Sécurité temps réel dans les systèmes embarqués critiques

Pierrick Buret

UNIVERSITÉ DE LIMOGES
ÉCOLE DOCTORALE « Sciences et Ingénierie pour l'Information »
FACULTÉ DES SCIENCES ET TECHNIQUES

THÈSE

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ DE LIMOGES

Discipline / Spécialité : Informatique

présentée et soutenue par

Pierrick Buret

le 1 décembre 2015

Sécurité temps réel dans les systèmes embarqués critiques

Thèse dirigée par Julien Iguchi-Cartigny

JURY

Rapporteurs :

M. Frédéric Boniol	Professeur à Université de Toulouse
Mme. Vania Marangozova-Martin	HDR à Université de Grenoble Examinateurs:
Mme. Laetitia Jourdan	Professeur à INRIA Lille
M. Benoit Martin	Agent DGA habilité, Bruz
M. Sylvain Lecomte	Professeur à l'Université de Valenciennes
M. Gilles Grimaud	Professeur à l'Université de Lille
M. Louis Rilling	Agent DGA habilité, Rennes
M. Julien Iguchi-Cartigny	Maitre de Conférences (HDR) à l'Université de Limoges
M. Philippe Gaborit	Professeur à l'Université de Limoges

Remerciements

Je tiens en premier lieu à adresser mes sincères remerciements, ainsi que ma profonde gratitude aux docteurs Frédéric Boniol et Vania Marangozova-Martin, d'avoir trouver le temps de jouer leur rôle de rapporteur malgré un emploi du temps chargé.

Je remercie également particulièrement Laetitia Jourdan d'avoir accepté la lourde tâche de présidente du jury, ainsi qu'à Louis Rilling, Benoit Martin, Sylvain Lecomte, Philippe Gaborit et Gilles Grimaud d'avoir accepté d'être mes examinateurs.

Je tiens également à remercier mon directeur de thèse Julien Iguchi-Cartigny pour la confiance qu'il m'a accordée, pour ses conseils éclairés et avisés, ainsi que pour les «coups de fouet» qui m'ont permis d'avancer dans les moments difficiles. (Néanmoins la musique et le poisson japonais nuit gravement à votre santé)

Je remercie également ma fiancée Oriane de m'avoir soutenu dans les moments les plus difficiles (et les plus faciles...) ces dernières années. Je remercie également très fortement ma famille dont le soutien moral et logistique (en transport tout temps et tout terrain, gâteaux et autre ressources vitales à la réalisation d'une thèse) fut sans faille et sans pareille.

Mes remerciements vont également à mes adorables collègues (doctorants et ingénieurs), je cite : Amaury, David, Baguia, de Limoges et Francois, Damien, Greg, Nadir, Narjes, pour leur amitié et pour tous les bon moments de rire et de folie que nous avons passés ensemble. Je remercie encore une fois ceux d'entre eux qui ont eu la gentillesse de lire des parties de ce mémoire.

Je remercie aussi tous les autres doctorants : Christophe, Quentin, Valentin pour qui le trajet est encore long ; les anciens doctorants : et plus particulièrement au Docteur Nassima (maman de l'équipe SSD en son temps), Docteur Bouffard (roi de la carte à puce), Docteur Jeff (pour son sens du rythme et son combiwagen comme on en fait plus...), et d'autres ; les stagiaires passés et présents de notre équipe ainsi que le personnel technique et administratif en particulier Hubert, Sylvie et Odile, pour leurs amitié et sympathie.

Je n'oublie pas aussi de remercier mes amis de longue date, ainsi que ceux encore debout après moult combats comme bibi, les canonniers sédentaires de lille, et tout les personnels des services de l'état qui se reconnaîtrons. Un grand merci à tous ceux tombés dans l'honneur mais qui m'ont apportés des souvenirs et connaissances inoubliables.

Merci à tous ceux qui ont contribué de prêt ou de loin à l'aboutissement de cette thèse. (Entre autre les crevettes naines qui furent source inépuisable d'idées sur la génétique).

Résumé

La croissance des flux d'information à travers le monde est responsable d'une importante utilisation de systèmes embarqués temps-réel, et ce notoirement dans le domaine des satellites. La présence de ces systèmes est devenue indispensable pour la géolocalisation, la météorologie, ou les communications. La forte augmentation du volume de ces matériels, impactée par l'afflux de demande, est à l'origine de l'accroissement de la complexité de ces derniers. Grâce à l'évolution du matériel terrestre, le domaine aérospatial se tourne vers de nouvelles technologies telles que les caches, les multi-coeurs, et les hyperviseurs.

L'intégration de ces nouvelles technologies est en adéquation avec de nouveaux défis techniques. La nécessité d'améliorer les performances de ces systèmes induit le besoin de réduction du coût de fabrication et la diminution du temps de production. Les solutions technologiques qui en découlent apportent pour majeure partie des avantages en matière de diminution du nombre global de satellites à besoin constant. La densité d'information traitée est parallèlement accrue par l'augmentation du nombre d'exploitants pour chaque satellite. En effet, plusieurs clients peuvent se voir octroyer tout ou partie d'un même satellite.

Intégrer les produits de plusieurs clients sur une même plateforme embarquée la rend vulnérable. Augmenter la complexité du système rend dès lors possible un certain nombre d'actes malveillants. Cette problématique autrefois à l'état d'hypothèse devient aujourd'hui un sujet majeur dans le domaine de l'aérospatial.

Figure dans ce document, en premier travail d'exploration, une présentation des actes malveillants sur système embarqué, et en particulier ceux réalisés sur système satellitaire. Une fois le risque exposé, je développe la problématique temps-réel.

Je m'intéresse dans cette thèse plus précisément à la sécurité des hyperviseurs spatiaux. Je développe en particulier deux axes de recherche.

Le premier porte sur l'évolution des techniques de producion et la mise en place d'un système de contrôle des caractéristiques temporelles d'un satellite.

Le deuxième axe améliore les connaissances techniques sur un satellite en cours de fonctionnement et permet une prise de décision en cas d'acte malveillant. Je propose plus particulièrement une solution physique permettant de déceler une anomalie sur la gestion des mémoires internes au satellite. En effet, la mémoire est un composant essentiel du fonctionnement du système, et ses propriétés communes entre tous les clients la rend particulièrement vulnérable. De plus, connaître le nombre d'accès en mémoire permet un meilleur ordonnancement et une meilleure prédiction d'un système temps réel. Notre composant permet la détection et l'interprétation d'une potentielle attaque ou d'un problème de sûreté de fonctionnement.

Cette thèse met en évidence la complémentarité des deux travaux proposés. En effet, la mesure du nombre d'accès en mémoire peut se mesurer via un algorithme génétique dont la forme est équivalente au programme cherchant le pire temps d'exécution. Il est finalement possible d'étendre nos travaux de la première partie vers la seconde.

Mots clé : *Systèmes embarqués*, *Hyperviseur*, *Temps-réel*, *WCET*, *Contention mémoire*, *Algorithme génétique*, *Sécurité*.

Abstract

Satellites are real-time embedded systems and will be used more and more in the world. Become essential for the geo-location, meteorology or communications across the planet, these systems are increasingly in demand. Due to the influx of requests, the designers of these products are designing a more and more complex hardware and software part. Thanks to the evolution of terrestrial equipment, the aero-space field is turning to new technologies such as caches, multi-core, and hypervisor.

The integration of these new technologies bring new technical challenges. In effect, it is necessary to improve the performance of these systems by reducing the cost of manufacturing and the production time. One of the major advantages of these technologies is the possibility of reducing the overall number of satellites in space while increasing the number of operators. Multiple clients softwares may be together today in a same satellite.

The ability to integrate multiple customers on the same satellite, with the increasing complexity of the system, makes a number of malicious acts possible. These acts were once considered as hypothetical. Become a priority today, the study of the vulnerability of such systems become major.

In this paper, we present first work a quick exploration of the field of malicious acts on onboard system and more specifically those carried out on satellite system. Once the risk presentation we will develop some particular points, such as the problematic real-time.

In this thesis we are particularly interested in the security of space hypervisors. We will develop precisely 2 lines of research.

The first axis is focused on the development of production technics and implementing a control system of a satellite temporal characteristics. The objective is to adapt an existing system to the constraints of the new highly complex systems. We confront the difficulty of measuring the temporal characteristics running on a satellite system. For this we use an optimization method called dynamic analysis and genetic algorithm. Based on trends, it can automatically search for the worst execution time of a given function.

The second axis improves the technical knowledge on a satellite in operation and enables decision making in case of malicious act. We propose specifically a physical solution to detect anomalies in the management of internal memory to the satellite. Indeed, memory is an essential component of system operation, and these common properties between all clients makes them particularly vulnerable to malicious acts. Also, know the number of memory access enables better scheduling and better predictability of a real time system. Our component allows the detection and interpretation of a potential attack or dependability problem.

The work put in evidence the complementarity of the two proposed work. Indeed, the measure of the number of memory access that can be measured via a genetic algorithm whose shape is similar to the program seeking the worst execution time. So we can expand our work of the first part with the second.

Key words : *Embedded system*, *Hypervisor* , *Real-Time*, *WCET*, *Memory contention*, *Genetic algorithm*, *Security.*

Sommaire

Liste des tableaux

Table des figures

Introduction générale

1 Introduction

Ma thèse est axée sur le domaine aérospatial [Ver82]. Celui-ci, en forte expansion ces dernières années ne cesse d'évoluer technologiquement et de se confronter à de nouveaux problèmes. L'augmentation du nombre d'utilisateurs et la diminution du nombre de places en orbite terrestre disponible complexifient les systèmes. Autrefois réservés à quelques nations, les satellites se banalisent dans le monde industriel, plus précisément civil. Il n'est plus rare aujourd'hui d'envisager plusieurs clients sur un même satellite, ce qui permet le partage des coûts de conceptions, et d'exploitations.

Les études montrent qu'un satellite passe, en l'espace de 50 années d'évolution, de 10 ans de production à 3 ans pour un satellite de même capacité. Hier doté d'un processeur unique [esi15], les satellites évoluent vers des systèmes multi-processeurs. La capacité de calcul accrue des multi-processeurs et la possibilité de mettre plusieurs clients sur un même satellite [MRPC10] sont des avancées décisives vers la conquête de l'espace. Nous présentons par la suite les conséquences de ces choix.

