne, seul le temps logique peut être représenté. Chacune des sous-figures représente l'exécution d'une EA et seulement un sous-ensemble des accès mémoires effectués. En effet, l'espace d'adressage étant important, une vue globale de tous les accès mémoires effectués sur l'ensemble de l'espace d'adressage deviendrait illisible. La première exécution de l'EA de cet agent est légèrement différente des suivantes, car elle comporte un code d'initialisation appelé une seule fois. Celui-ci consiste principalement en l'appel aux fonctions `cos()` et `sin()`,

que l'on peut observer en ①. Le restant de la première exécution est la même que pour toutes les autres. Des appels à `pow()`, `atan()` et `sqrt()` sont visibles en ② et ④. Ils sont suivis par des appels à `sin()` et `cos()`, comme en ③ et ⑤.

La chronologie en figure 7.17 — qui reprend la même légende que la figure 7.16 — montre le problème de clarté d'affichage quand l'intégralité de l'espace d'adressage est montré (sous-figure du haut). On peut toutefois y identifier deux zones distinctes. Ainsi, ⑥ rassemble le code et les données de l'agent, montrées en figure 7.16, en incluant la pile, qui n'était pas représentée précédemment. ⑦ montre la région mémoire occupée par le RTOS. Cette zone est détaillée dans la sous-figure du bas. On y identifie les accès à sa pile en ⑧ et le reste des instructions et données globales dans la partie basse de l'espace d'adressage en ⑨.

Ces résultats nous permettent de saisir la nature des accès mémoire effectués. Cette analyse qualitative peut être affinée par la figure 7.18, qui trace uniquement les accès en écriture effectués par cet agent PsyC, avec différentes échelles. La première sous-figure montre un sous-ensemble de l'espace d'adressage dans lequel résident les données de l'agent et dans la seconde la totalité de l'espace d'adressage. ⑩ montre en particulier les écritures de la pile par l'agent, qui est dans notre cas allouée de `0x10043000` à `0x10044000`.

Les représentations temporelles montrées plus haut (figures 7.16 à 7.18) permettent de visualiser l'évolution des accès mémoires entre deux tics logiques. Il peut être intéressant de pouvoir observer la répartition spatiale de ces accès et ainsi évaluer l'utilisation de l'espace d'adressage. Les figures 7.19 et 7.20 en offrent une vue d'ensemble. Ces figures sont obtenues après analyse de l'exécution de tous les agents PsyC constituant ROSACE, entre les tics logiques 10 et 60. Plus la taille de l'espace d'adressage croît, moins ce genre de représentation devient lisible. La figure 7.19 exclut le code d'initialisation du RTOS, qui est laissé dans la figure 7.20. Les régions mémoires montrées dans ces figures présentent une occupation de faible densité. Cela est dû à la politique de placement mémoire par défaut appliquée dans notre exemple, qui privilégie une capacité de protection mémoire fine à une utilisation compacte. ❶ montre les instructions et données accédées par le RTOS sans sa pile. Cette dernière est montrée en ❷. Le binaire de « runtime » est montré dans son intégralité en ❸. On aperçoit en bas de cette sous-figure les piles utilisées par les agents. Il s'agit ici de vingt-six piles descendantes, chacune ayant une capacité maximale de 4096 octets, mais seulement une fraction de cette capacité est utilisée. Chaque agent dispose de deux piles, car Asterios implémente deux contextes par agent. On remarque que quatre piles ne sont pas utilisées : l'exécution concolique entre les tics 10 et 60 ne capture pas l'exécution des agents contrôlant les blocs fonctionnels « VA_{C_0} » et « H_{C_0} ». On constate que toutes les piles sont entièrement écrites en figure 7.20 : le RTOS peint les piles avec des valeurs-clés à l'initialisation. Enfin, ❹ montre les accès effectués par les agents (en dehors de leurs piles, qui sont contenues dans ❸). On remarque également que du code supplémentaire est exécuté en ❶ : celui-ci correspond au code d'initialisation du RTOS.

Toutes ces données collectées visent à développer des stratégies de placement mémoire permettant une meilleure utilisation des mémoires et des caches. L'objectif principal étant d'améliorer certaines des propriétés des temps d'exécution — notamment du WCET — par rapport à un placement par défaut. Nous laissons leur développement pour de futures recherches, mais quelques pistes sont proposées en section 7.2.

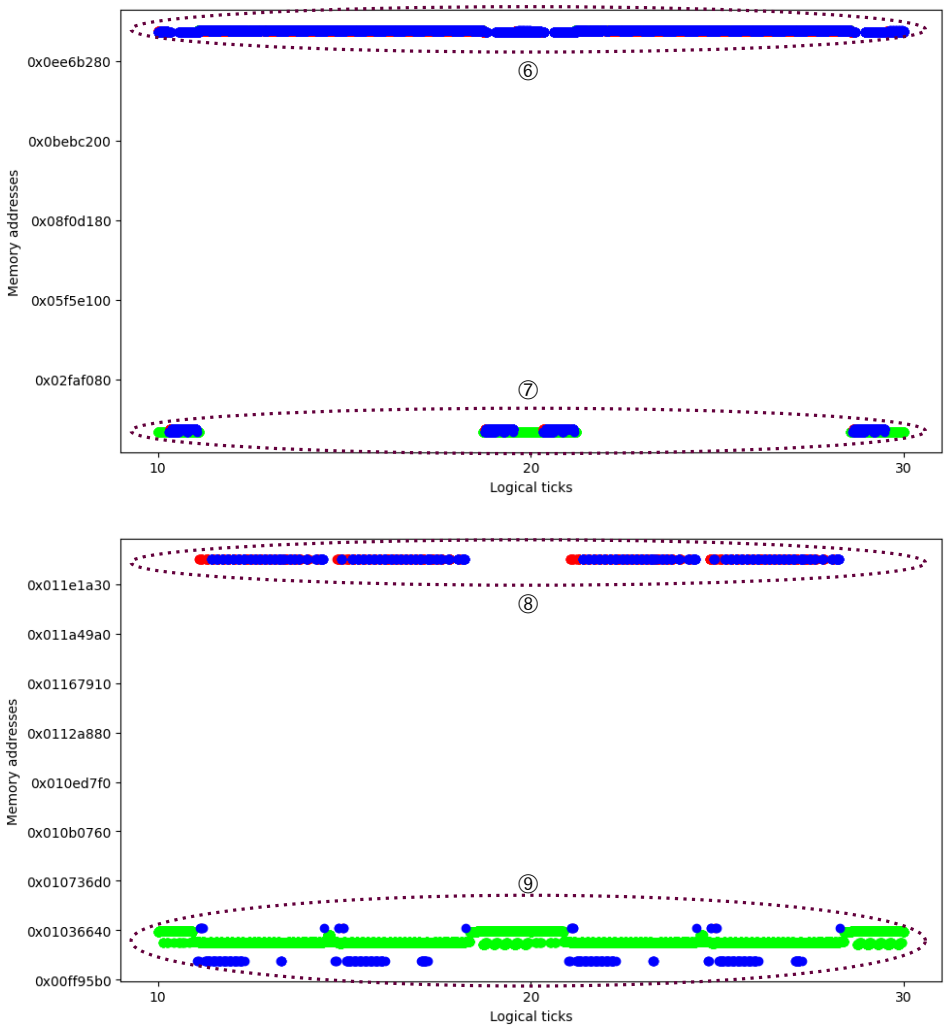


Figure 7.17 – Chronologie des accès mémoires effectués en figure 7.16, montrant la totalité des accès sur l'ensemble de l'espace d'adressage (figure du haut) et dans l'espace dans lequel réside le RTOS — c'est-à-dire ⑦ — (figure du bas).

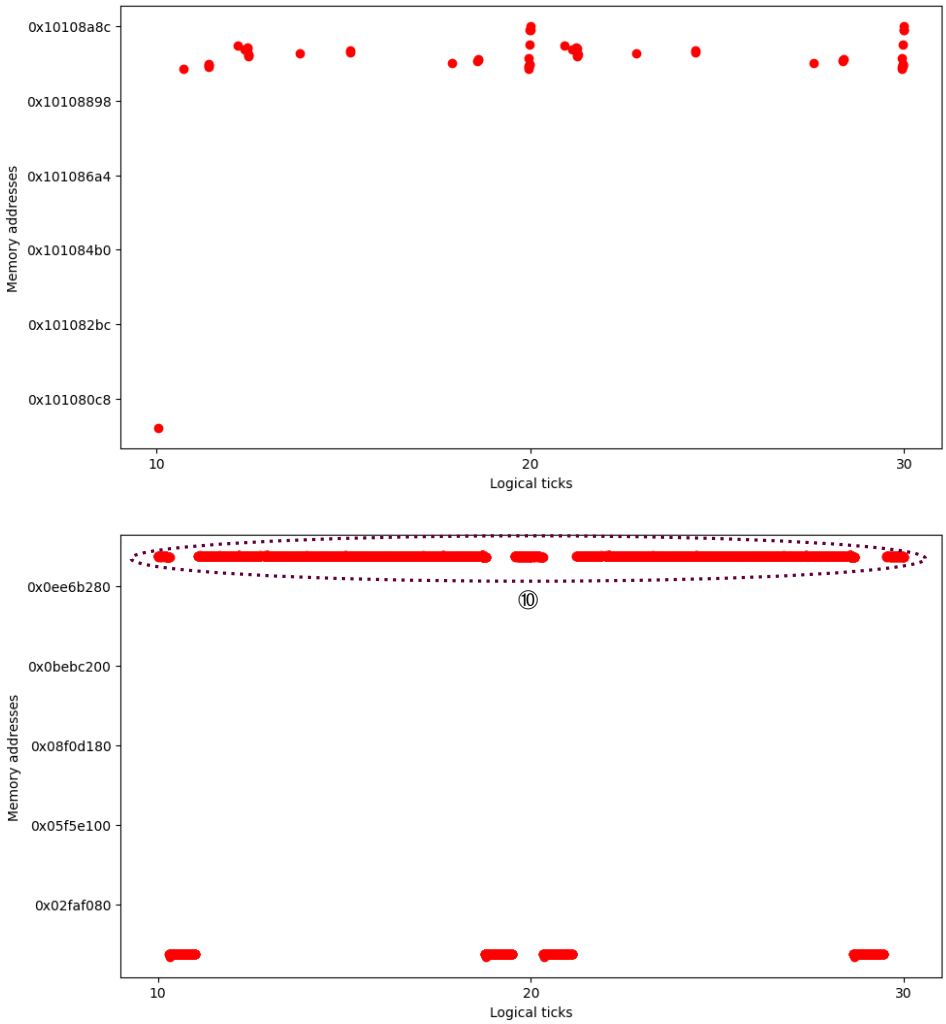


Figure 7.18 – Chronologie des accès mémoires effectués en figure 7.16, montrant les écritures effectuées par l'agent (figure du haut) et la totalité des écritures sur l'ensemble de l'espace d'adressage (figure du bas).