Les satellites sont de nos jours, plus performants, rapides à produire et sûrs. Les temps de production se voient fortement réduits, tout comme le coût d'achat (réparti sur plusieurs clients) [VVPV08]. La production devient même généralisée et provoque un effet de masse. Ces évolutions amènent pourtant plusieurs difficultés. La nouvelle possibilité offerte aux clients de développer leurs propres codes sur un satellite en est un exemple. En effet, il est possible que ce client soit malveillant ou que ses exigences de sécurité ne soient pas la même que celle de l'industriel fabriquant le satellite. Il est donc possible de voir apparaître des attaques.

Les systèmes embarqués temps-réels tels que les satellites restent des noeuds de communication et de renseignement essentiels, aussi bien d'un point de vue militaire que civil. La non-sécurisation de ces vecteurs d'informations pourrait mettre à mal un pays complet. C'est pour cela que les efforts de recherche se portent sur ce domaine. Lors de la rédaction de cette thèse, j'ai souhaité privilégier l'usage de "nous" à l'usage de "je" car j'ai eu la chance de discuter avec de nombreux chercheurs et équipes sur le domaine. C'est également un (mon) style d'écriture permettant à mon avis une meilleur lisibilité.

2 Cadre de la thèse

Les travaux présentés dans cette thèse proposent un ensemble de solutions pour répondre aux nouvelles problématiques posées en matière de sécurité. Cette thèse a un intérêt double. Celle-ci

est co-financée entre une ANR et la DGA.

L'ANR se nomme SOBAS et a pour objectif de mettre en avant les défauts et problèmes de sécurité des avions de ligne et des satellites de production française (et par là-même européens). Cet ANR regroupe les partenaires suivants : ANSSI (Agence National de Securité des Systèmes d'Information) répondant directement au premier ministre et spécialisé dans la sécurité des systèmes de l'état ; EADS IW (aujourd'hui Airbus Defence and Space) spécialiste de la sécurité informatique au sein du groupe EADS (aujourd'hui groupe Airbus) ; Airbus (Conservant ce nom) qui conçoit la gamme des avions de ligne commerciaux les plus vendus au monde ; Astrium (Aujourd'hui regroupé sous Airbus Defence and Space) spécialiste de la conception des satellites civilo-militaires ; Le LAAS, goupe de recherche Toulousain spécialisé dans l'aéronautique et l'aviation commerciale ; XLIM, mon centre de recherche, situé à Limoges et spécialisé dans la sécurité des systèmes embarqués.

L'ANR SOBAS a pour objectif de révéler aux industriels et scientifiques les failles potentielles dans les systèmes embarqués produits et apporter des solutions techniques concrètes pour les résoudre.

A cela s'ajoute le co-financement de la DGA (Délégation Général de l'Armement) spécialiste de la recherche, de la vente, de l'achat et du traitement de l'ensemble du matériel militaire ou en lien avec celui-ci. La DGA intègre ce projet pour faire évoluer plus précisément les solutions aux attaques et peut ré-utiliser le matériel avancé dans la suite de cette thèse, par exemple, pour contrôler le matériel qui lui est fourni par les industriels.

Cette thèse est donc entièrement axée sur un cadre d'utilisation en milieu contraint. La sécurité y est primordiale et c'est cette vision que nous tentons de présenter à travers nos solutions.

Ce document est partagé en trois parties. La première développe les avancés des satellites dans le temps et leurs conséquences en terme de sécurité. Notre seconde contribution est l'analyse temporelle de l'exécution de fonctions dans les systèmes satellitaires modernes. La troisième contribution est la fourniture d'une solution permettant d'observer et gérer des problèmes d'accès à différents périphériques contenus dans les satellites.

2.1 Histoire et sécurité

Dans cette partie nous reprenons l'histoire des satellites dès l'origine. Nous développons plus particulièrement le développement technique de ceux-ci et les contraintes de fabrication de ces derniers. Puis nous observons les conséquences en terme de sûreté de fonctionnement et de sécurité sur ceux-ci. Cette partie présente également les évolutions technologiques et industrielles du domaine. A chaque évolution correspond un changement dans la politique de sûreté de fonctionnement et de sécurité face aux nouvelles menaces émergeantes que sont les actes malveillants à l'encontre des noeuds de communication que sont les satellites.

Contribution : Notre solution est basée sur une analyse des problèmes de sécurité au sein des satellites et sur l'analyse concrète de certaines menaces allant à l'encontre de ces systèmes. Dans ce chapitre, seules les menaces sont développées et non les solutions mises en place pour s'en protéger, bien que des pistes soient fournies. L'ensemble de ces expérimentations sont faites sur plate-forme spatiale réelle.

2.2 Recherche du pire temps d'exécution

Notre contribution majeure développe, dans le contexte présenté dans la première partie, un changement dans l'industrialisation amenant à des risques de sûreté ou de sécurité. Pour s'en prémunir, nous amenons l'idée de rechercher le pire temps d'exécution des différentes fonctions s'exécutant sur le système. Cette recherche, déjà développée pour les anciens systèmes satellitaires se confronte aux problématiques des nouvelles technologies. En effet, il n'existe pas de solution analytique simple permettant l'estimation du pire temps d'exécution sur un système aussi complexe que ceux utilisés aujourd'hui. Il existe trop de dépendance entre les différents systèmes d'un même satellite.

Contribution : Notre contribution est l'apport d'une évolution d'un algorithme dynamique déjà existant. La solution apportée prend en compte de manière automatique la complexité du système, puis recherche de lui même, sur plate-forme réelle, la valeur optimale mesurable. Notre algorithme génétique y est évalué sur système réel et comparé.

2.3 Problématique de la contention mémoire

Cette dernière partie met en oeuvre une problématique déjà connue du monde scientifique. Le problème de la contention. Nous développons dans cette partie la problématique de la contention mémoire sur système multi-coeur. Nous présentons le problème dans le monde satellitaire, bien que semblable dans les autres domaines. Puis nous développons une solution technique permettant la détection du phénomène et la remontée de l'information au système pouvant gérer celui-ci.

Contribution : Notre contribution dans cette partie est la conception d'un composant électronique permettant le décompte du nombre d'accès en mémoire et la détection d'une sur-utilisation par un des processeurs du satellite. Le dépassement y est ensuite envoyé au gestionnaire du système. En l'occurrence dans notre satellite, ce rôle est réalisé par un hyperviseur. Notre apport est complété par un ensemble de propositions de gestion de la contention mémoire. Nous montrons également l'intérêt de l'algorithme génétique développé dans la contribution précédente et parfaitement applicable dans ce cas précis.

3 Organisation du document

L'ensemble de ce document est organisé en trois parties et contient en tout 11 chapitres. La première partie de ma thèse développe l'histoire du domaine spatial et les contraintes qui lui sont propres. Ce chapitre amène des problématiques à résoudre à l'aide d'un état de l'art du domaine.

— Le chapitre 1 rappelle les anciennes techniques de fabrication des satellites ainsi que leur histoire. Nous présentons également dans ce chapitre les impacts en terme de sûreté de fonctionnement et les limites de ces premiers systèmes.
— Le chapitre 2 développe la conception moderne des satellites et les changements dans la fabrication. Nous mettons en avant une technologie particulière que nous utilisons dans cette thèse. Les hyperviseurs que nous comparerons entre eux.

— Le chapitre 3 reprend l'analyse de la sûreté de fonctionnement et y ajoute celle de la sécurité. Nous développons plus particulièrement dans ce chapitre les techniques d'actes malveillants réalisables sur satellites et offrons un exemple concret.

La seconde partie de notre thèse développe une technique de mesure du pire temps d'exécution dans un système aussi complexe que les satellites modernes. Le choix de mesure de ce paramètre est en lien direct avec le chapitre 3, développant les conséquences en terme de sûreté de fonctionnement et sécurité.

— Le chapitre 4 développe les raisons du choix de la mesure du temps sur de tels systèmes. Un état de l'art du domaine et des différentes techniques applicables y est développé, suivi de la justification de notre orientation.
— Le chapitre 5 modélise le problème de manière plus formelle. Il met en avant l'ensemble des hypothèses prises en compte et des contraintes du système étudié. Nous y développons le principe de développement de l'algorithme génétique suivi de deux cas d'applications de celui-ci. L'un sans notre apport, et l'autre développant notre idée.
— Le chapitre 6 est le coeur de notre recherche sur le pire temps d'exécution. Nous y développons un cas d'étude qui n'était pas pris en compte par les autres algorithmes génétiques. En effet, ils ne prenaient pas en compte notre apport majeur qu'est l'utilisation du contexte du système. Puis nous présentons la plate-forme étudiée ainsi que les premiers résultats obtenus sur celle-ci.
— Le chapitre 7 est réservé aux expérimentations et à la validation sur un grand nombre d'échantillons de notre apport avec notre algorithme génétique. Y est développé l'ensemble des fonctions étudiées et leurs résultats y sont commentés.
— Le chapitre 8 conclut nos travaux et ouvre la voie à de nouveaux axes de recherche permettant d'améliorer notre apport. Ce chapitre conclut la seconde partie.

La troisième et dernière partie de notre thèse étudie une contrainte connue mais non encore résolue et tente d'y apporter une solution. Le problème observé est celui de la contention et l'objectif de notre apport est sa détection.

— Le chapitre 9 présente le problème de la contention et plus particulièrement de la contention mémoire dans un état de l'art. La problématique y est soulevée et plusieurs propositions de solutions y sont comparées. Une solution de détection est proposée.
— Le chapitre 10 développe notre solution de détection, à savoir, un composant électronique observant le phénomène et amenant à résoudre le problème de la contention. Ce chapitre contient les expérimentations et résultats réalisés sur plate-forme réelle.
— Le chapitre 11 offre des pistes afin de gérer les informations fournies par notre composant. Plus particulièrement, nous proposons des politiques de gestion de la contention à l'aide de notre apport. Nous développons également l'intérêt particulier qu'offre notre seconde partie de la thèse (les algorithmes génétiques) dans cet autre domaine. Nous mettons en avant la capacité de nos recherches dans la seconde partie de la thèse à être utilisées de la même manière dans cette troisième partie. Ce chapitre conclut également la troisième partie de cette thèse.

Nous finissons ce rapport par une conclusion générale, présentée dans une partie distincte, et de quelques perspectives pour des travaux futurs.

Première partie

Domaine d'étude

Chapitre 1

Historique : Un constructeur, un client et un système bare-métal

Sommaire

Dans ce chapitre sont développées les connaissances spécifiques nécessaires à la compréhension du domaine d'étude, à savoir, l'aérospatial. Ce chapitre permet d'appréhender l'ensemble des contraintes qui ont amené la technologie spatiale au niveau actuel, et plus particulièrement en termes de sûreté de fonctionnement des systèmes embarqués temps-réels.

1.1 Introduction

L'Homme a toujours cherché à explorer l'inconnu, et c'est naturellement ce qu'il a fait dans les domaines aéronautiques et spatiaux. L'amélioration des systèmes d'exploration et d'exploitation de ces milieux, amènent de nouveaux défis technologiques et scientifiques. L'espace est un monde vaste que l'Homme explore depuis peu avec des satellites artificiels. Ainsi, le premier satellite, appelé Spoutnik 1, est lancé le 4 octobre 1957 [Var02]. Loin des technologies actuelles, il démontre la faisabilité du projet et lance la course à l'espace. En 3 ans et demi, 115 satellites sont en orbite [Ver82], et en 1965, la France envoie son premier satellite nommé Asterix [KHQ66].

En 2007, nous comptions déjà 5 500 satellites mis en orbite dont plus de 1 000 en fonctionnement [VVPV08]. De nos jours, les satellites jouent un rôle prépondérant sur les plans suivants :

— scientifique à partir de 1957 (observation astronomique, observation de la Terre et du système solaire, altimétrie et océanographie, études en apesanteur) ;
— militaire à partir de 1960 (renseignement optique, électromagnétique, etc...) ;
— économique à partir de 1972 (télécommunications et télévision, positionnement, prévision météorologique).

Ce chapitre présente les points communs entre tous les satellites de 1957 (Spoutnik 1) à 2000 en passant par le télescope spatial Hubble de 1990 [FMG+01].

Les satellites sont développés via un constructeur unique pour une mission spécifique et développé pour un client unique [BCE+78]. Historiquement, cela s'explique par le prisme stratégique de ces systèmes. Les clients étaient, durant la guerre froide, les États eux-même. Les pertes de satellites durant la phase exploratoire de l'espace étaient compensées par le nombre de satellites envoyés en orbite [Ver82]. La vitesse de conception et d'exploration de l'espace prévalait sur les performances. Seule les sociétés agréées de chaque état pouvaient prétendre à la conception d'un satellite [Sta85]. Aujourd'hui, la conception reste fortement réglementée comme en témoigne la loi *2012-304 du 6 mars 2012* avec son Décret *2013-700 du 30 juillet 2013*, rappelant le potentiel militaire de tout satellite et donc la nécessité de contrôle par l'état de la conception de ceux-ci.

Au fur et à mesure, les connaissances limitées du domaine spatial et des contraintes liées ont fait place à des systèmes prônant la performance. Dès les années 1990, les sociétés comprennent mieux les spécificités physiques de ce milieu mais il reste un schéma historique de développement d'un satellite spécifique pour chaque mission et chaque client [GE02]. L'ouverture sur le marché civil permet d'envisager des satellites plus robustes, avec des applications telles que la télévision [Par05]. La stratégie industrielle pour faire face à ces nouvelles demandes fut, dans un premier temps d'améliorer les performances des satellites et de faire grossir la taille et donc la complexité de ceux-ci. Un parfait exemple est l'ensemble des évolutions apportées à l'alimentation du satellite CHAMP par la société Astrium dans les années 2000 [KIE02]. La première contrainte des satellites est donc la capacité d'évolution de la technique dans le milieu contraint de l'espace. Par définition, un satellite est un système embarqué.

1.2 La contrainte embarquée

Un système embarqué est un système en auto-suffisance et autonomie énergétique en matière de fourniture et de gestion du courant électrique [VG02]. La taille du satellite dépend de sa mission, sa complexité, et son déploiement. Par déploiement, nous entendons le terme de lancement du satellite pour une mise en orbite. C'est la première contrainte technologique qui détermine les caractéristiques techniques d'un satellite [DC98]. Historiquement, les lanceurs étaient de faible capacité en taille et en poids. Le lanceur Diamant qui a permis en 1965 de lancer le satellite Asterix A1 de 39Kg avait une capacité de 150Kg maximum pour une mise en orbite en couche basse de l'atmosphère (200 Km) [Ngu01]. Les conquêtes spatiales permirent d'arriver à des lanceurs de type Ariane 5 avec une capacité avérée de 18 Tonnes en orbite basse et 10 Tonnes de charge utile en position géostationnaire (36 000 Km) [MW84].

L'une des caractéristiques définissant un satellite est donc avant tout sa masse, chaque gramme supplémentaire ayant un coût prohibitif. Cette volonté d'allègement des satellites à l'aire des technologies analogiques nécessitera l'optimisation des composants électroniques utilisés. Cette optimi-

sation a un coût en termes de développement et de temps très important. Le développement du satellite Hubble a pris 20 ans (de 1970 à 1990), la moyenne du temps de production et développement étant de 10 ans [Lau11].

Lorsque les technologies spatiales étaient encore analogiques, l'architecture matérielle de chaque satellite était donc spécifiquement développée pour un satellite unique et un client unique. Il n'existait pas encore de normalisation dans le domaine, chaque pays choisissant une norme nationale avantageuse pour ses propres entreprises.

Les dimensions des satellites amenèrent à la même conclusion. Il fut primordial de gagner de la place, et donc de complexifier les systèmes embarqués tout en en minimisant la taille. L'exemple le plus connu est l'utilisation des panneaux solaires qui doivent permettre l'alimentation en énergie du satellite tout en étant repliés pour entrer dans le lanceur. Ils sont dépliés lors de la mise en orbite avec tout les risques associés à cette manœuvre [FZBW10].

L'environnement spatial est particulièrement rigoureux d'un point de vue physique. En effet, outre les contraintes liées aux vibrations générées par le lanceur (vibration >9G), les composants électroniques doivent résister à des gammes exceptionnelles de températures avec de fortes fluctuations de celles-ci (pouvant aller jusqu'à 200 degrés Celsius). Cette différence de température s'explique par l'exposition ou non au soleil qui varie selon les mouvements du satellite dans l'espace [TH03], ce qui inciterait à réduire la taille de ceux-ci pour en limiter les parties exposées. Les composants doivent pourtant être suffisamment grands pour résister au bombardement du rayonnement généré par les éruptions solaires [GE02].

L'ensemble de ces contraintes embarquées est d'autant plus difficile à gérer qu'il est impossible d'intervenir dans une phase de maintenance d'un satellite déjà en orbite. Seul Hubble a fait l'objet d'une intervention humaine post-déploiement. Cette intervention a eu des coûts difficilement envisageables aujourd'hui [BER97].

1.3 La contrainte temps-réel

La terre tourne sur elle-même. Certains satellites sont dit géostationnaires. Ce sont ceux qui tournent à la même vitesse que la planète autour de laquelle ils sont en orbite. Ils ne sont néanmoins pas tous géostationnaires, c'est-à-dire que leur vitesse autour de la terre est non nulle. En d'autres termes, ils peuvent donc, lorsqu'ils se trouvent en dessous de la zone géostationnaire, passer plusieurs fois au dessus d'un même point terrestre. Dans ce cas de figure, il n'est pas envisageable de rater l'échéance où le satellite passe au-dessus d'une station sol qui communique avec lui pour échanger données et instructions. Un exemple de la complexité à communiquer avec des satellites extrêmement rapides vis-à-vis de la terre est détaillé dans [IRS+04]. Cela montre la difficulté du problème de synchronisation. Il faut donc garantir que chaque tâche applicative du satellite réponde en un temps donné garantissant son transfert d'information avec les systèmes terrestres. Cette tâche est prédictible et dite périodique car elle se répète régulièrement dans le temps à des dates connues et déterminées à l'avance. Le respect des contraintes temporelles dans l'exécution des tâches applicatives est aussi important que le résultat de celles-ci.

Cette contrainte temps-réel est d'autant plus forte aujourd'hui qu'il existe un grand nombre de débris et satellites divers et proches les uns des autres. Ils sont incontrôlables et appelés communément objets fantômes ou déchets [Lau11]. Outre la nécessité de blinder les satellites pour résister

à des collisions inévitables avec de petits objets à grande vitesse, il faut être en mesure de se déplacer sur son orbite pour en éviter les plus gros. Ainsi la station spatiale internationale change régulièrement d'orbite pour y éviter des satellites et autre "encombrants" de l'espace. C'est une contrainte temps-réel qui amène un côté apériodique au phénomène. En effet, malgré des moyens mis en œuvre, il est très difficile de repérer l'ensemble des débris dans l'espace. Calculer la trajectoire, la vitesse ou la position de tous les débris est donc inenvisageable. Les modifications de trajectoires sont donc calculées dynamiquement, au gré des détections, durant la vie du satellite en orbite. C'est donc le cas de la station spatiale international ISS (*International Space Station*), qui se déplace à l'aide des systèmes ATV afin d'éviter des nuages de débris ou de gros satellites [PS09].

Un nouvel accord international demande également à faciliter la destruction des satellites anciens par déplacement de ceux-ci en direction de l'atmosphère terrestre ou de l'espace lointain de manière contrôlé. Cette contrainte temps-réel est quant à elle fixée et prévue avant lancement [Lau11].

De plus, le satellite doit, par normalisation internationale [Smi68], être dans la capacité de répondre à des transmissions sol afin de pouvoir, le cas échéant, modifier son comportement potentiellement gênant. Ce que nous entendons par comportement gênant est par exemple un satellite fou émettant sur de nombreuses fréquences différentes des signaux aléatoires, perturbant ainsi les autres satellites. L'industriel doit donc garantir la sureté de fonctionnement de son application de transmission temps-réel.

Certaines parties du satellite sont considérées comme critiques. Elles subissent de fortes contraintes. Le niveau critique de ces contraintes dépend entièrement de la mission finale du satellite. Ainsi un système militaire à courte durée de vie devra être en mesure de se déplacer rapidement sur son orbite pour observer une situation inattendue et communiquer le plus rapidement possible avec les systèmes sol. Un satellite civil à longue durée de vie devra, quant à lui, être plus tolérant aux radiations spatiales, et aux phénomènes naturels, etc... . Au niveau temps-réel, cela se traduit par un ensemble de propriétés garantissant le bon fonctionnement des composants électroniques. Outre une horloge interne très précise calibrée en prenant en compte la position du satellite selon de la position des étoiles, il faut un ordonnancement adapté aux contraintes. Le positionnement d'un satellite est déterminé par capteur stellaire, prenant une photographie de l'espace et la comparant à une base de donnée, ainsi que par Gyromètre et Accéléromètre car il n'y a pas de GPS dans l'espace (sauf en orbite basse) [VZS98].