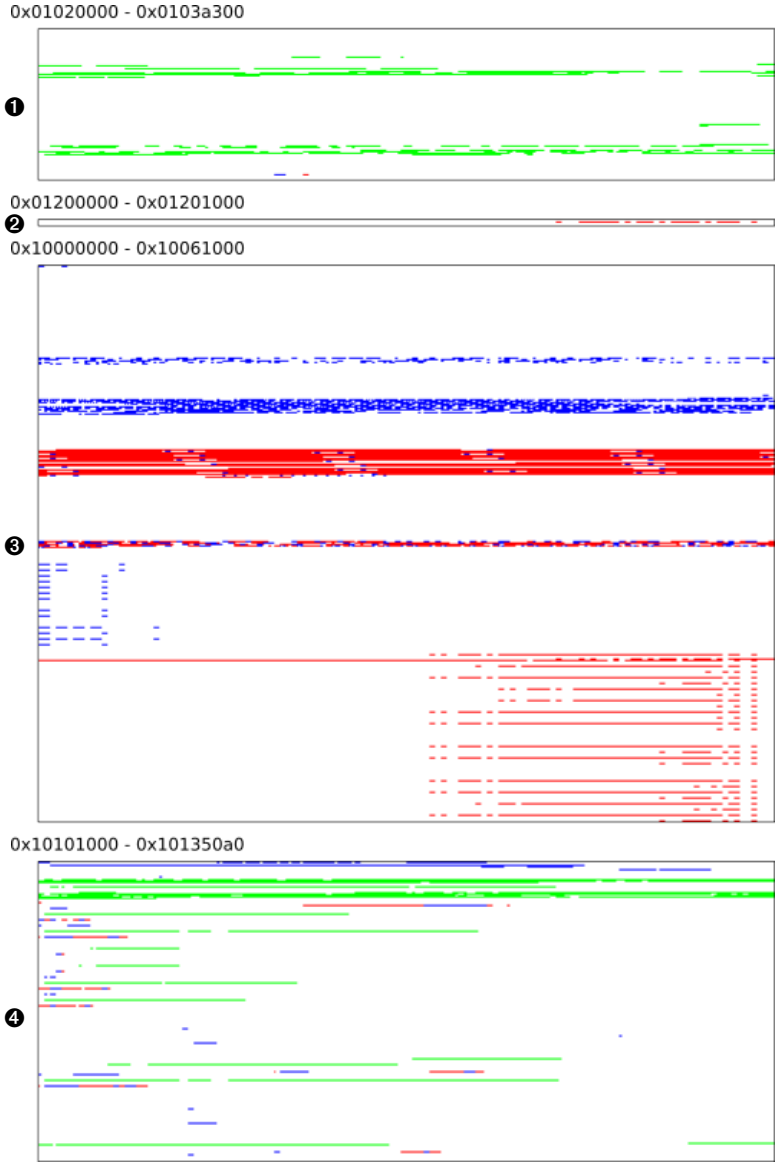


Figure 7.19 – Carte des accès mémoires réalisés par l'application ROSACE, sans tenir compte des accès effectués durant l'initialisation. Chaque ligne représente 1024 octets. La légende est la même qu'en figure 7.16.

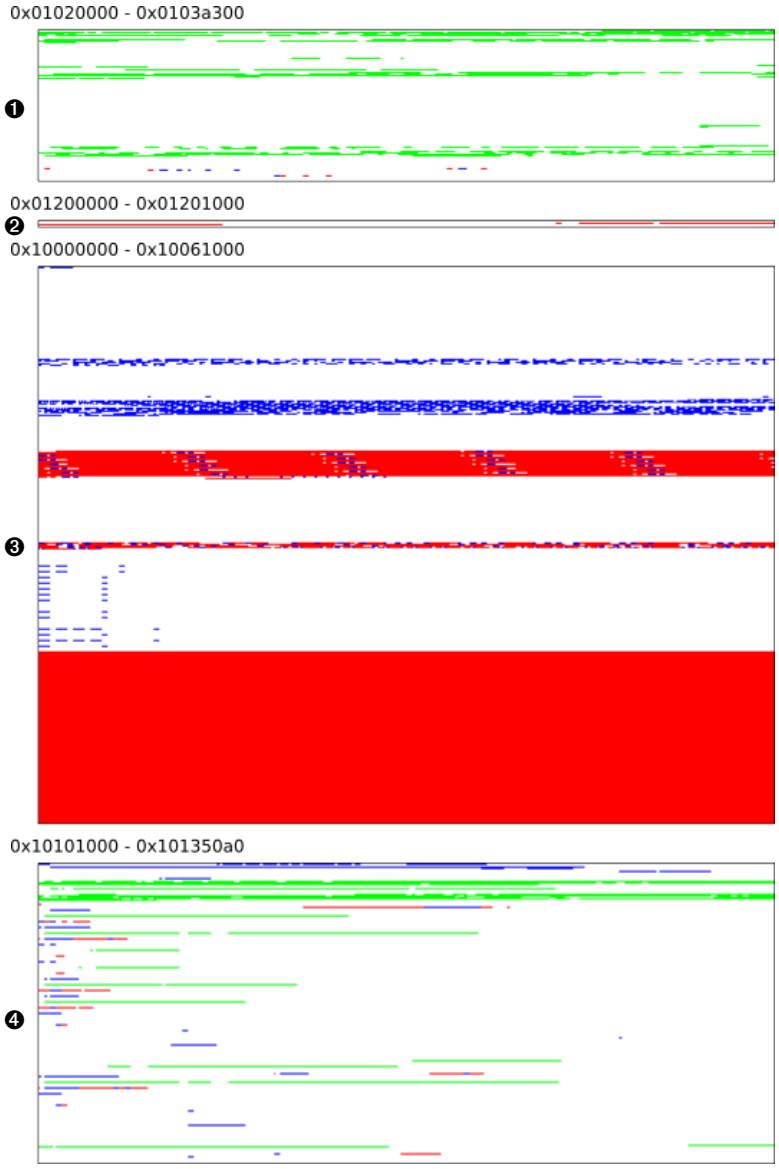


Figure 7.20 – Carte des accès mémoires réalisés par l'application ROSACE. À la différence de la figure 7.19, les accès mémoires effectués par l'initialisation du RTOS sont laissés. La légende est la même qu'en figure 7.16.

7.7 Conclusion

Ces travaux autour de l'analyse par exécution concolique permettent d'extraire un modèle des accès mémoires réalisés par une application temps-réelle de sûreté se conformant à quelques hypothèses qui nous paraissent raisonnables au regard des applications ciblées. Cette approche présente l'intérêt de capturer les accès mémoires effectués par le RTOS sous-jacent et ainsi d'obtenir une vision globale du profil des accès aux mémoires. Les informations collectées rendent compte de la répartition spatiale et temporelle des accès, ce qui permet à des algorithmes de placement mémoire une meilleure prise en compte des caches matériels. Nous n'avons pas développé de tels algorithmes exploitant les données produites par notre technique d'analyse : ceux-ci seront étudiés dans de futurs travaux.

L'analyse présentée ici souffre d'une difficulté pratique pouvant nuire à sa précision : son critère d'arrêt. Le problème étant indécidable, il nous semble difficile — même en multipliant les hypothèses — d'arriver à trouver automatiquement et de manière systématique un tic d'horloge logique à partir duquel les accès mémoires observés se répètent de manière invariante. La solution retenue, consistant à fixer une date maximale d'exploration, manque ainsi quelque peu d'élégance et contraint l'utilisateur à prendre le risque d'effectuer une analyse incomplète. Si cette limitation nous semble acceptable, il est nécessaire que cette hypothèse soit éprouvée lors d'une utilisation à plus large échelle et sur des applications industrielles représentatives. Une piste de recherche pour contourner ce problème consiste à exécuter des EA de manière indépendante. Toutefois, il est possible que cela dégrade les possibilités de concrétisation, rendant alors l'analyse imprécise.

Dans l'hypothèse où ce type d'analyse serait effectivement exploitable sur des applications temps-réelles de sûreté industrielles, des stratégies de partitionnement mémoires — pouvant être implémentées comme au chapitre 4 — permettraient par exemple d'implémenter des mécanismes de partitionnement de caches efficaces. Ceux-ci sont dans la pratique souvent limités par le manque d'informations sur le modèle des accès mémoires de l'application, mais notre méthode permet d'en extraire une approximation précise et de manière automatique.

Il est intéressant de constater un certain recouvrement entre notre analyse par co-exécution présentée au chapitre 6 et l'analyse par exécution concolique. Elles permettent toutes les deux d'identifier les mémoires auxquelles accèdent les EA composant l'application, facilitant l'écriture de groupes d'exclusion, indispensables à la mise en œuvre du principe de non-simultanéité décrit au chapitre 5. Si la méthode présentée dans ce chapitre ne permet pas de caractériser le matériel, elle permet toutefois une certaine forme de caractérisation du logiciel (découvrir l'utilisation des mémoires) hors ligne, ce qui permet de conduire des analyses fréquentes pendant la phase de développement.

IV

Quatrième partie

Conclusion générale

Chapitre 8

Conclusion et perspectives

Sommaire

8.1 Synthèse des contributions

Les travaux de cette thèse cherchent à établir des stratégies permettant de réduire la surestimation des temps d'exécution provisionnés lors de la conception d'applications temps-réelles de sûreté en mitigeant les interférences temporelles. Affiner la borne haute de ces temps d'exécution, souvent assimilée avec le WCET, sans dégrader la sûreté de ces systèmes, en permet un meilleur dimensionnement, limitant leurs coûts. Mitiger les interférences temporelles causées par une utilisation suboptimale de la mémoire nous permet d'atteindre cet objectif. Nous avons abordé ce vaste sujet sur plusieurs fronts, donnant lieu à quatre contributions principales. Rappelons que nous faisons l'hypothèse d'applications temps-réelles de sûreté construites avec Asterios, une technologie documentée au chapitre 3.