1.4 L'ordonnanceur

L'ordonnanceur d'un système temps-réel critique est l'application qui gère un ensemble de tâches en fonction d'une politique d'ordonnancement. Une tâche est définie comme un ensemble de code permettant de réalisé une fonction précise. Comme par exemple, envoi d'une transmission, prise d'une photographie, pivotement d'une antenne, etc. La politique d'ordonnancement est, quant à elle, définie comme étend la stratégie d'ordonnancement permettant de garantir que l'ensemble des tâches dites critiques seront exécutées complètement dans le temps qui leur est alloué [Bur91].

Ainsi, une tâche à une date de début, de fin, ainsi qu'une durée et une période de répétition (dans le cas des tâches dites périodiques ou répétitives), la durée devant être inférieure ou égale à la période de la tâche (date de fin - date de début). L'ordonnanceur devra gérer l'ensemble des

tâches afin de garantir l'accomplissement de celles-ci dans leurs périodes respectives.

Historiquement, les satellites étant analogiques, le code informatique reste limité. A des fins d'optimisation, l'ordonnanceur est dit statique. Un ordonnanceur statique a une politique d'ordonnancement fixe déterminée avant le lancement du satellite et qui ne bougera pas durant toute la vie de celui-ci. Cela implique également qu'il n'est pas envisageable de reconfigurer de nouvelles tâches durant la vie du satellite en orbite [PG01].

Plus récemment, afin de pouvoir observer des phénomènes scientifiques rares, il existe une possibilité de changer de plan d'ordonnancement durant le vol. C'est-à-dire qu'il est possible de garder éteins ou inactifs certains capteurs d'observations, qui, au vu d'un phénomène inattendu peut être activé pour collecter un maximum de données. La politique d'ordonnancement restera tout-de-même statique. C'est-à-dire que l'ensemble des scénarios seront envisagés au sol lors du développement du satellite, et des plans d'ordonnancements seront mis en place pour garantir à tout moment les tâches critiques, quel que soit le plan d'ordonnancement suivi. L'objectif est de garantir qu'une photographie, par exemple, ne sera pas prise trop tôt ou trop tard et rater ainsi le phénomène à observer [CFLH06].

Anciennement, le système reposait sur l'électronique et non le logiciel. La partie logicielle est dite bare-métal [SGGS98]. C'est-à-dire que le code logiciel est directement situé au dessus de la partie matérielle et fonctionne sans aucune installation préalable (système d'exploitation sous-jacent). La figure 1.1 montre une vision simpliste d'un satellite des années 1990. Le code machine ou assembleur fut le premier utilisé, suivie par le langage de programmation C. Les langages de programmation utilisés sont bas-niveau (langages impératifs). L'objectif de ce code est d'être le plus simple, le plus proche du matériel et le plus performant possible.

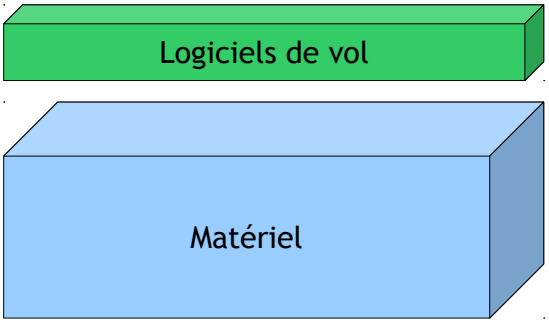

Fig. 1.1 – Bare-métal

1.5 Les propriétés de sécurité = sûreté

La sécurité était, dans un premier temps, définie comme la garantie du bon fonctionnement d'un système dans la vie "normal" de celui-ci. Ce que nous entendons par "normale", est la prise en compte de l'ensemble des influences extérieurs [GE02] pour garantir un taux de performance et de disponibilité acceptable pour le client. Par exemple, l'influence des radiations et autres perturba-

tions idoines, étaient considérées comme de la sécurité. Cela avait amené à doubler les composants électroniques pour obtenir la redondance des systèmes et des calculs [Mon99].

De nos jours, cette définition a fortement évolué. Les politiques de sécurité de l'époque qui se limitaient uniquement à la performance et à la fiabilité du satellite, s'apparentaient uniquement à de la sûreté de fonctionnement. Or, la sécurité des systèmes, et plus particulièrement la sécurité des systèmes embarqués, est définie comme un ensemble de moyen de détection ou de contre-mesure mis en place contre un acte malveillant [KLM+04]. L'acte malveillant comprend une volonté humaine de nuire au bon fonctionnement du système, ce qui n'est pas considéré dans le cas de la sûreté de fonctionnement.

Dans ce document, il a été choisi de comparer 3 petits satellites pour des raisons pratiques. Les satellites SARA, ANAIS et CUTE-I sont un sous-groupement particulier des satellites, appelés cubesat. De petites taille, ils remplissent une mission unique. Bien que ceux-ci ne soient conçu dans des objectifs de généricité ou de multi-client, tel que discuté dans la suite de la thèse, cela ne retire en rien les observations faites sur les plus gros satellites. En effet, historiquement, l'electronique prédominait au logiciel, et les évolutions internes et temps de productions ce sont avérés identiques. La comparaison aux systèmes présentés par la suite (satellites d'un poids pouvant aller a 800Kg) devant rester sur ces points communs spécifiques.

1.5.1 Plate-forme physique et transmission

Nous observons dans ce chapitre la sûreté de fonctionnement mise en place dans les différentes parties du satellite en prenant en exemple un satellite de 1991. Nous étudierons la sécurité dans le chapitre suivant.

La sûreté de fonctionnement d'un satellite nécessite, d'un point de vue physique, de prendre en compte l'ensemble des contraintes nommées au-dessus et bien d'autres encore [GE02]. Ainsi, nous pouvons résumer les architectures système comme étant robustes, résistantes aux vibrations et variations de chaleurs, mais aussi redondantes en composants et calculs. Toute la complexité du satellite se trouvant, à l'époque de la technologie analogique, dans cette partie physique. L'optimisation de cette partie, afin d'en réduire la taille et le poids, créait des systèmes complexes où le code, fortement limité et optimisé, était directement exécutable sans aucune installation préalable d'un système ou d'un logiciel pour son fonctionnement. Le système, dit "bare-métal" [SGGS98], était souvent utilisé comme en atteste les plans 1.2 du premier satellite amateur français SARA lancé le 17 juillet 1991 par une Ariane IV à Kourou (Guyane Française). Ces plans internes du satellite SARA proviennent de la documentation technique de GEMINISAT et Esieespace, associations réalisant des satellites amateurs depuis 1980.

Le système physique du satellite amateur est quasi-exclusivement analogique. Robuste de par ces redondances et son code informatique très limité, SARA est efficace en termes de disponibilité et de sûreté de fonctionnement. A aucun moment le satellite ne fut produit pour résister à une tentative d'action malveillante. Une copie de ce satellite est visible à Paris et l'ensemble des informations techniques proviennent des documentations relatives à celui-ci. Une image du satellite montre sa simplicité et sa robustesse 1.3.

L'architecture étant fixée, et uniquement réalisée à l'aide de composants physiques, il serait fortement complexe d'y introduire une action malveillante, si ce n'est par perturbation d'un composant électronique trop sensible ou défaillant [DN09]. La seule voix d'entrée d'un satellite comme

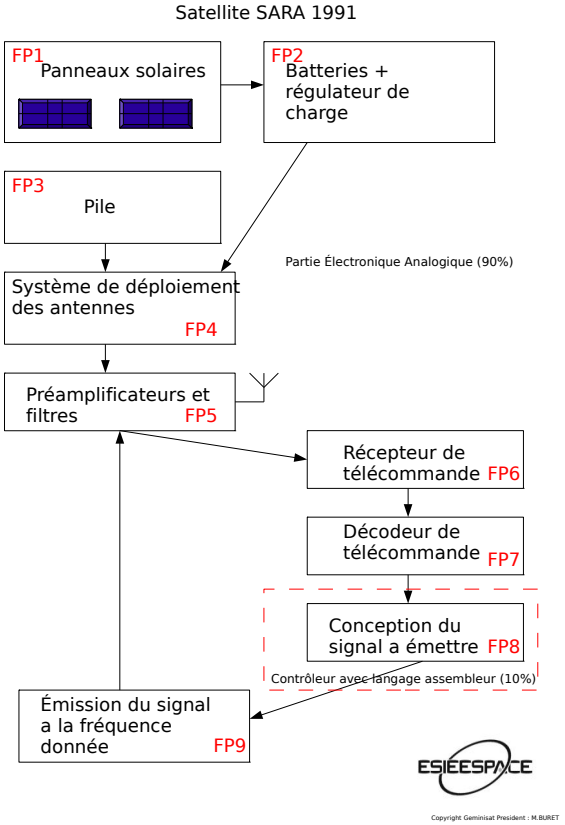

Fig. 1.2 – Plan interne du satellite SARA

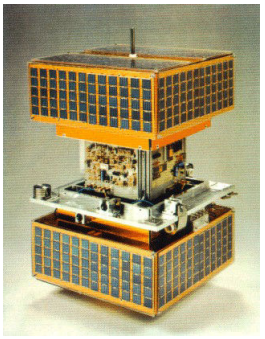

Fig. 1.3 – Photo du satellite SARA

celui-ci est la chaîne de transmission. En effet, en orbite autour de la terre, il n'est pas possible, à ce jour, de récupérer, modifier sur place, ou influencer par laser les composants électroniques comme nous le ferions sur terre [BICL11].

La chaîne de transmission est donc le point faible de ce type de système [Haw91]. Il peut être influencé par un système sol, c'est-à-dire une station spécialisée dans la communication par satellite, avec de fausses informations qui perturberaient le cycle "normal" du satellite. Les perturbations pourraient également venir de phénomènes naturels car les antennes captent dans leurs bandes

de fréquences tout ce qui passe. Il était donc déjà normalisé de crypter ou coder les messages transmis [Mar78]. Ainsi nous observons un décodeur dans le bloc de transmission du projet SARA figure 1.2. Bien que non mis en place pour lutter contre des actes malveillants, c'est la seule partie dite de sécurité.

1.5.2 Logiciel

La partie logicielle du satellite SARA se limite à la partie récupération des données de ses capteurs 1.2. En l'occurrence, ce satellite scientifique devait écouter Jupiter dans les fréquences de 10MHz à 100MHz. Puis le satellite renvoyait les données sur terre en haute fréquence.