8.1.1 Partitionnement mémoire statique et vérifiable

L'exploitation pratique des stratégies de mitigation des interférences temporelles causées par une utilisation suboptimale de la mémoire (p. ex. mémoires caches, accès simultanés) repose sur une capacité de placement du logiciel en mémoire avec une fine granularité. Nous avons ainsi proposé une méthode de partitionnement spatiale, compatible avec les fortes contraintes de certification auxquelles sont soumises les applications industrielles (chapitre 4). Si ce problème est sans doute traité par chaque industriel, on en trouve peu de traces dans la littérature. Nous proposons un partitionnement mémoire statique, donc fixé à la génération de l'exécutable, combinant contraintes de placement et de protection mémoire. Notre système nous permet de produire des exécutables en s'appuyant sur des chaînes de compilation sur étagère, de sorte que les tables de protection mémoire soient pré-calculées et auto-descriptives. Ainsi, aucune opération, autre que le renseignement des permissions au matériel, n'est requis à l'exécution. Cela confère aux programmes que nous générons la capacité d'être vérifiés hors ligne par des outils compatibles avec les contraintes de certification, tout en réduisant la quantité de code embarqué à certifier.

8.1.2 Principe de non-simultanéité

Les plateformes multicœurs, utilisant plusieurs unités de calcul pour effectuer des accès simultanés à des ressources matérielles partagées posent un problème d'envergure à leur utilisation dans un contexte temps-réel de sûreté. Nous cherchons à prévenir par conception les accès simultanés, en les confinant à des fenêtres d'exécution bien identifiées. Grâce à un mécanisme d'automates temporels, nous pouvons vérifier automatiquement des propriétés de sûreté traduisant l'impossibilité de superposition de ces fenêtres temporelles d'intérêt. Dans la pratique, cela se traduit par la démonstration de l'impossibilité d'accès simultanés aux ressources matérielles partagées.

Ce concept, ou principe de non-simultanéité (chapitre 5), nécessite d'être pris en compte lors de la conception temporelle de l'application. C'est sa vérification au fur et à mesure du développement de l'application qui permet de construire un système exempt d'accès simultanés aux ressources matérielles partagées. Sa mise en œuvre nécessite toutefois d'avoir réalisé une étude complète des ressources matérielles accédées pour chaque fenêtre temporelle qui constitue le système. Cette étude peut être laborieuse, mais peut être grandement facilitée par nos contributions suivantes.

8.1.3 Analyse de systèmes multicœurs par co-exécution

Les interférences temporelles engendrées par l'utilisation du multicœurs peuvent être imprévisibles pour un œil extérieur. Plus le matériel sur étagère se complexifie, moins leurs modèles sont précis. Aussi, l'expérimentation empirique du matériel reste nécessaire pour en éprouver le comportement. Nous exploitons le principe des *co-runners* afin de caractériser la plateforme matérielle ainsi que le logiciel qui y est implanté. Ces deux phases sont respectivement désignées de « caractérisation matérielle » et de « caractérisation logicielle » (chapitre 6).

La caractérisation matérielle permet d'évaluer expérimentalement la réponse du matériel à un stress provoqué par un ou plusieurs cœurs auxiliaires. Un observateur

extérieur peut ainsi constater l'évolution des temps d'exécution en fonction des ressources matérielles sollicitées, indépendamment du logiciel final. Ces premiers résultats sont nécessaires à l'évaluation de la plateforme pour un usage temps-réel de sûreté lorsque plusieurs cœurs sont activés.

La caractérisation logicielle vise à tester les fenêtres temporelles constituant chaque tâche et à évaluer l'évolution de leurs temps d'exécution respectifs en réponse à un stress contrôlé. Elle se repose sur l'exécution de tests couvrants de l'applicatif, requis par les contraintes de certification. Si cette technique n'a pas vocation à se substituer à une analyse de WCET en bonne et due forme, elle permet toutefois d'identifier avec précision les ressources matérielles partagées accédées durant chaque fenêtre temporelle. Cela permet de collecter expérimentalement les données nécessaires à la vérification du principe de non-simultanéité.

8.1.4 Analyse de programmes par exécution concolique

Si les deux contributions précédentes concernent particulièrement les systèmes multicœurs, notre dernière contribution est plus générale et s'applique aussi bien aux systèmes monocœurs. Un obstacle important à l'élaboration des stratégies de placement mémoire permettant une meilleure utilisation de la mémoire (en particulier des mémoires caches) réside dans la difficulté à prévoir avec précision les accès mémoires effectués par l'application lors de son exécution. Nous nous sommes attaqués de front à ce problème, en considérant des applications développées suivant certaines restrictions, que nous jugeons acceptables dans un cadre temps-réel de sûreté.

En nous basant sur les mécanismes d'émulation et sur l'exécution symbolique, nous avons développé des principes d'exécution concolique permettant d'explorer hors ligne des binaires compilés et d'en extraire un modèle des accès mémoires (chapitre 7). Tous les accès mémoires réalisés (et réalisables) par une application sont collectés par une analyse hors ligne automatique, jusqu'à un tic d'horloge logique qui nous sert de critère d'arrêt. Ces accès sont ordonnés temporellement et retranscrivent l'utilisation spatiale de la mémoire. Les informations collectées sont amenées à être utilisées par des algorithmes tiers pour mettre en œuvre des stratégies de partitionnement mémoire permettant une meilleure utilisation des mémoires mises à disposition. Cela permet un partitionnement des caches, mais aussi l'application du principe de non-simultanéité dans le cas des systèmes multicœurs.

8.2 Perspectives

Le développement d'applications temps-réelles de sûreté est un domaine industriel dans lequel la propriété intellectuelle et le secret industriel ont une place importante. Il est ardu pour un tiers à ces industriels (même pour un partenaire) d'avoir accès à des applications représentatives. Aussi, nous n'avons pu bénéficier au cours de nos travaux d'applications industrielles représentatives sur lesquelles éprouver nos travaux. Toutefois, la maturation et l'industrialisation des résultats de cette thèse pourront être intégrés au produit commercial Asterios et ainsi à moyen ou long terme côtoyer la réalité industrielle. Les retours que nous pourrons avoir aiguilleront le futur des travaux développés ici. Nous prévoyons tout de même d'approfondir certains sujets.

8.2.1 Vérifier la non-simultanéité à moindre complexité

Le principe de non-simultanéité, détaillé au chapitre 5, peut être vérifié grâce à un algorithme qui se base sur l'état de l'art de la théorie des langages. Il se base sur les automates UNFA pour déterminer l'expression d'un langage unaire exprimé par ces automates. Notre solution proposée n'admet ni faux positifs, ni faux négatifs, mais au prix d'une complexité algorithmique en $O(n^4)$. Une heuristique a été développée pour Krono-Safe (non publiée) : elle est en temps linéaire, mais est parfois sensible à des faux positifs. Une piste d'amélioration serait de proposer un algorithme offrant le même niveau de garanties, mais avec une complexité algorithmique moindre.

8.2.2 Exclusion temporelle des frames à l'intérieur des EA

Le principe de non-simultanéité (chapitre 5) permet de décrire des exclusions temporelles entre EA. Si ce point distingue nos travaux de l'état de l'art, il peut être intéressant d'y adjoindre un mécanisme d'exclusion temporelle des *frames* lorsqu'une RSF est impliquée. Ce travail au sein du plan d'ordonnancement est plus répandu dans la littérature [Bon+12 ; Bec+16]. Combiner ces deux approches permettrait de valider les contraintes de non-simultanéité en deux temps. Une première vérification, sur les EA, offre de plus fortes garanties : quelle que soit la stratégie de provision de temps choisie, tant que les groupes d'exclusion sont respectés, aucune interférence causée par des accès simultanés à une ressource partagée ne peuvent se produire. Toutefois, dans la pratique, il est possible que ces contraintes ne puissent être honorées. Si un recouvrement temporel est détecté, il serait intéressant de procéder à une nouvelle analyse pour que le générateur de plan d'ordonnancement puisse placer les *frames* de sorte que les tâches qu'elles hébergent ne s'exécutent simultanément. Suivre cette piste demande la maîtrise du modèle d'exécution et en particulier du comportement de l'ordonnanceur.

8.2.3 Caractériser complètement une cible matérielle

La méthode d'analyse d'une plateforme par co-exécution, détaillée au chapitre 6 comprend une étape dite de caractérisation matérielle. Si nous avons effectué quelques caractérisations du matériel, nous n'avons pas réalisé l'exercice d'en effectuer une en profondeur. Il pourrait être intéressant de compléter celle réalisée en section 6.3.5 en faisant varier davantage de paramètres. Par exemple la plage des adresses mémoires stressées, celle de l'applicatif de contrôle, ou encore d'autres paramètres propres au matériel comme les politiques d'arbitrage des bus. Un matériel plus élaboré (de la famille QorIQ par exemple) pourrait aussi être testé, afin d'éprouver sa robustesse.

8.2.4 Amélioration du moteur d'exécution concolique

Le moteur d'exécution concolique, détaillé au chapitre 7, est un prototype qui a permis d'éprouver les concepts d'exploration des accès mémoires que nous avons posés. Certains choix d'implémentation sont discutables et méritent d'être revus.

Le solutionneur SMT utilisé ne gère pas les théories des nombres flottants, ce qui peut s'avérer limitant pour les applications temps-réelles de sûreté. En effet, il s'agit

souvent de systèmes cyber-physiques, manipulant des valeurs « réelles » pour effectuer des tâches comme du contrôle moteur. Un support des nombres flottants permettrait de rendre concrètes davantage d'opération, réduisant le nombre de branchements à explorer et améliorant peut-être la précision du modèle.

L'exploration d'EA individuelles est intéressante à investiguer. Si d'un point de vue pratique l'exploration d'une tâche entière permet plus de concrétisation, elle se heurte au besoin de trouver un critère d'arrêt, qui est aujourd'hui laissé à la charge de l'utilisateur. Proposer les deux approches plutôt qu'une seule est une amélioration dont la pertinence devra être éprouvée par la pratique.

Les principes de l'exécution concolique sont adhérents à Asterios. Nous n'avons pas étudié leur transposition à d'autres modèles de calcul et d'exécution. Il peut être intéressant de faire l'exercice d'adapter ces principes à un cadre différent ou plus large.

8.2.5 Exploitation des résultats de l'exploration concolique

Nous nous sommes concentrés dans cette thèse sur des méthodes d'analyse. Celles-ci permettent de collecter des informations de valeur, indispensable à la mise en œuvre de stratégies d'amélioration des propriétés temporelles d'intérêt comme le WCET. En effet, ces stratégies sont susceptibles d'être dépendantes de l'applicatif, qui est contrôlé par un industriel. Il nous semblait préférable de fournir des outils lui fournissant des données clef en main qu'il puisse utiliser en fonction de ses propres contraintes.

Nous pouvons toutefois faire l'exercice d'utiliser les données collectées sur des applications illustratives afin de montrer des exemples de leur utilisation. Il serait particulièrement intéressant de mettre en place des stratégies de partitionnement de cache, peu utilisées dans la pratique, car complexe à implémenter faute de données détaillées sur les accès mémoires effectués.

8.2.6 Publication des dernières contributions

À l'heure où nous écrivons ces lignes, nos deux dernières contributions majeures que sont l'analyse par co-exécution [Guy+20b] et par exécution concolique [GV21] ne sont pas encore publiées. Deux articles ont été rédigés et nous travaillons à leur publication dans des journaux ou conférences sélectives.

Glossaire

CFG (Control Flow Graph)
Graphe de flot de contrôle. 120, 122, 123, 124, 126, 127, 128, 132

COTS (Component Off-The-Shelf)
Composant sur étagère : matériel ou logiciel utilisable par un tiers sans besoin d'investir dans son développement. 10, 16

CRC (Cyclic Redundancy Check)
Valeur numérique permettant la détection d'erreurs de transfert de données. 71, 73, 74

DSE (Dynamic Symbolic Execution)
Exploration dynamique d'un programme en ayant recours à des techniques d'exécution symbolique. 111, 124

EA (Elementary Action)
En français : action élémentaire. Exécution d'un bloc d'instructions délimité par deux instructions PsyC `advance`.. 28, 31, 32, 36, 71, 73, 74, 75, 76, 77, 78, 80, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 111, 117, 131, 134, 140, 146, 147

EDF (Earliest Deadline First)
Stratégie d'ordonnancement privilégiant parmi les tâches à élire celle dont l'échéance est la plus proche. 12, 27

ESD (Earliest Start Date)
Contrainte temporelle traduisant une date de début au plus tôt. 25

GPR (General-Purpose Registers)
Ensemble de registres matériels utilisables par l'applicatif logiciel. 124

HIL (Hardware-In-the-Loop)
Dans un processus de développement industriel, le HIL est l'étape durant laquelle le logiciel est implanté sur matériel et au cours de laquelle les tests de bout-en-bout sont réalisés. 111

IMA (Integrated Modular Avionics)
Réseau temps-réel de nœuds de calculs, particulièrement utilisé dans les domaines avioniques. 10

ISA (Instruction Set Architecture)
Ensemble des instructions machines supportées par une architecture matérielle donnée. 114, 119, 124, 130

ISR (Interrupt Service Routine)
Routine d'interruption : fonction logicielle exécutée à la survenue d'une interruption matérielle. 11, 12

LET (Logical Execution Time)
Paradigme de programmation dans lequel l'environnement d'exécution n'est altéré que lors de l'écoulement du temps logique. 29, 87, 129

LR (Link Register)
Registre matériel contenant l'adresse à laquelle le flot d'instructions doit reprendre après une rupture (par exemple due à un appel de fonction). 124

MIL (Model-In-the-Loop)
Dans un processus de développement industriel, le MIL est l'étape durant laquelle le système est modélisé et au cours de laquelle l'intégration travaille sur le modèle produit. 111

MMU (Memory Management Unit)
Unité matérielle de gestion de la mémoire : composant micro-architectural en charge de la gestion de la pagination mémoire et de l'espace d'adressage virtuel. 36, 38, 45, 46, 52, 54, 55, 151

MPU (Memory Protection Unit)
Unité matérielle de protection de la mémoire : composant micro-architectural en charge de l'application de droits d'accès mémoire sur certaines plages d'adresses d'un espace d'adressage plat (sans pagination). 36, 38, 42, 43, 55

PC (Program Counter)
Registre matériel contenant l'adresse depuis laquelle lire la prochaine instruction à exécuter. 124, 126, 127

PGO (Profile-Guided Optimization)
Technique d'optimisation du logiciel se basant sur une observation de son exécution sur matériel. Les données collectées permettent de générer une nouvelle version du logiciel, optimisée par rapport à la précédente. 112

RSF (Repetitive Sequence of Frames)
Politique d'ordonnancement s'appuyant sur des tables d'ordonnancement générées hors ligne. 32, 33, 74, 75, 78, 146

RTOS (Real-Time Operating System)
Système d'exécution temps-réel. Dans ces travaux, le noyau occupe la partie prépondérante de ce composant d'architecture logicielle. 23, 49, 50, 51, 55, 77, 92, 93, 111, 113, 117, 118, 119, 120, 124, 128, 129, 130, 135, 136, 139, 140

SHIM (Software-Hardware Interface for Multi-many-core)
Standard IEEE permettant aux constructeurs de matériel de fournir à ses utilisateurs une description formalisée de la plateforme matérielle. 84

SIL (Software-In-the-Loop)
Dans un processus de développement industriel, le SIL est l'étape durant laquelle le logiciel est développé et au cours de laquelle l'intégration travaille sur ce logiciel. C'est durant cette phase que sont notamment écrits les tests unitaires. 111

SMT (Satifiability Modulo Theories)
Problème de décision appliqué à des formules logiques. 3, 124, 126, 130, 146

TCA (Time-Constrained Automata)
Automates contraints en temps. Il s'agit d'un modèle de calcul permettant d'associer des contraintes temporelles à des séquences de calculs. 25, 26, 27, 28, 29, 32, 60, 61, 62, 63, 64, 65, 66, 68, 70, 71, 72, 80, 82, 91, 92

TDM (Time-Division Multiplexing)
Technique de découpe du temps d'exécution d'un ensemble de tâches en une succession d'intervalles temporels. 13, 21, 32, 60

TLB (Translation Lookaside Buffer)
Cache interne à une MMU permettant une mise à disposition rapide des dernières traductions d'adresses virtuelles. 18, 45, 46

UNFA (Unary Non-deterministic Finite Automaton)
Automate fini unaire et non-déterministe. 64, 66, 146

Vcpu (Virtual CPU)
Cœur processeur émulé. 119, 124, 127

VT (Variable Temporelle)
Une variable temporelle est un des moyens de communications inter-agents mis à disposition par le PsyC. Il s'agit d'un flot discret où chaque donnée est estampillée par un tic d'horloge logique. 29, 30, 31, 117, 118, 124, 130

WCET (Worst-Case Execution Time)
Temps d'exécution dans le pire cas. C'est une grandeur théorique faisant office de borne supérieure aux temps d'exécution observables en toutes circonstances d'un ensemble de calculs. 1, 2, 6, 14, 15, 16, 17, 18, 19, 21, 32, 34, 35, 36, 55, 74, 75, 77, 79, 80, 84, 90, 94, 97, 98, 99, 100, 109, 110, 111, 112, 115, 129, 135, 143, 145, 147

Bibliographie

[Abe+14] Jaume ABELLA, Damien HARDY, Isabelle PUAUT, Eduardo QUINONES et Francisco J CAZORLA. « On the comparison of deterministic and probabilistic WCET estimation techniques ». In : *26th Euromicro Conference on Real-Time Systems*. 2014, p. 266-275. DOI : 10.1109/ECRTS.2014.16.

[ACD10] Christophe AUSSAGUÈS, Damien CHABROL et Vincent DAVID. *Method for the deterministic execution and synchronisation of an information processing system comprising a plurality of processing cores executing system tasks*. Patent WO 2010/043706 A2. Avr. 2010.

[Ach20] Reto ACHERMANN. « On Memory Addressing ». Thèse de doct. ETH Zurich, 2020. DOI : 10.3929/ethz-b-000400029.

[Agi+17] Irune AGIRRE, Jaume ABELLA, Mikel AZKARATE-ASKASUA et Francisco J CAZORLA. « On the tailoring of CAST-32A certification guidance to real COTS multicore architectures ». In : *12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2017, p. 1-8. DOI : 10.1109/SIES.2017.7993376.

[Air15] AIRLINES ELECTRONIC COMMITTEE. *Avionics Application Software Standard Interface - Part 1 : Required Services*. ARINC 653P1. Airlines Electronic Committee, août 2015.

[Alm+10] Gabriel Marchesan ALMEIDA, Sameer VARYANI, Rémi BUSSEUIL, Gilles SASSATELLI, Pascal BENOIT, Lionel TORRES, Everton Alceu CARARA et Fernando Gehm MORAES. « Evaluating the impact of task migration in multi-processor systems-on-chip ». In : *Proceedings of the 23rd symposium on Integrated circuits and system design*. 2010, p. 73-78. DOI : 10.1145/1854153.1854174.

[Alt+16] Sebastian ALTMEYER, Roeland DOUMA, Will LUNNISS et Robert I DAVIS. « On the effectiveness of cache partitioning in hard real-time systems ». In : *Real-Time Systems* 52.5 (2016), p. 598-643. DOI : 10.1007/s11241-015-9246-8.

[Ama+20] Roberto AMADINI, Graeme GANGE, Peter SCHACHTE, Harald SØNDERGAARD et Peter J STUCKEY. « Abstract Interpretation, Symbolic Execution and Constraints ». In : *Recent Developments in the Design and Implementation of Programming Languages*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020. DOI : 10.4230/OASIcs.Gabbrielli.7.

[And94] George E Andrews. *Number theory*. Courier Corporation, 1994. isbn : 978-0-486-68252-5.

[AS87] Bowen Alpern et Fred B Schneider. « Recognizing safety and liveness ». In : *Distributed computing* 2.3 (1987), p. 117-126. doi : 10.1007/BF01782772.

[Aug+07] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A Penry, Olivier Temam et Neil Vachharajani. « UNISIM : An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development ». In : *IEEE Computer Architecture Letters* 6.2 (2007), p. 45-48. doi : 10.1109/L-CA.2007.12.