Contrairement aux satellites industriels de la même époque, le satellite n'a pas de redondance dans les calculs, ceux-ci étant trop faibles pour justifier un tel coût en termes de composant supplémentaire. Le logiciel est donc très limité (10% figure 1.2) et très difficile à mettre en défaut, surtout de manière malveillante. Il faudrait de toute façon perturber la chaîne de transmission. Cette vision basique du système montre la complexité d'un petit système comme celui-ci. Étant entièrement développée bare-métal, il est donc dur à modifier et à adapter à un autre satellite. Son portage nécessiterait le développement complet d'un nouveau satellite pour répondre à une nouvelle mission.

C'est le cas du projet ANAIS (Analyse Numérique Amateur de la Ionosphère par Satellite) réalisé par l'association GEMINISAT qui succéda à l'association précédente pour le développement de micro-satellites en 2009. Son architecture a été entièrement revue pour répondre à une problématique quasi-identique techniquement à celle du premier micro-satellite. Le poids de l'ancien satellite (<50Kg) était trop lourd pour répondre aux contraintes actuelles des micro-satellites (<1.5Kg) [BDL10], et son électronique analogique ne pouvait être adapté au nouveau sujet (Analyse de la ionosphère de 1MHz à 50Mhz). Cela est d'autant plus complexe que chaque constructeur a sa propre politique de développement.

Outre la conception de la partie physique d'un satellite en salle blanche (où le taux de poussière est fortement contrôlé), il en va de même pour le logiciel. Celui-ci est implémenté et testé dans une zone sécurisée afin qu'aucune erreur humaine (ou acte malveillant) ne puisse intervenir durant la phase de conception. Cette maîtrise de l'ensemble de la fabrication est facilitée par l'existence d'un seul constructeur par satellite produit. Pour le satellite SARA, la phase de conception a duré 10 ans.

Dans ce satellite, le code est compact, du à une électronique simple, robuste et limitée à des composants qualifiés militaire durci. L'optimisation du code s'explique par l'obligation d'une mémoire de faible dimension. Le code est donc peu flexible, non évolutif, fortement limité et contraint par la partie physique. Le logiciel est néanmoins contraint de s'exécuter dans un intervalle de temps donné.

1.5.3 Temporel

Le satellite a besoin de respecter des contraintes temporelles. Cela se traduit, pour la sûreté de fonctionnement, par une surveillance forte des performances du satellite. De nombreux tests sont réalisés sur l'ordonnanceur afin de garantir mathématiquement et physiquement son bon fonctionnement [FSLM12]. C'est également le cas des systèmes sol [GL01]. Les contraintes étant très forte,

il y a d'importantes optimisations sur cette partie du développement. Il ne serait pas envisageable de rater une communication entre le système sol et le satellite. Dans le satellite SARA 1.3, cela ce traduit par un composant physique. Une horloge très précise, permet la synchronisation de tâches. Ce principe rustique, impossible à changer en vol, est basé sur la qualité du lanceur et sa capacité à mettre correctement le satellite en orbite sur le bon axe et à la bonne vitesse. Nous voyons ici l'ensemble des contraintes techniques remis à la charge et à la responsabilité du lanceur.

Les attaques temporelles sur de tels systèmes sont très difficiles car les systèmes sont fortement surveillés, robustes, à l'ordonnancement statique et de conception simple. Comme il est impossible de changer l'ordonnancement en vol, nous serions tenté de le faire au sol, mais les optimisations et multiples vérifications ainsi que l'absence de sous-traitant et de pertes d'informations lors du développement rendent la manœuvre difficile et hasardeuse [Kan73].

1.6 Conclusion

Les premiers satellites ouvrent une voie sur la complexité technique du domaine spatial et les problèmes de sûreté de fonctionnement. Bien qu'ayant l'avantage de la robustesse et de la simplicité d'un point de vue logiciel et temporel, les satellites avaient une complexité physique très importante. Celle-ci les caractérisent comme résistants, par défaut, aux attaques malveillantes, mais les limite à leur seule tâche initiale. Cela les rends spécifiques et non flexibles. Leur coût de développement ainsi que le temps de production sont très importants. Il n'est pas possible de développer un tel satellite en collaboration avec plusieurs entreprises car la maîtrise d'œuvre doit être complète. Un seul client pouvait en commander la production. Il existe néanmoins des redistributions des données apportées par le satellite, mais cette phase existe au sol.

Seules les stations sol et les transmissions sont protéger. Les stations étaient peu nombreuses et principalement militaire, les actions malveillantes se limitant aux défauts exploitables des composants. Dans le chapitre suivant nous allons aborder les nouveaux problèmes liés à la généralisation de la production des satellites et à la volonté de production de masse, ainsi qu'à l'ouverture du marché au domaine publique. Puis nous observerons les conséquences en termes de sécurité.

Chapitre 2

De nos jours : Des sous-traitants, des clients et un hyperviseur

Sommaire

Dans ce chapitre nous présentons l'évolution des satellites et du monde aérospatial avec l'arrivée des standardisations et de la production de masse. Nous développons son influence sur le dessein général du système et son impact sur la sûreté de fonctionnement. Puis nous développerons particulièrement le point de vue de la sécurité face aux attaques malveillantes dans le chapitre suivant. En conclusion, nous reprenons une attaque que nous avons réalisé sur un système standardisé aérospatial afin de démontrer ses faiblesses potentielles.

2.1 Introduction

De nos jours, les satellites se complexifient au niveau logiciel et se standardisent côté électronique. En effet, le monde évolue rapidement et les politiques des entreprises sont bouleversées. La demande en satellite sur les marchés civiles et militaires explose. L'ensemble des pays émergents souhaite acquérir ce type de système afin d'être indépendant des grandes puissances de la guerre froide. Nous pouvons observer, par exemple, la course aux satellites de géo-localisations qui, historiquement, étaient basés sur une technologie américaine (le GPS) [MUL02], et qui aujourd'hui

existent indépendamment pour chaque partie du monde. Ainsi l'Europe s'est doté de la constellation Galiléo [BDG+00], les Chinois de la constellation Beidou (ou COMPASS) [BJF05], et les Russes de la constellation GLONASS [PKI+02].

L'augmentation du nombre de clients, d'entreprises conceptrices et donc de satellites amène à une saturation de l'orbite terrestre. L'ensemble des déchets lié aux lancement est déjà réglementé à 4 par mise en orbite [VVPV08]. Mais qu'en est-il des satellites ? Chaque pays pouvant construire un lanceur peut mettre en orbite ce type de systèmes. L'organisation internationale de régulation du nombre de satellites et de leurs positions n'est pas toujours consultée pour des raisons politiques car elle appartient aux États-unis. Un exemple est la mise en orbite de satellites Nord-Coréen non déclarés [Sav13]. Il y a donc une plus grande probabilité de rencontre fortuite entre deux objets spatiaux en orbite terrestre. La demande et le nombre de satellites déjà en orbite sont tels que les industriels ne sont plus en mesure de répondre à toutes les demandes sans accélérer leurs temps de production et augmenter le nombre de clients utilisant ceux-ci simultanément. En 2010, le marché annuel des satellites de télécommunications en orbite géostationnaire est en moyenne de 20 à 25. Ils sont produits par quatre sociétés américaines : Space Systems/Loral, Boeing, Lockheed Martin, Orbital Sciences ; et deux européennes : Astrium Satellites (Aujourd'hui Airbus Defence and Space) et Thales Alenia Space [1].

Le modèle logiciel développé par les industriels depuis 2010 prend en compte la mise en place d'un schéma hypervisé. La durée de développement d'un tel système logiciel est estimée par l'ensemble des industriels du domaine aérospatial à 10 ans. Ce modèle d'hyperviseur est potentiellement effectif en "production" jusqu'en 2036 et en "fonctionnement" jusqu'en 2046. En attendant la mise en oeuvre complète d'un tel schéma, le système logiciel choisi est un développement bare-métal.

La réponse des industriels a été de normaliser et standardiser des procédés de production. Cette standardisation, nommée ARINC 653, a permis aux entreprises de faire appel à des sociétés sous-traitantes extérieures [Spe06].

2.2 La contrainte embarquée (ARINC 653)

ARINC (*Avionics Application Software Standard Interface*) [Spe06], est un standard qui permet d'améliorer les procédures de développement d'un satellite ou d'un avion. Le standard ARINC 653 définit en 3 volumes les techniques générales de développement des parties électroniques et logicielles pour répondre aux besoins industriels. Ce document ne fournit pas une solution technique précise mais une aide au développement afin que chaque concepteur suive le même standard et utilise un langage commun aux les sous-traitants et autres producteurs de satellites.

La figure 2.1 montre la répartition des tâches de productions de nos jours et historiquement. Nous observons dans cette figure que les responsabilités de productions ont bien changé. Anciennement uniquement basées sur l'unique société productrice du satellite, elles sont vérifiées en fin de chaîne par l'Etat (car un satellite reste une arme de guerre) et par le client final. Aujourd'hui, le schéma de production est bien plus complexe. En effet, bien que la recherche reste propriété exclusive de l'industriel (avec des partenariats publics-privés), le reste est confié à des sous-traitants. Ces sous-traitants sont responsables de leurs productions et remettent le produit fini à l'industriel.

1. Christian Lardier, « Les satcoms ne connaissent pas (encore) la crise », dans Air & Cosmos, N 2325, 7 septembre 2012

Le client participe également en réalisant la partie propre à son besoin qui n'est pas dépendante de la vie du satellite mais de la mission qu'il occupe. L'industriel réalisent de nos jours des contrôles intermédiaires et finaux sous surveillance de l'Etat.