[Aus+10] Christophe Aussaguès, Damien Chabrol, Vincent David, Didier Roux, Natalia Willey, Arnaud Tournadre et Marc Graniou. « PharOS, a multicore OS ready for safety-related automotive systems : results and future prospects ». In : *5th European Congress on Embedded Real Time Software and Systems (ERTS)*. Toulouse, France, 2010.

[Axe+14] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm et Wang Yi. « Building timing predictable embedded systems ». In : *ACM Transactions on Embedded Computing Systems (TECS)* 13.4 (2014), p. 1-37. doi : 10.1145/2560033.

[Bak+12] Stanley Bak, Gang Yao, Rodolfo Pellizzoni et Marco Caccamo. « Memory-aware scheduling of multicore task sets for real-time systems ». In : *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2012, p. 300-309. doi : 10.1109/RTCSA.2012.48.

[Bak02] Thomas G Baker. « Lessons learned integrating COTS into systems ». In : *International Conference on COTS-Based Software Systems*. Springer. 2002, p. 21-30. doi : 10.1007/3-540-45588-4_3.

[Bal+10] Clément Ballabriga, Hugues Cassé, Christine Rochange et Pascal Sainrat. « OTAWA : an open toolbox for adaptive WCET analysis ». In : *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer. 2010, p. 35-46. doi : 10.1007/978-3-642-16256-5_6.

[BB09] Robert Brummayer et Armin Biere. « Boolector : An Efficient SMT Solver for Bit-Vectors and Arrays ». In : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. T. 5505. 2009, p. 174-177. doi : 10.1007/978-3-642-00768-2_16.

[Bec+16] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis et Thomas Nolte. « Contention-free execution of automotive applications on a clustered many-core platform ». In : *28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2016, p. 14-24. doi : 10.1109/ECRTS.2016.14.

[Bel05] Fabrice Bellard. « QEMU, a fast and portable dynamic translator ». In : *USENIX annual technical conference, FREENIX Track*. T. 41. California, USA. 2005, p. 46.

[Ber+07] Guillem Bernat, Robert Davis, Nick Merriam, John Tuffen, Andrew Gardner, Michael Bennett et Dean Armstrong. « Identifying opportunities for worst-case execution time reduction in an avionics system ». In : *Ada User Journal* 28.3 (2007), p. 189-195.

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman et Yunshan Zhu. « Bounded model checking ». In : *Adv. Comput.* 58 (2003), p. 117-148. doi : 10.1016/S0065-2458(03)58003-2.

[Bie+18] Pierre Bieber, Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Claire Pagetti, Olivier Poitou, Thomas Polacsek, Luca Santinelli et Nathanaël Sensfelder. « A model-based certification approach for multi/many-core embedded systems ». In : *9th European Congress on Embedded Real Time Software and Systems (ERTS)*. Toulouse, France, 2018.

[Bin+11] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill et David A. Wood. « The gem5 simulator ». In : *ACM SIGARCH computer architecture news* 39.2 (2011), p. 1-7. doi : 10.1145/2024716.2024718.

[Bin+14a] Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset et Alain Mérigot. « Studying co-running avionic real-time applications on multi-core COTS architectures ». In : *7th European Congress on Embedded Real Time Software and Systems (ERTS)*. Toulouse, France, fév. 2014.

[Bin+14b] Jingyi Bin, Sylvain Girbal, Daniel Gracia Perez et Alain Merigot. « Using monitors to predict co-running safety-critical hard real-time benchmark behavior ». In : *International Conference on Information and Communication Technology for Embedded Systems (ICICTES)*. 2014.

[BMB10] Adam Betts, Nicholas Merriam et Guillem Bernat. « Hybrid measurement-based WCET analysis at the source level using object-level traces ». In : *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010. doi : 10.4230/OASIcs.WCET.2010.54.

[BO16] Jacques Brygier et Mehmet Oezer. « Safety and security for the internet of things ». In : *8th European Congress on Embedded Real Time Software and Systems (ERTS)*. TOULOUSE, France, jan. 2016.

[Boe88] Barry W Boehm. « A spiral model of software development and enhancement ». In : *Computer* 21.5 (1988), p. 61-72. doi : 10.1109/2.59.

[Bon+12] Frédéric Boniol, Hugues Cassé, Eric Noulard et Claire Pagetti. « Deterministic execution model on COTS hardware ». In : *International Conference on Architecture of Computing Systems*. Springer. 2012, p. 98-110. doi : 10.1007/978-3-642-28293-5_9.

[Bon+18] Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Kevin Delmas, Claire Pagetti, Thomas Polacsek et Nathanaël Sensfelder. « PHY-LOG : a model-based certification framework ». In : *37th Digital Avionics Systems Conference (DASC)*. IEEE. 2018, p. 1-9.

[BP05] François Bodin et Isabelle Puaut. « A WCET-oriented static branch prediction scheme for real time systems ». In : *17th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2005, p. 33-40.

[BPS19] Frédéric Boniol, Claire Pagetti et Nathanaël Sensfelder. « Identification of multi-core interference ». In : *19th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE. 2019, p. 98-106. doi : 10.1109/HASE.2019.00024.

[BT18] Clark Barrett et Cesare Tinelli. « Satisfiability modulo theories ». In : *Handbook of Model Checking*. Springer, 2018, p. 305-343.

[Bur90] George Burns. *Technology trends and fab costs : an overview*. Rapp. tech. 1290 Ridder Park Drive, San Jose, CA 95131-2398 : Dataquest, nov. 1990.

[CAD09] Damien Chabrol, Christophe Aussaguès et Vincent David. « A spatial and temporal partitioning approach for dependable automotive systems ». In : *Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2009, p. 1-8. doi : 10.1109/ETFA.2009.5347125.

[Cam+08] J S Camier, C Aussagues, M. Lemerre et V David. « A Toolchain For Designing And Implementing Efficient, Flexible And Safety-Critical Applications ». In : *4th European Congress on Embedded Real Time Software and Systems (ERTS)*. Toulouse, France, jan. 2008.

[Car+20] Françoise Caron, Dominique Blouin, Paolo Crisafulli et Cristian Maxim. « Seamless Integration between Real-time Analyses and Systems Engineering with the PST Approach ». In : *International Systems Conference (SysCon)*. IEEE. 2020, p. 1-8.

[CBS13] Hugues Cassé, Florian Birée et Pascal Sainrat. « Multi-architecture value analysis for machine code ». In : *13th International Workshop on Worst-Case Execution Time Analysis (WCET)*. 2013, pp-42. doi : 10.4230/OASIcs.WCET.2013.42.

[CC18] Thomas Carle et Hugues Cassé. « Reducing timing interferences in real-time applications running on multicore architectures ». In : *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. 2018, p. 1-11. doi : 10.4230/OASIcs.WCET.2018.3.

[CC77] Patrick Cousot et Radhia Cousot. « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints ». In : *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, p. 238-252. doi : 10.1145/512950.512973.

[CC92] Patrick Cousot et Radhia Cousot. « Abstract interpretation frameworks ». In : *Journal of logic and computation* 2.4 (1992), p. 511-547. doi : 10.1093/logcom/2.4.511.

[CDE08] Cristian Cadar, Daniel Dunbar et Dawson R. Engler. « KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs ». In : *OSDI*. T. 8. 2008, p. 209-224.

[Cer16] Certification Authorities Software Team. *CAST-32A — Multi-core processors*. Rapp. tech. Nov. 2016.

[Cha+12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert et David Brumley. « Unleashing mayhem on binary code ». In : *Symposium on Security and Privacy*. IEEE. 2012, p. 380-394. doi : 10.1109/SP.2012.31.

[Cha+13] Damien Chabrol, Didier Roux, Vincent David, Mathieu Jan, Moha Ait Hmid, Patrice Oudin et Gilles Zeppa. « Time- and angle-triggered real-time kernel ». In : *Design, Automation, and Test in Europe (DATE)*. IEEE. 2013, p. 1060-1062. doi : 10.7873/DATE.2013.223.

[Cha+14] Damien Chabrol, Vincent David, Patrice Oudin, Gilles Zeppa et Mathieu Jan. « Freedom from interference among time-triggered and angle-triggered tasks : a powertrain case study ». In : *7th European Congress on Embedded Real Time Software and Systems (ERTS)*. Toulouse, France, 2014.

[Cor+11] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard et Claire Pagetti. « Developing critical embedded systems on multicore architectures : the Prelude-SchedMCore toolset ». In : *19th International Conference on Real-Time and Network Systems (RTNS)*. 2011, p. 107-116.

[CS13] Cristian Cadar et Koushik Sen. « Symbolic execution for software testing : three decades later ». In : *Communications of the ACM* 56.2 (2013), p. 82-90. doi : 10.1145/2408776.2408795.

[Cuc+12] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones et Francisco J Cazorla. « Measurement-based probabilistic timing analysis for multi-path programs ». In : *24th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2012, p. 91-101. doi : 10.1109/ECRTS.2012.31.

[Dar+19] Mickaël Dardaillon, Stefanos Skalistis, Isabelle Puaut et Steven Derrien. « Reconciling Compiler Optimizations and WCET Estimation Using Iterative Compilation ». In : *Real-Time Systems Symposium (RTSS)*. IEEE. 2019, p. 133-145. doi : 10.1109/RTSS46320.2019.00022.

[Dav+04] Vincent David, Christophe Aussaguès, Stéphane Louise, Philippe Hilsenkopf, Bertrand Ortolo et Christophe Hessler. « The OASIS based qualified display system ». In : *Proceedings ANS (NPIC & HMIT)* (2004), p. 33.

[Dav+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet et Jean-Yves Marion. « BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis ». In : *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. T. 1. IEEE. 2016, p. 653-656. doi : 10.1109/SANER.2016.43.

[Dav+98] Vincent David, Jean Delcoigne, Evelyne Leret, Alain Ourghanlian, Philippe Hilsenkopf et Philippe Paris. « Safety properties ensured by the OASIS model for safety critical real-time systems ». In : *Computer Safety, Reliability and Security, 17th International Conference, SAFECOMP'98*. T. 1516. Lecture Notes in Computer Science. Springer. Heidelberg, Germany : Springer, 1998, p. 45-59. doi : 10.1007/3-540-49646-7_4.

[Dav20] Robert I. Davis. « Guest editorial : Special Issue on Predictable multicore systems ». In : *Real Time Syst.* 56.2 (2020), p. 121-123. doi : 10.1007/s11241-020-09348-x.