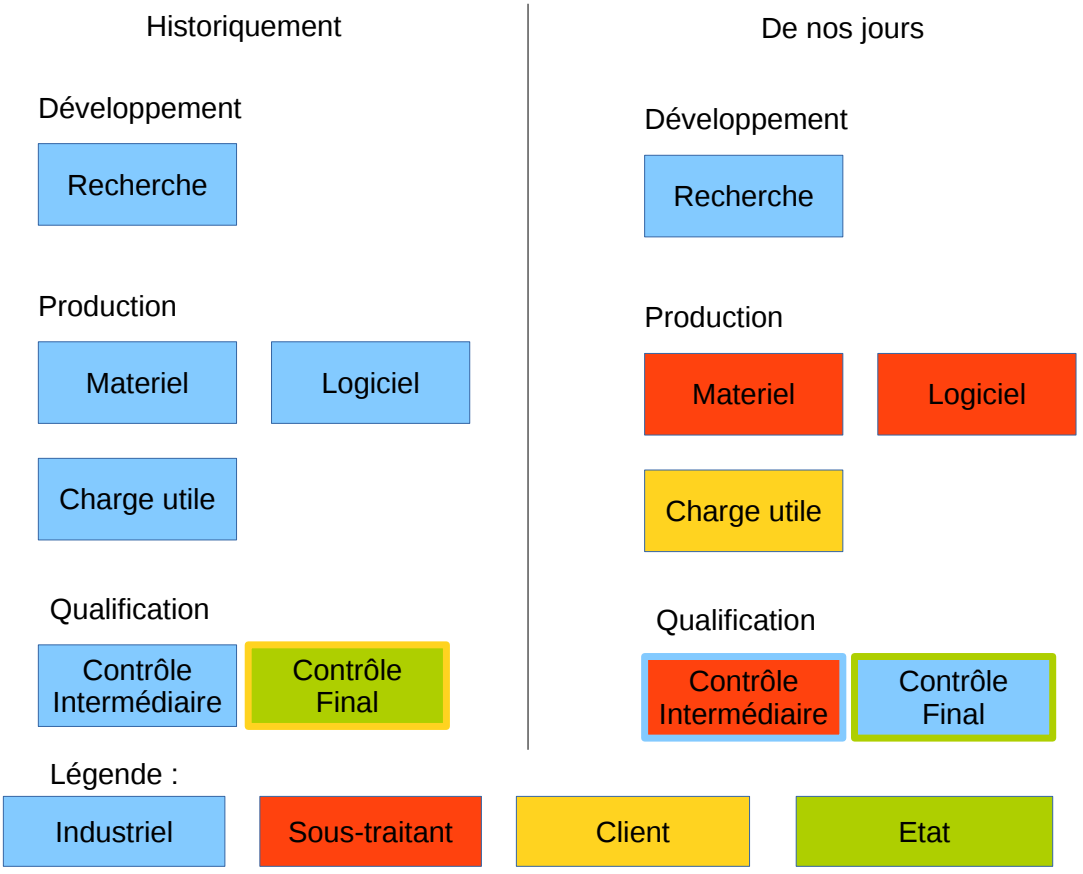

Fig. 2.1 – Vu graphique de la répartition de la production d'un satellite

La documentation ARINC 653, facilite donc le schéma de production explicité au-dessus. Elle permet une généralisation des satellites, avec une évolution systématique qui prend en compte les nouvelles technologies apparaissant sur le marché et pouvant réduire :

— les coûts de production ;
— le délai de production ;
— la masse du satellite ;
— la taille du satellite ;
— la complexité du système ;

et augmenter :

— le nombre de clients par satellite ;
— le nombre de tâches effectuées sur celui-ci ;
— la capacité de calcul du système ;
— le nombre d'entreprises pouvant participer au développement.

Cette avancée est une évolution majeure, clairement établie aujourd'hui. Le temps de développement de satellites d'une capacité de calcul 1 000 fois supérieure aux générations précédentes est passé d'un délai de production de 10 ans, comme par exemple le satellite SARA, à une production de 3 ans pour un cubesat tel que le satellite ANAIS de GEMINISAT [2] ou le cubesat CUTE-I de tokyo tech [NKS+03]. Leur poids est passé de 50 Kg en 1991 avec le satellite SARA à 1.5 Kg pour les satellites ANAIS et CUTE-I. Leurs dimensions passent de 50cm x 50cm x 50cm pour SARA à 10cm x 10cm x 10cm pour les nouveaux Cubesat. A noter que les satellites SARA et ANAIS ont un objectif scientifique commun avec comme seule différence la technologie utilisée. Il en va de même pour les satellites industriels plus imposants. C'est le cas des satellites Hélios 1A (1995) et 1B (1999) [Lan87] qui seront remplacés par les satellites Hélios 2A (2004) et 2B (2009) [Isn01]. Ils ont demandé 10 ans de développement comme les premiers mais sont passés d'une précision photographique de la terre de 2m à moins d'un mètre.

2.2.1 L'évolution de l'électronique

La standardisation permet la mise en place d'un bloc électronique généralisé et numérique. Ainsi, seuls les capteurs sont ajoutés et spécifiques à chaque satellite, mais le coeur de celui-ci est identique quelles que soit leurs missions. La spécificité des satellites est dans le code logiciel, qui ce complexifie fortement. Il permet néanmoins, avec plus de mémoire, de prendre en compte plus de technologie (tel que les multi-coeurs) et de capteurs. Le logiciel normalisé permet également aux nombreux clients de développer eux-même le code qui leur servira sur le satellite même indépendamment de la connaissance du code des autres clients.

Afin d'obtenir ce résultat, le standard ARINC 653 spécifie pour la partie électronique qu'il existe un processeur central et que celui-ci doit répondre aux exigences suivantes :

- il doit avoir suffisamment de traitement de calcul pour répondre aux contraintes de temps maximal de chacune des partitions (tâches) temps-réel ;
- il doit pouvoir accéder aux périphériques d'entrée/sortie et aux différentes mémoires ;
- il doit pouvoir connaître à tout instant le temps afin de pouvoir répondre aux contraintes temps-réels ;
- il doit pouvoir transférer le contrôle d'une partition réalisant une opération invalide à un code de gestion logiciel ;
- il doit être constitué d'opération atomiques (plus petites opérations réalisables) avec un impact faible sur l'ordonnancement temps-réel.

Le terme partition développé au-dessus est plus complexe en réalité qu'une simple tâche. Une partition est un bloc de code représentant un système de vol, ou un code client. Ce bloc a la particularité de devoir être isolé des autres. En effet, nous ne voulons pas qu'un client puisse accéder au bloc de gestion du domaine de vol d'un satellite, ou que deux clients puissent, par erreur ou non, se perturber mutuellement. Le terme partition implique donc la nécessité d'un gestionnaire logiciel visible Figure 2.2.

2.2.2 Les propriétés des partitions

Comme définit dans la norme ARINC 653 [Spe06], le principe de la gestion de partition est le point central de la philosophie de cette norme. "Les applications résidant dans le module sont

2. www.geminisat.com

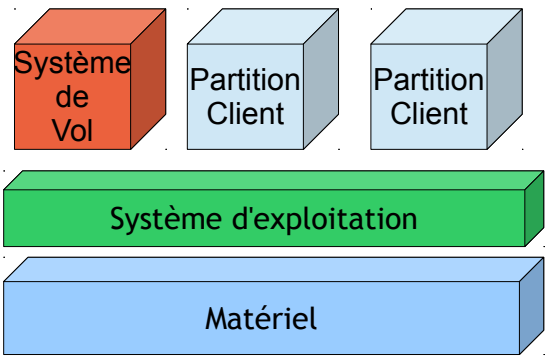

Fig. 2.2 – Système proposé par la norme ARINC 653

partitionnés avec un respect de l'espace (partitionnement de la mémoire) et du temps (partitionnement temporel). Une partition est donc une unité de programme qui réalise une application en satisfaisant les contraintes de partitionnements."

"Il est également spécifié que le système d'exploitation qui opère directement sur la couche matérielle est responsable de la gestion des partitions qu'elle contrôle et du partitionnement de celles-ci. Ce système d'exploitation à l'obligation d'être isolé et protégé contre les dysfonctionnements logiciels de tout code s'exécutant sur une partition." Ceci indique la nécessité d'isoler le matériel de la partie logicielle proprement dite, à l'aide d'un système d'exploitation, car ce système est garant de l'intégrité de l'ensemble du code s'exécutant au-dessus de lui.

La norme ARINC définit également les attributs nécessaires aux partitions. Ces attributs fixent les contours minimum bien que non suffisant. Ainsi il faut pouvoir pour une partition :

— l'identifier à l'aide d'une identité numérique qui permet au système d'exploitation de l'activer et de le gérer ;
— lui donner un nom de type string ;
— lui définir un espace mémoire fixe ;
— indiquer la période correspondant à l'intervalle de temps requis pour que la partition puisse satisfaire l'ensemble de ces tâches ;
— fournir la durée d'exécution de la partition réelle, obligatoirement inférieure ou égale à la période ;
— stipuler si celle-ci nécessite des communications inter-partitions, ainsi que l'identification de ces dernières, le cas échéant ;
— avoir une table de gestion et de détection des défauts lors de son exécution.

L'ensemble de ces attributs nous permettent de mieux appréhender les problèmes que peuvent rencontrer nos satellites. Le respect de ces attributs est impératif, une partition est donc en défaut lors d'un manquement à l'une de ces définitions. Nous retrouvons ici la notion de tâche avec pour chaque partition une date de début et de fin, une durée et une période. Cette notion de tâche prend bien en compte la nécessité temps-réel de l'ensemble du système malgré l'ajout d'un système d'exploitation au-dessus du matériel.

2.2.3 Les propriétés temporelles

D'un point de vue temporel, 3 notions d'ordonnancement sont ajoutés :

— pour un développeur d'application sur le système d'exploitation, l'unité d'ordonnancement est la partition ;
— la partition n'a pas de priorité ;
— l'algorithme d'ordonnancement des partitions est fixe et prédéterminé, répétitif avec une périodicité et configurable via une table de configuration fixe uniquement.

Ces notions sont importantes car elles définissent le coté statique de l'ordonnancement du système d'exploitation. Nous développerons ces caractéristiques au fur et à mesure de la thèse, car les isolations spatiales et temporelles sont les points critiques de la gestion des partitions. Notre objectif dans la suite du document sera de mettre en difficulté ces deux règles.

La technologie retenue par les industriels pour réaliser la gestion du partitionnement spatial et temporel (le système d'exploitation) est l'utilisation d'hyperviseurs et de micronoyaux.

2.3 Introduction des micronoyaux, et hyperviseurs

De manière générale les hyperviseurs et micronoyaux sont considérés comme des systèmes de virtualisation. La virtualisation est une technologie matérielle ou logicielle permettant de simuler le comportement d'un système informatique donné, sur lequel fonctionne un ou plusieurs logiciels. Il est possible de virtualiser des composants matériels comme un processeur, des contrôleurs de mémoire ou des cartes réseau, ainsi que des éléments logiciels comme des systèmes de fichier ou des piles réseau. Cette technologie est apparue à la fin des années 1960 avec l'apparition de l'ordinateur central (mainframe) IBM System/370. D'après J.Wlodarz [Wlo07] elle apporte des avantages notables dans les domaines suivants : gestion des ordinateurs, développement et débogage de logiciel ou encore pour l'isolation forte de logiciel.