[DB08] Leonardo De Moura et Nikolaj Bjørner. « Z3 : An efficient SMT solver ». In : *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, p. 337-340. doi : 10.1007/978-3-540-78800-3_24.

[DBC14] Vincent David, Adrien Barbot et Damien Chabrol. « Dependable real-time system and mixed criticality : Seeking safety, flexibility and efficiency with Kron-OS ». In : *Ada User Journal* 35.4 (2014), p. 259-265.

[DD07] Vincent David et Jean Delcoigne. *Security method making deterministic real time execution of multitask applications of control and command type with error confinement*. US Patent 7,299,383. Nov. 2007.

[EL07] Stephen A Edwards et Edward A Lee. « The case for the precision timed (PRET) machine ». In : *Proceedings of the 44th annual Design Automation Conference*. ACM. 2007, p. 264-265. doi : 10.1145/1278480.1278545.

[Fer+15] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega et Francisco J Cazorla. « Resource usage templates and signatures for COTS multicore processors ». In : *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2015, p. 1-6. doi : 10.1145/2744769.2744901.

[FH05] Christian Ferdinand et Reinhold Heckmann. « Verifying timing behavior by abstract interpretation of executable code ». In : *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer. 2005, p. 336-339. doi : 10.1007/11560548_26.

[FJK20] Antoine Ferlin, Eric Jenn et Marc Kaufmann. « Accounting for interferences in the design of Time-Triggered Applications ». In : *10th European Congress on Embedded Real Time Software and Systems (ERTS)*. 2020.

[FL10] Heiko Falk et Paul Lokuciejewski. « A compiler framework for the reduction of worst-case execution times ». In : *Real-Time Systems* 46.2 (2010), p. 251-300. doi : 10.1007/s11241-010-9101-x.

[FLT06] Heiko Falk, Paul Lokuciejewski et Henrik Theiling. « Design of a wcet-aware c compiler ». In : *IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*. IEEE. 2006, p. 121-126. doi : 10.1109/ESTMED.2006.321284.

[Fre09] Freescale Semicondutor, Inc. *P2020DS – a P2020 Development Platform, Rev 1.0. alpha*. Avr. 2009.

[Fre12] Freescale Semicondutor, Inc. *P2020 QorIQ Integrated Processor Reference Manual, Rev2*. Déc. 2012.

[Fre15] Freescale Semicondutor, Inc. *QorIQ T1040 Reference Manual, Rev1*. Août 2015.

[FW98] Christian Ferdinand et Reinhard Wilhelm. « On predicting data cache behavior for real-time systems ». In : *Languages, Compilers, and Tools for Embedded Systems*. Springer. 1998, p. 16-30. doi : 10.1007/BFb0057777.

[GH20] Jean Guyomarc'h et Jean-Baptiste Hervé. « Static and Verifiable Memory Partitioning for Safety-Critical Systems ». In : *International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Coimbra, Portugal : IEEE, oct. 2020, p. 79-84. doi : 10.1109/ISSREW51248.2020.00041.

[GMO19] Jean Guyomarc'h, Alain Mérigot et Emmanuel Ohayon. « Reducing the variability of execution times in safety-critical real-time systems (poster) ». In : *International School on Rewriting*. Paris, France, juill. 2019.

[Guy+20a] Jean Guyomarc'h, François Guerret, Bilal El Mejjati, Emmanuel Ohayon, Bastien Vincke et Alain Mérigot. « Non-Simultaneity as a Design Constraint ». In : *27th International Symposium on Temporal Representation and Reasoning (TIME)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020. doi : 10.4230/LIPIcs.TIME.2020.13.

[Guy+20b] Jean Guyomarc'h, François Guerret, Emmanuel Ohayon et Bastien Vincke. « Towards Trusted Timing Analysis of Safety-Critical Multi-Core COTS Systems ». Déc. 2020.

[GV21] Jean Guyomarc'h et Bastien Vincke. « Concolic Execution as an Analysis Technique to Leverage Timing Anomalies Mitigations ». Mai 2021.

[Heb+18] Farouk Hebbache, Mathieu Jan, Florian Brandner et Laurent Pautet. « Shedding the shackles of time-division multiplexing ». In : *Real-Time Systems Symposium (RTSS)*. IEEE. 2018, p. 456-468. doi : 10.1109/RTSS.2018.00059.

[HF04] Reinhold Heckmann et Christian Ferdinand. « Worst-case execution time prediction by static program analysis ». In : *In 18th International Parallel and Distributed Processing Symposium (IPDPS), pages 26–30. IEEE Computer Society*. 2004. doi : 10.1007/978-1-4020-8157-6_29.

[HJR16] Sebastian Hahn, Michael Jacobs et Jan Reineke. « Enabling compositionality for multicore timing analysis ». In : *Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS)*. Brest,France, oct. 2016, p. 299-308. doi : 10.1145/2997465.2997471.

[HN98] Jerry L Hintze et Ray D Nelson. « Violin plots : a box plot-density trace synergism ». In : *The American Statistician* 52.2 (1998), p. 181-184. doi : 10.1080/00031305.1998.10480559.

[HRP17] Damien Hardy, Benjamin Rouxel et Isabelle Puaut. « The Heptane Static Worst-Case Execution Time Estimation Tool ». In : *17th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017. doi : 10.4230/OASIcs.WCET.2017.8.

[IEC06] IEC-60880. *Nuclear power plants – Instrumentation and control systems important to safety*. Rapp. tech. 2006.

[IEE19] IEEE 2804-2019. *IEEE Standard for Software-Hardware Interface for Multi-Many-Core*. Rapp. tech. IEEE Computer Society, nov. 2019. doi : 10.1109/IEEESTD.2020.8985663.

[Ish+13] Takuya Ishikawa, Toshikazu Kato, Shinya Honda et Hiroaki Takada. « Investigation and improvement on the impact of TLB misses in real-time systems ». In : *Proc. of OSPERT* (2013).

[ISO17] ISO/IEC 21778 :2017. *The JSON data interchange syntax*. Rapp. tech. ISO, nov. 2017.

[ISO18] ISO-26262. *Road vehicles — Functional safety*. International Organization for Standardization, 2018.

[JCD11] Mathieu Jan, Jean-Sylvain Camier et Vincent David. « Scheduling safety-critical real-time bus accesses using Time-Constrained Automata ». In : *19th International Conference on Real-Time and Network Systems (RTNS)*. 2011, p. 87-96.

[Jea+13] Xavier Jean, Marc Gatti, David Faura, Laurent Pautet et Thomas Robert. « A software approach for managing shared resources in multicore IMA systems ». In : *IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*. IEEE. 2013, p. 7D1-1.

[Käs+10] Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné et Xavier Rival. « Astrée : Proving the absence of runtime errors ». In : *5th European Congress Embedded Real Time Software and Systems (ERTS)*. 2010, p. 9.

[Käs+18] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy et Sandrine Blazy. « CompCert : Practical experience on integrating and qualifying a formally verified optimizing compiler ». In : *9th European Congress Embedded Real Time Software and Systems (ERTS)*. 2018, p. 1-9.

[Käs+19] Daniel KÄSTNER, Markus PISTER, Simon WEGENER et Christian FERDINAND. « TimeWeaver : A Tool for Hybrid Worst-Case Execution Time Analysis ». In : *19th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019. DOI : 10.4230/OASIcs.WCET.2019.1.

[Kin76] James C KING. « Symbolic execution and program testing ». In : *Communications of the ACM* 19.7 (1976), p. 385-394. DOI : 10.1145/360248.360252.

[Kir89] David B KIRK. « SMART (strategic memory allocation for real-time) cache design ». In : *1989 Real-Time Systems Symposium*. 1989, p. 229-230. DOI : 10.1109/REAL.1989.63574.

[Kop11] Hermann KOPETZ. *Real-time systems : design principles for distributed embedded applications*. Real-Time Systems Series. Springer, 2011. ISBN : 978-1-4419-8236-0. DOI : 10.1007/978-1-4419-8237-7.

[Kos+16] Leonidas KOSMIDIS, Davide COMPAGNIN, David MORALES, Enrico MEZZETTI, Eduardo QUIÑONES, Jaume ABELLA, Tullio VARDANEGA et Francisco J. CAZORLA. « Measurement-Based Timing Analysis of the AURIX Caches ». In : *16th International Workshop on Worst-Case Execution Time Analysis (WCET)*. T. 55. 2016, 9 :1-9 :11. DOI : 10.4230/OASIcs.WCET.2016.9.

[KP08] Raimund KIRNER et Peter PUSCHNER. « Obstacles in worst-case execution time analysis ». In : *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE. 2008, p. 333-339. DOI : 10.1109/ISORC.2008.65.

[KS12] Christoph M KIRSCH et Ana SOKOLOVA. « The logical execution time paradigm ». In : *Advances in Real-Time Systems*. 2012, p. 103-120. DOI : 10.1007/978-3-642-24349-3_5.

[Kuz+12] Volodymyr KUZNETSOV, Johannes KINDER, Stefan BUCUR et George CANDEA. « Efficient state merging in symbolic execution ». In : *Acm Sigplan Notices* 47.6 (2012), p. 193-204. DOI : 10.1145/2254064.2254088.

[LA04] Chris LATTNER et Vikram ADVE. « LLVM : A compilation framework for lifelong program analysis & transformation ». In : *International Symposium on Code Generation and Optimization*. IEEE. 2004, p. 75-86. DOI : 10.1109/CGO.2004.1281665.

[Lam78] Leslie LAMPORT. « Time, Clocks, and the Ordering of Events in a Distributed System ». In : *Commun. ACM* 21.7 (1978), p. 558-565. DOI : 10.1145/359545.359563.

[Lee08] Edward A LEE. « Cyber physical systems : Design challenges ». In : *11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*. 2008, p. 363-369. DOI : 10.1109/ISORC.2008.25.

[Lem+10] Matthieu Lemerre, Vincent David, Christophe Aussagues et Guy Vidal-Naquet. « An introduction to time-constrained automata ». In : *Proceedings of the 3rd Interaction and Concurrency Experience Workshop (ICE)*. T. 38. Juin 2010, p. 83-98. doi : 10.4204/EPTCS.38.9.