Pour faire fonctionner plusieurs systèmes d'exploitation simultanément sur un même ordinateur, il est possible de faire un recours à un moniteur de machine virtuelle, appelé communément hyperviseur, se chargeant de gérer l'état interne des composants contenus dans un ordinateur. Les composants sont, entre autre, le processeur et le contrôleur de mémoire. Un système d'exploitation utilise les périphériques en se considérant seul système en fonctionnement sur la machine virtuelle. L'image 2.3 schématise le fonctionnement d'un satellite avec un système d'exploitation classique (à gauche) et avec un hyperviseur (à droite). Dans le premier cas, le système d'exploitation gère le processeur, la mémoire et les périphériques. Dans le second cas, ceux-ci sont gérés par un hyperviseur. Les systèmes d'exploitation utilisent alors des composants virtualisés.

Les propriétés d'un hyperviseur

Un hyperviseur doit respecter plusieurs propriétés décrites dans les travaux de Popek et Goldberg [PG74]. Ces propriétés sont axées sur la performance et la sécurité du système. L'hyperviseur doit tout d'abord être le seul logiciel contrôlant totalement les ressources du système. Les machines virtuelles gérées par l'hyperviseur doivent ensuite respecter les trois propriétés suivantes :

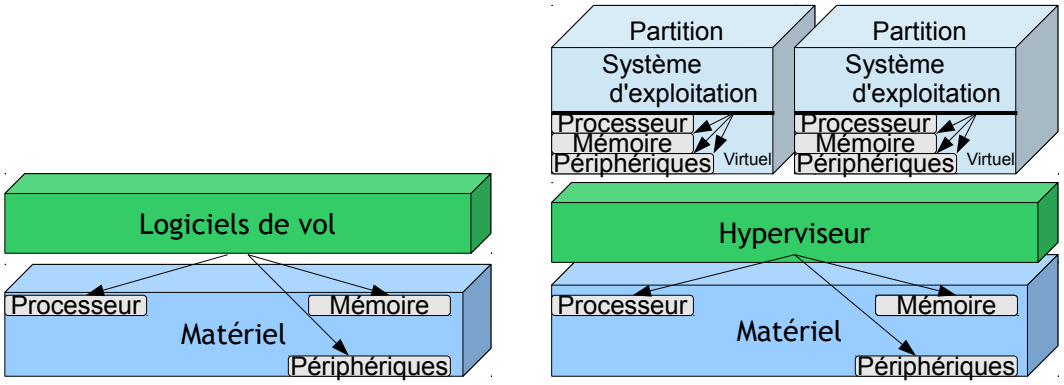

Fig. 2.3 – Un satellite fonctionnant sans et avec hyperviseur.

Efficacité : toutes les instructions dites inoffensives[3] doivent être exécutées directement sur le processeur sans intervention de l'hyperviseur ;

Contrôle des ressources : l'hyperviseur contrôle complètement les ressources physiques de la machine sur laquelle il fonctionne. Il doit pouvoir garantir le non-accès aux informations d'une machine virtuelle via une autre si elle n'y a pas été autorisée au préalable, car une machine virtuelle ne peut pas s'attribuer plus de ressources que celles attribuées par l'administrateur ;

Équivalence : tous les programmes fonctionnant dans une machine virtuelle doivent avoir un comportement identique à une exécution sur une machine réelle. En revanche, leur temps d'exécution et leurs ressources à disposition peuvent être différents. Dans le cas d'une exécution sur une machine virtuelle, l'hyperviseur peut occasionnellement intervenir lors de l'exécution de certaines instructions d'un programme, ce qui augmente son temps d'exécution. Il est donc difficile d'effectuer une estimation précise du temps d'exécution d'un programme fonctionnant dans une machine virtuelle (Nous développerons plus ce point dans la suite du document).

Pour réaliser un hyperviseur respectant ces propriétés, l'architecture matérielle en-dessous doit respecter deux critères :

3. Les instructions inoffensives sont des instructions qui n'influence, n'accède ou ne modifie pas de registres critiques, comme par exemple, celui configurant le mode d'exécution du processeur.

Instructions privilégiées : le processeur génère une faute si un logiciel s'exécutant dans le contexte utilisateur tente d'utiliser une instruction privilégiée ;

Instructions sensibles : ces instructions permettent de modifier ou lire la configuration des ressources du système. Elles doivent être définies en tant que sous-ensemble des instructions privilégiées. C'est-à-dire que le processeur doit aussi générer une faute si une de ces instructions est exécutée.

Lorsqu'un processeur vérifie ces deux critères, l'hyperviseur a la possibilité d'agir avant qu'une machine virtuelle exécute une instruction privilégiée. L'hyperviseur pourra donc être en mesure de vérifier l'innocuité de cette instruction vis-à-vis de lui-même, du système et des autres machines virtuelles. Il y a plusieurs années, l'architecture processeur x86, la plus utilisée dans le monde, ne respectait pas le critère des instructions sensibles. Ainsi, plusieurs de ces instructions échouaient silencieusement sans générer de faute lorsqu'elles étaient exécutées en dehors du mode noyau tel que nous pouvons le voire à la figure 2.4. Une solution peut être utilisée pour permettre l'utilisation d'hyperviseur sur des processeurs non adaptés. Cette technique s'appelle la translation dynamique de code.

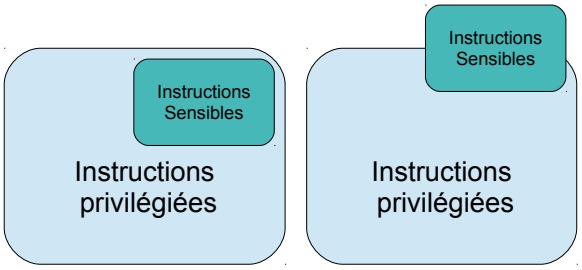

Fig. 2.4 – A gauche, les instructions sensibles sont un sous-ensemble des instructions privilégiées. Le processeur est virtualisable. A droite, le processeur n'est pas virtualisable en l'état (x86).

La translation dynamique de code

La translation dynamique de code permet la transformation d'une suite d'instruction en une autre pendant l'exécution d'un programme. Cette technologie est utilisée dans la majorité des hyperviseurs lorsque le processeur ne respecte pas les propriétés d'instruction privilégiées ou sensibles. Ces instructions non gérées sont transformées avant leur exécution afin de générer des fautes processeurs. Cela permet de simuler le comportement d'un processeur virtualisable. L'hyperviseur peut donc maintenant émuler les instructions sensibles [AA06] en respectant les propriétés vitales de celui-ci. Le schéma 2.5 illustre cette technologie :

— les instructions courantes sont exécutées directement par le processeur ;
— le processeur émet une faute et donne le contrôle à l'hyperviseur lorsqu'un système d'exploitation d'une machine virtuelle tente d'exécuter une instruction privilégiée ou sensible ;

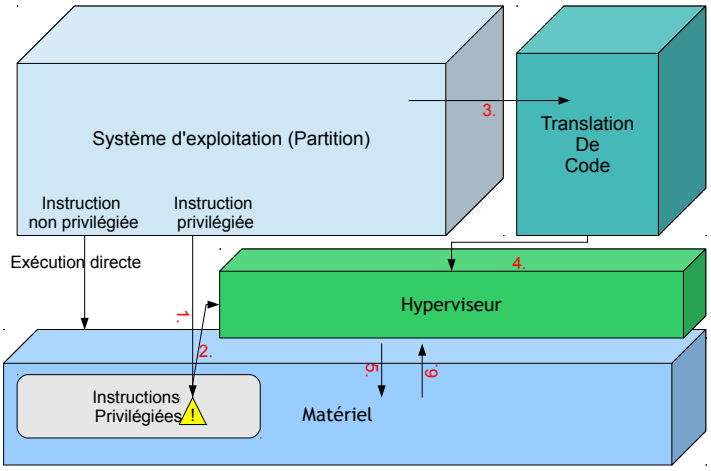

Fig. 2.5 – Schéma d'exécution des instructions normales et privilégiées (allant de 1. à 6.)

— l'hyperviseur analyse dynamiquement le code binaire des machines virtuelles pour remplacer automatiquement les instructions sensibles [4].

2.3.1 Classification des hyperviseurs

Les contraintes exposées précédemment ne fournissent pas de solutions technologiques précises pour concevoir l'architecture logicielle d'un hyperviseur. Nous présentons dans cette section les trois grandes catégories d'architectures existantes, à savoir, les hyperviseurs natifs, applicatifs et la paravirtualisation.

Hyperviseur natif

Les hyperviseurs natifs englobent ceux conçus comme des systèmes d'exploitation dédiés à la virtualisation. C'est-à-dire qu'ils remplacent les systèmes d'exploitation traditionnels. Pour administrer ces hyperviseurs natifs, il faut généralement dédier une machine virtuelle privilégiée sur laquelle sont installés un système d'exploitation et les outils de gestion de l'hyperviseur. Cette machine virtuelle possède des droits administratifs et doit seulement être utilisée pour la configuration de l'hyperviseur. Ce type d'hyperviseur est utilisé sur les serveurs et sa gestion s'effectue dans la majeure partie des cas à distance en se connectant à la machine virtuelle d'administration.

Le schéma 2.6 représente l'architecture d'un hyperviseur natif. Les hyperviseurs natifs possèdent de bonnes performances grâce à leur simplicité de fonctionnement. Ils sont très utilisés pour renforcer les infrastructures informatiques et réduire les coûts de maintenance des centres de traitement de données. Ces hyperviseurs apportent également une grande modularité en permettant la migration en cours de fonctionnement de machines virtuelles d'un hyperviseur à un autre [5]. Cela facilite

4. En pratique, l'hyperviseur remplace directement les instructions sensibles et privilégiées par une suite d'instructions pour simuler correctement leurs fonctions. Cela évite des exceptions processeurs systématiques ayant un coût très important en cycles d'horloge

5. cette fonctionnalité de migration de machine virtuelle présente un intérêt dans la gestion des serveurs.

la gestion de la répartition des charges et la gestion des défaillances. Les hyperviseurs suivants fonctionnent sur ce principe : VMware ESXi [ESX], VMware ESX [VMW05], Parallels Server for Mac [RWR96] et Microsoft Hyper-V [VV09].