[Lem+11] Matthieu Lemerre, Emmanuel Ohayon, Damien Chabrol, Mathieu Jan et Marie-Benedicte Jacques. « Method and tools for mixed-criticality real-time applications within PharOS ». In : *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE. 2011, p. 41-48. doi : 10.1109/ISORCW.2011.15.

[Lis14] Björn Lisper. « SWEET–a tool for WCET flow analysis ». In : *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2014, p. 482-485. doi : 10.1007/978-3-662-45231-8_38.

[LO12] Matthieu Lemerre et Emmanuel Ohayon. « A model of parallel deterministic real-time computation ». In : *33rd Real-Time Systems Symposium*. IEEE. 2012, p. 273-282. doi : 10.1109/RTSS.2012.78.

[Loh+19] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A Lee et Alberto Sangiovanni-Vincentelli. « Reactors : A deterministic model for composable reactive systems ». In : *Cyber Physical Systems. Model-Based Design*. 2019, p. 59-85. doi : 10.1007/978-3-030-41131-2_4.

[Lou+11] Stéphane Louise, Matthieu Lemerre, Christophe Aussagues et Vincent David. « The OASIS kernel : A framework for high dependability real-time systems ». In : *13th International Symposium on High-Assurance Systems Engineering*. IEEE. 2011, p. 95-103. doi : 10.1109/HASE.2011.38.

[LRS+94] Todd Little, M Ahsan Rahi, Craig Sinclair et al. « Buy Don't Build : What Does That Mean for a Software Developer ? » In : *Petroleum Computer Conference*. Society of Petroleum Engineers. 1994.

[LRZ17] Edward Lee, Jan Reineke et Michael Zimmer. « Abstract PRET machines ». In : *Real-Time Systems Symposium (RTSS)*. IEEE. 2017, p. 1-11. doi : 10.1109/RTSS.2017.00041.

[LS99a] Thomas Lundqvist et Per Stenstrom. « Timing anomalies in dynamically scheduled microprocessors ». In : *20th IEEE Real-Time Systems Symposium*. IEEE. 1999, p. 12-21. doi : 10.1109/REAL.1999.818824.

[LS99b] Thomas Lundqvist et Per Stenström. « An integrated path and timing analysis method based on cycle-level symbolic execution ». In : *Real-Time Systems* 17.2 (1999), p. 183-207. doi : 10.1023/A:1008138407139.

[Mai+17] Claire Maiza, Pascal Raymond, Catherine Parent-Vigouroux, Armelle Bonenfant, Fabienne Carrier, Hugues Cassé, Philippe Cuenot, Denis Claraz, Nicolas Halbwachs, Erwan Jahier, Hanbing Li, Marianne De Michiel, Vincent Mussot, Isabelle Puaut, Christine Rochange,

Erven Rohou, Jordy Ruiz, Pascal Sotin et Wei-Tsun Sun. « The W-SEPT project : Towards semantic-aware WCET estimation ». In : *17th International Workshop on Worst-Case Execution Time Analysis*. 2017. doi : 10.4230/OASIcs.WCET.2017.9.

[Mai+19] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer et Robert I Davis. « A survey of timing verification techniques for multi-core real-time systems ». In : *ACM Computing Surveys (CSUR)* 52.3 (2019), p. 1-38. doi : 10.1145/3323212.

[Man+13] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo et Rodolfo Pellizzoni. « Real-time cache management framework for multi-core architectures ». In : *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2013, p. 45-54. doi : 10.1109/RTAS.2013.6531078.

[Man+15] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha et Heechul Yun. « WCET (m) estimation in multi-core systems using single core equivalence ». In : *27th Euromicro Conference on Real-Time Systems*. IEEE. 2015, p. 174-183. doi : 10.1109/ECRTS.2015.23.

[Mas+21] Alfonso Mascareñas González, Frédéric Boniol, Youcef Bouchebaba, Jean-Loup Bussenot et Jean-Baptiste Chaudron. « Heterogeneous multicore SDRAM interference analysis ». In : *29th International Conference on Real-Time Networks and Systems (RTNS)*. 2021, p. 12-23. doi : 10.1145/3453417.3453426.

[Mas21] Maspa Technologies. *Microbenchmark Technology*. 2021. url : https://maspatechnologies.com/microbenchmark-technology/ (visité le 01/06/2021).

[MB68] Frederick H Martin et Richard H Battin. « Computer-controlled steering of the Apollo spacecraft ». In : *Journal of Spacecraft and Rockets* 5.4 (1968), p. 400-407.

[MIS19] MISRA. *MISRA C 2012 Guidelines for the use of the C language in critical systems*. 2019.

[Mit17] Sparsh Mittal. « A survey of techniques for cache partitioning in multicore processors ». In : *ACM Computing Surveys (CSUR)* 50.2 (2017), p. 1-39. doi : 10.1145/3062394.

[MOT20] Amira Methni, Emmanuel Ohayon et François Thurieau. « ASTERIOS Checker : A Verification Tool for Certifying Airborne Software ». In : *10th European Congress on Embedded Real Time Systems (ERTS)*. Toulouse, France, jan. 2020.

[Mue95] Frank Mueller. « Compiler support for software-based cache partitioning ». In : *ACM Sigplan Notices* 30.11 (1995), p. 125-133. doi : 10.1145/216636.216677.

[NHC12] Jonathan Nibert, Marc E Herniter et Zachariah Chambers. « Model-based system design for MIL, SIL, and HIL ». In : *World Electric Vehicle Journal* 5.4 (2012), p. 1121-1130.

[NP12] Jan Nowotsch et Michael Paulitsch. « Leveraging multi-core computing architectures in avionics ». In : *9th European Dependable Computing Conference*. IEEE. 2012, p. 132-143. doi : 10.1109/EDCC.2012.27.

[NPB19] Aina Niemetz, Mathias Preiner et Armin Biere. « Boolector at the SMT competition 2019 ». In : *17th International Workshop on Satisfiability Modulo Theories*. Sous la dir. de Joe Hendrix et Natasha Sharygina. 2019, p. 2.

[NS07] Nicholas Nethercote et Julian Seward. « Valgrind : a framework for heavyweight dynamic binary instrumentation ». In : *ACM Sigplan notices* 42.6 (2007), p. 89-100. doi : 10.1145/1250734.1250746.

[NXP11] NXP. *PowerPC e500 Core Family Reference Manual, Rev1*. Nov. 2011.

[NXP13] NXP. *e5500 Core Reference Manual, Rev4*. Mars 2013.

[NXP17] NXP. *MPC5777M Reference Manual, Revision 4.3*. Jan. 2017.

[Pag+14] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard et Pierre Siron. « The ROSACE case study : From Simulink specification to multi/many-core execution ». In : *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, p. 309-318. doi : 10.1109/RTAS.2014.6926012.

[Pal+20] Xavier Palomo, Sylvain Girbal, Jaume Abella Ferrer, Laurent Rioux, Mikel Fernández, Enrico Mezzetti et Francisco Javier Cazorla Almeida. « Tracing hardware monitors in the gr712rc multicore platform : Challenges and lessons learnt from a space case study ». In : *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020) : 7-10 July 2020 : proceedings*. T. 165. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2020, p. 15-1. doi : 10.4230/LIPIcs.ECRTS.2020.15.

[Pan+19] Maksim Panchenko, Rafael Auler, Bill Nell et Guilherme Ottoni. « Bolt : a practical binary optimizer for data centers and beyond ». In : *International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, p. 2-14. doi : 10.1109/CGO.2019.8661201.

[Pel+10] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo et Lothar Thiele. « Worst case delay analysis for memory interference in multicore systems ». In : *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2010, p. 741-746. doi : 10.1109/DATE.2010.5456952.

[Pel+11] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo et Russell Kegley. « A predictable execution model for COTS-based embedded systems ». In : *17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2011, p. 269-279. doi : 10.1109/RTAS.2011.33.

[Per+16] Quentin Perret, Pascal Maurere, Eric Noulard, Claire Pagetti, Pascal Sainrat et Benoit Triquet. « Temporal isolation of hard real-time applications on many-core processors ». In : *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2016, p. 1-11. doi : 10.1109/RTAS.2016.7461363.

[Per02] Ken Perlin. « Improving noise ». In : *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002, p. 681-682. doi : 10.1145/566654.566636.

[PF21] Sebastian Poeplau et Aurélien Francillon. « SymQEMU : Compilation-based symbolic execution for binaries ». In : *28th Annual Network and Distributed System Security Symposium (NDSS)*. Fév. 2021.

[PH90] Karl Pettis et Robert C Hansen. « Profile guided code positioning ». In : *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 1990, p. 16-27. doi : 10.1145/93542.93550.

[Pha+18] Guillaume Phavorin, Pascal Richard, Joël Goossens, Claire Maiza, Laurent George et Thomas Chapeaux. « Online and offline scheduling with cache-related preemption delays ». In : *Real-Time Systems* 54.3 (2018), p. 662-699. doi : 10.1007/s11241-017-9275-6.

[PKP09] Peter Puschner, Raimund Kirner et Robert G Pettit. « Towards composable timing for real-time programs ». In : *Software Technologies for Future Dependable Distributed Systems*. IEEE. 2009, p. 1-5. doi : 10.1109/STFSSD.2009.26.

[PR02] Erez Petrank et Dror Rawitz. « The hardness of cache conscious data placement ». In : *ACM SIGPLAN Notices* 37.1 (2002), p. 101-112. doi : 10.1145/503272.503283.

[Ray13] Michel Raynal. *Concurrent programming : algorithms, principles, and foundations*. Springer Science, 2013. isbn : 978-3-642-32026-2. doi : 10.1007/978-3-642-32027-9.

[Rei+06] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger et Bernd Becker. « A definition and classification of timing anomalies ». In : *6th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2006.

[RTC11a] RTCA DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*. 2011.

[RTC11b] RTCA DO-330. *Software Tool Qualification Considerations*. 2011.

[RTP18] D Radack, Harold G Tiedeman et P Parkinson. « Civil certification of multi-core processing systems in commercial avionics ». In : *27th Safety-critical Systems Symposium Proceedings*. 2018.

[RUS14] Christine Rochange, Sascha Uhrig et Pascal Sainrat. *Time-Predictable Architectures*. FOCUS - Computer Engineering Series. 2014. isbn : 978-1-84821-593-1.

[Sal74] Jerome H Saltzer. « Protection and the control of information sharing in Multics ». In : *Communications of the ACM* 17.7 (1974), p. 388-402. doi : 10.1145/361011.361067.

[Saw13] Zdeněk Sawa. « Efficient construction of semilinear representations of languages accepted by unary nondeterministic finite automata ». In : *Fundamenta Informaticae* 123.1 (2013), p. 97-106. doi : 10.3233/FI-2013-802.

[SBP19] Nathanaël Sensfelder, Julien Brunel et Claire Pagetti. « Modeling cache coherence to expose interference ». In : *31st Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019. doi : 10.4230/LIPIcs.ECRTS.2019.18.

[Smi82] Alan Jay Smith. « Cache memories ». In : *ACM Computing Surveys (CSUR)* 14.3 (1982), p. 473-530. doi : 10.1145/356887.356892.

[SS15] Florent Saudel et Jonathan Salwan. « Triton : A dynamic symbolic execution framework ». In : *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*. 2015, p. 31-54.

[Tab+19] Rohan Tabish, Renato Mancuso, Saud Wasly, Rodolfo Pellizzoni et Marco Caccamo. « A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems ». In : *Real-Time Systems* 55.4 (2019), p. 850-888. doi : 10.1007/s11241-019-09340-0.

[Ter+10] Dan Terpstra, Heike Jagode, Haihang You et Jack Dongarra. « Collecting performance data with PAPI-C ». In : *Tools for High Performance Computing*. Springer, 2010, p. 157-173. doi : 10.1007/978-3-642-11261-4_11.

[The04] Stephan Thesing. « Safe and precise WCET determination by abstract interpretation of pipeline models ». Thèse de doct. 2004.

[Thi+11] Philippe Thierry, Laurent George, Jean-Franois Hermant, Fabien Germain, Dominique Ragot et Jean-Marc Lacroix. « Toward a predictable and secure data cache algorithm : a cross-layer approach ». In : *10th International Symposium on Programming and Systems*. IEEE. 2011, p. 148-155.

[Tho17] Romain Thomas. *LIEF - Library to Instrument Executable Formats*. https ://lief.quarkslab.com/. Avr. 2017.

[TIS95] TIS Committee. *Executable and Linking Format (ELF) Specification*. Standard. Tool Interface Standard, mai 1995.

[Tom88] James E Tomayko. *Computers in spaceflight : The NASA experience*. Rapp. tech. NASA, 1988.

[Uhl19] Rich Uhlig. « From labs to impact ». In : *DARPA ERI Summit*. Intel Labs. 2019.

[Ung+10] Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quiñones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff et Jörg Mische. « MERASA : Multicore execution of hard real-time applications supporting analyzability ». In : *IEEE Micro* 30.5 (2010), p. 66-75. doi : 10.1109/MM.2010.78.

[Ung+16] Theo Ungerer, Christian Bradatsch, Martin Frieb, Florian Kluge, Jörg Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel G. Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panic, Jaume Abella, Carles Hernández, Francisco J. Cazorla, Sascha Uhrig, Mathias Rohde et Arthur Pyka. « Parallelizing industrial hard real-time applications for the parMERASA multicore ». In : *ACM Transactions on Embedded Computing Systems (TECS)* 15.3 (2016), p. 1-27. doi : 10.1145/2910589.

[VE20] Steven H. VanderLeest et Christos Evripidou. « An Approach to Verification of Interference Concerns for Multicore Systems (CAST-32A) ». In : *SAE Int. J. Adv. & Curr. Prac. in Mobility* 2 (mars 2020), p. 1174-1181. doi : 10.4271/2020-01-0016.

[Ves+12] Stephen C Vestal, Pamela Binns, Aaron Larson, Murali Rangarajan et Ryan Roffelsen. *Safe partition scheduling on multi-core processors*. US Patent 8,316,368. Nov. 2012.

[Vol59] Jack E Volder. « The CORDIC trigonometric computing technique ». In : *IRE Transactions on electronic computers* 3 (1959), p. 330-334. doi : 10.1109/TEC.1959.5222693.

[VT20] Steven H VanderLeest et Samuel R Thompson. « Measuring the Impact of Interference Channels on Multicore Avionics ». In : *AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*. IEEE. 2020, p. 1-8. doi : 10.1109/DASC50938.2020.9256647.

[War+13] Franck Wartel, Leonidas Kosmidis, Code Lo, Benoit Triquet, Eduardo Quiñones, Jaume Abella, Adriana Gogonel, Andrea Baldovin, Enrico Mezzetti, Liliana Cucu, Tullio Vardanega et Francisco J. Cazorla. « Measurement-based probabilistic timing analysis : Lessons from an integrated-modular avionics case study ». In : *8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2013, p. 241-248. doi : 10.1109/SIES.2013.6601497.

[Weg17] Simon Wegener. « Towards Multicore WCET Analysis ». In : *17th International Workshop on Worst-Case Execution Time Analysis (WCET)*. T. 57. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017, 7 :1-7 :12. doi : 10.4230/OASIcs.WCET.2017.7.

[WH09] James Windsor et Kjeld Hjortnaes. « Time and space partitioning in spacecraft avionics ». In : *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*. IEEE. 2009, p. 13-20. DOI : 10.1109/SMC-IT.2009.11.

[Whi+12] Jack Whitham, Robert I Davis, Neil C Audsley, Sebastian Altmeyer et Claire Maiza. « Investigation of scratchpad memory for preemptive multitasking ». In : *33rd Real-Time Systems Symposium*. IEEE. 2012, p. 3-13. DOI : 10.1109/RTSS.2012.54.

[Wil+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat et Per Stenström. « The worst-case execution-time problem—overview of methods and survey of tools ». In : *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), p. 1-53. DOI : 10.1145/1347375.1347389.

[Wol93] Andrew Wolfe. « Software-based cache partitioning for real-time applications ». In : *Third International Workshop on Responsive Computer Systems*. 1993.

[Yun+13] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo et Lui Sha. « Memguard : Memory bandwidth reservation system for efficient performance isolation in multi-core platforms ». In : *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2013, p. 55-64. DOI : 10.1109/RTAS.2013.6531079.

[Yun+14] Heechul Yun, Renato Mancuso, Zheng-Pei Wu et Rodolfo Pellizzoni. « PALLOC : DRAM bank-aware memory allocator for performance isolation on multicore platforms ». In : *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, p. 155-166. DOI : 10.1109/RTAS.2014.6925999.

[Yun+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang et Taesoo Kim. « QSYM : A practical concolic execution engine tailored for hybrid fuzzing ». In : *27th USENIX Security Symposium (USENIX Security 18)*. 2018, p. 745-761.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Analyse de systèmes temps-réels de sûreté et mitigation de leurs interférences temporelles

Mots clés : systèmes temps-réels de sûreté, analyse de programmes, interférences temporelles

Résumé : La mise en œuvre de systèmes temps-réels de sûreté requiert souvent l'élaboration de stratégies de provisionnement de temps ; cette pratique est particulièrement répandue dans le domaine industriel de l'avionique. Elles consistent à assigner des quotas de temps aux tâches qui composent le système, de sorte que chaque tâche dispose de suffisamment de temps d'exécution disponible pour compléter, tout en respectant des contraintes de cadence et de latence strictes. Les temps d'exécution dans les pires cas, ou WCET (Worst-Case Execution Times), sont particulièrement utilisés pour effectuer ce dimensionnement temporel. L'estimation des WCET est ainsi une étape cruciale du développement des systèmes temps-réels de sûreté. Leurs valeurs doivent être sûres afin de garantir la bonne exécution du système, sans être trop surestimées, pour que le système bénéficie d'un équilibre satisfaisant entre sûreté et efficacité. Une difficulté majeure dans la détermination des WCET réside dans l'apparition d'interférences temporelles à l'exécution, venant du fait que les architectures sur étagère actuelles reposent sur des fonctionnalités matérielles statistiquement efficaces (par exemple, les caches), qui réduisent les temps d'exécution moyens, mais dégradent certaines propriétés temporelles d'intérêt, nuisant à la caractérisation des systèmes temps-réels. Les WCET sont ainsi fortement accrus, affectant le dimensionnement temporel. Les travaux contenus dans cette thèse visent à analyser certains systèmes temps-réels de sûreté dans le but de mitiger les interférences temporelles auxquels ils sont sensibles. Face au constat que l'utilisation croissante de plateformes multicœurs engendre de nouvelles interférences temporelles causées par des accès simultanés à des ressources matérielles partagées, ces travaux cherchent à les prévenir par conception, afin d'éviter leur survenue à l'exécution. Les caches matériels sont également une source importante d'interférences temporelles ; le second volet de ces travaux est dédié à l'analyse de programmes compilés pour caractériser l'intégralité des accès mémoires qu'ils effectuent, et ce, afin de proposer des stratégies de placement mémoire permettant une meilleure utilisation des caches, au fait des pires cas.

Title: Analysis of safety-critical real-time systems and mitigation of their timing interferences

Keywords: safety-critical real-time systems, program analysis, timing interferences

Abstract: Safety-critical real-time systems often rely on time provisioning strategies. These are especially a standard practice in avionics. They consist in assigning quotas of time to the tasks that compose the system, such that each task is allotted with a sufficient amount of execution time to complete while respecting strict latency and pacing constraints. WCET (Worst-Case Execution Times) are frequently used as the foundation of this temporal partitioning. WCET estimation is therefore a crucial step in the development of safety-critical real-time systems. Their values must be sound to guarantee the nominal execution of the system, but they also must be tight, so the system can benefit from a satisfactory equilibrium between safety and efficiency. A major obstacle to WCET evaluation resides in the occurrence of timing interferences at run-time. They are caused by contemporary off-the-shelf architectures that rely on statistically efficient hardware features (for example, the caches), that greatly reduce average execution times, but degrade some timing properties of interest, which is detrimental to the characterization of real-time systems. As a result, WCET are often greatly overestimated, affecting temporal partitioning. The work presented in this thesis seek to analyze specific safety-critical real-time systems to mitigate the timing interferences they are the most sensitive with. The increasing use of multicore platforms leads to new classes of timing interferences: simultaneous accesses to shared hardware resources. This work aims at preventing them by design, effectively avoiding their occurrence at run-time. Hardware caches are also a well-known source of timing interferences. The second part of this thesis is dedicated to analyzing compiled programs to characterize the wholeness of their memory accesses. The end goal of this technique is to propose memory partitioning strategies allowing a better use of caches that encompasses the worst cases.