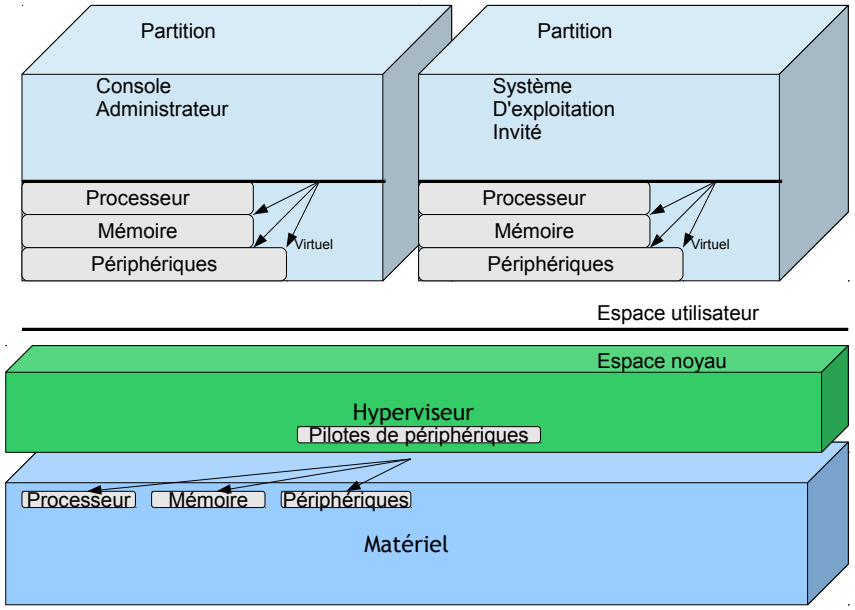

Fig. 2.6 – Représentation d'un hyperviseur natif

Hyperviseur applicatif

Cette catégorie englobe les hyperviseurs conçus pour fonctionner comme une application dans un système d'exploitation classique (GNU/Linux, Mac OS, Windows, ... etc.). L'hyperviseur délègue une partie de ses tâches au système hôte et effectue les autres à l'aide d'un module noyau. La programmation de l'hyperviseur est simplifiée mais peut quand même effectuer les tâches privilégiées dont il a besoin. Le schéma 2.7 représente l'architecture d'un hyperviseur applicatifs avec des machines virtuelles qui s'exécutent au côté d'applications du système d'exploitation hôte. Les hyperviseur applicatifs sont utilisés sur les ordinateurs de travail. Leurs performances dépendent fortement du système d'exploitation hôte. Ces hyperviseurs simplifient la tâche des utilisateurs en permettant de faire fonctionner une application conçue pour Windows, par exemple, sur un système d'exploitation GNU/Linux sans redémarrage complet du système. Les hyperviseurs qui fonctionnent sur ce principe sont principalement : VMware Player [Zim06], VMware Workstation [SVL01], VMware Fusion [Bou07], Oracle VirtualBox [Ora13], Parallels Desktop for Mac [MAC], QEMU [Bel07] et Microsoft VirtualPC [Hon03].

Paravirtualisation

La paravirtualisation est une technique moins intuitive que les deux précédentes. Elle ne respecte pas la propriété d'équivalence définie précédemment, ce qui veut dire qu'elle requiert la modification des systèmes d'exploitation installés dans les machines virtuelles.

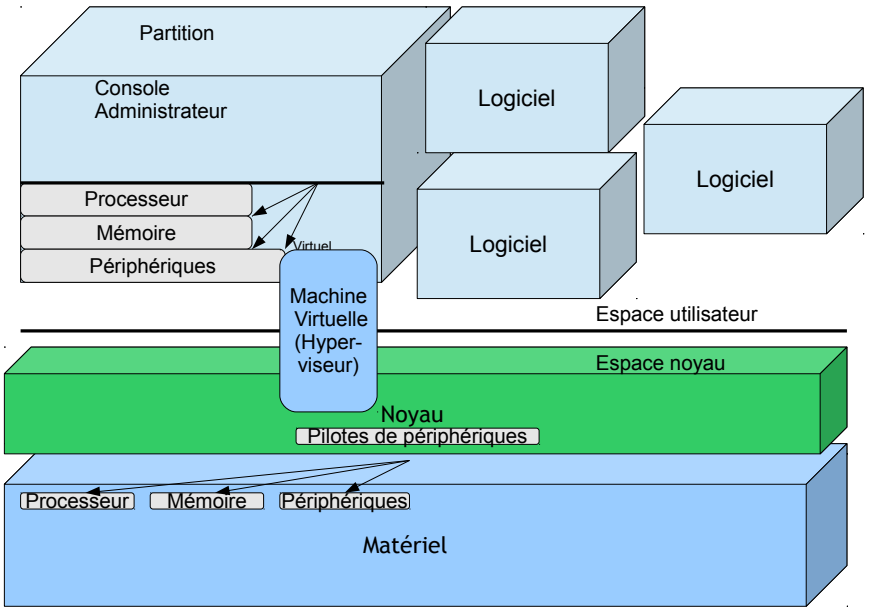

Fig. 2.7 – Représentation d'un hyperviseur applicatif

La paravirtualisation a été présentée pour la première fois par une équipe de l'université de Cambrigde [BDF+03]. Le but était d'éviter la complexité et la perte de performance provoquées par la translation dynamique de code. En supprimant la contrainte d'équivalence, il est possible de modifier le noyau des systèmes d'exploitation invités. Les instructions privilégiées ou sensibles présentes dans ces derniers sont alors supprimées et remplacées par un appel à l'hyperviseur (hypercall) qui les émulent (schéma 2.8). La modification de l'interface binaire/programme (Application Binary Interface) des systèmes invités n'est pas autorisée afin de garantir le bon fonctionnement des logiciels précompilés sur ces systèmes [6].

Le canal de communication explicite entre les machines virtuelles et l'hyperviseur via l'utilisation d'hypercalls présente deux avantages significatifs par rapport aux autres types d'hyperviseurs :

— la conception de l'hyperviseur est simplifiée (pas d'instructions problématiques à traiter) ;
— le surcoût de la virtualisation est faible puisque les vérifications sur les instructions à exécuter ne sont plus nécessaires et les exceptions processeurs sont évitées.

En revanche, les noyaux des systèmes d'exploitation nécessitent une modification pour fonctionner. Cela peut être problématique pour les systèmes d'exploitation propriétaires dont le code source n'est pas disponible.

Comme explicité dans [HL10] et [WSG02], cette technologie présente une forte ressemblance avec les micronoyaux. En plus de Xen [BDF+03], les hyperviseurs VMware Workstation [SVL01] et Microsoft Hyper-V [VV09] peuvent fonctionner dans ce mode pour accroître leurs performances.

6. L'interface binaire/programme (ABI) est l'interface de communication entre les logiciels utilisateurs et le noyau du système d'exploitation. Un programme utilisateur peut utiliser un appel système (« syscall » en anglais) pour demander au noyau d'effectuer une tâche qui nécessite une vérification de sa part (comme par exemple, une demande d'accès à une nouvelle zone de mémoire). Cette interface de communication est rendue transparente par l'utilisation d'une bibliothèque de bas niveau (« libc » dans GNU/Linux), ce qui facilite la programmation logicielle. Une modification de l'ABI au niveau du noyau implique une adaptation de la bibliothèque. Cela nécessite une recompilation de l'ensemble des logiciels utilisateurs (qui dépendent tous de cette bibliothèque).

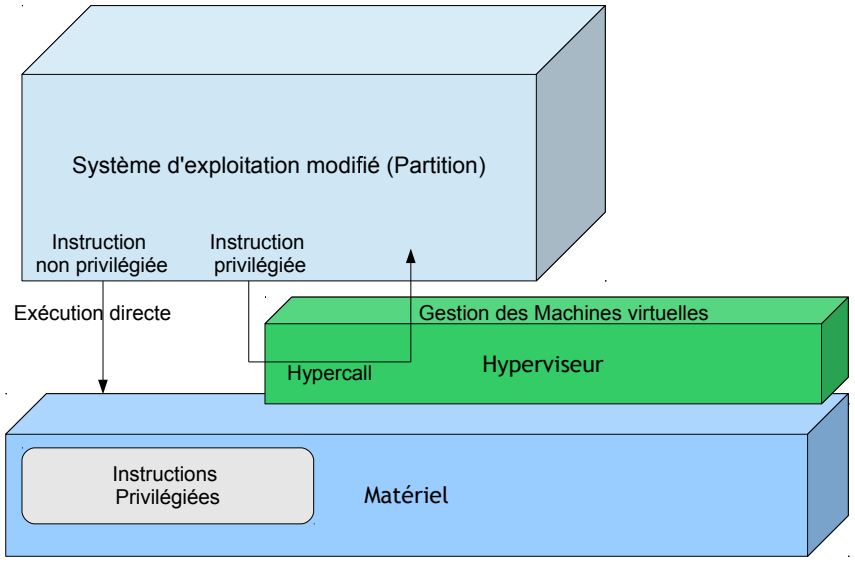

Fig. 2.8 – Représentation d'un hyperviseur utilisant la paravirtualisation

Partie matérielle

Nous avons observé précédement qu'un processeur doit posséder certaines propriétés afin de faire fonctionner un hyperviseur. La virtualisation étaient peu utilisée jusqu'au début des années 2000. Les deux architectures qui font fonctionner parallèlement plusieurs systèmes d'exploitation sont, à cette époque, ESA/370 et ESA/390 d'IBM.

Depuis la généralisation de la virtualisation, entre autre dans les centres de données, les architectures ont été modifiées pour répondre aux propriétés des hyperviseurs fixés par Popek et Goldberg. L'architecture x86 qui possède des instructions sensibles non privilégiées [For00] évolue avec l'apport d'un nouveau mode (technologie Intel VT-x et AMD-V) dans les processeurs des deux principaux fabricants. Ce nouveau mode permet à l'hyperviseur de configurer le processeur afin de détourner le flot d'exécution vers ce premier lorsqu'une machine virtuelle exécute une instruction à problème.

Les technologies Intel Extended Page Table et AMD Rapid Virtualization Indexing permettent, elles, d'améliorer les performances de la translation d'adresse auparavant effectuées par l'hyperviseur. Les hyperviseurs cités dans les sections précédentes supportent ces technologies.

Aujourd'hui, les architectures les plus utilisées (Sparc, Power, ARM) possèdent des technologies équivalentes.

Virtualisation de périphérique

Il ne suffit pas de virtualiser le processeur pour qu'une machine virtuelle puisse tourner dessus. D'autres composants sont essentiels au bon fonctionnement d'un ordinateur : contrôleur d'affichage, contrôleur IDE ou SATA, contrôleur audio, carte Ethernet, etc. Ceux-ci sont normalement émulés par l'hyperviseur, mais il est aussi possible d'en virtualiser certains afin d'augmenter les performances des machines virtuelles. Il existe deux façons de virtualiser un périphérique :

— dédier une machine virtuelle à cette tâche et l'utiliser comme proxy pour les autres machines virtuelles (figure 2.9). Cette technique est courament utilisée dans l'aérospatial ;
— utiliser des périphériques qui offrent une aide à la virtualisation [RS07, RSW+06] (figure 2.10).

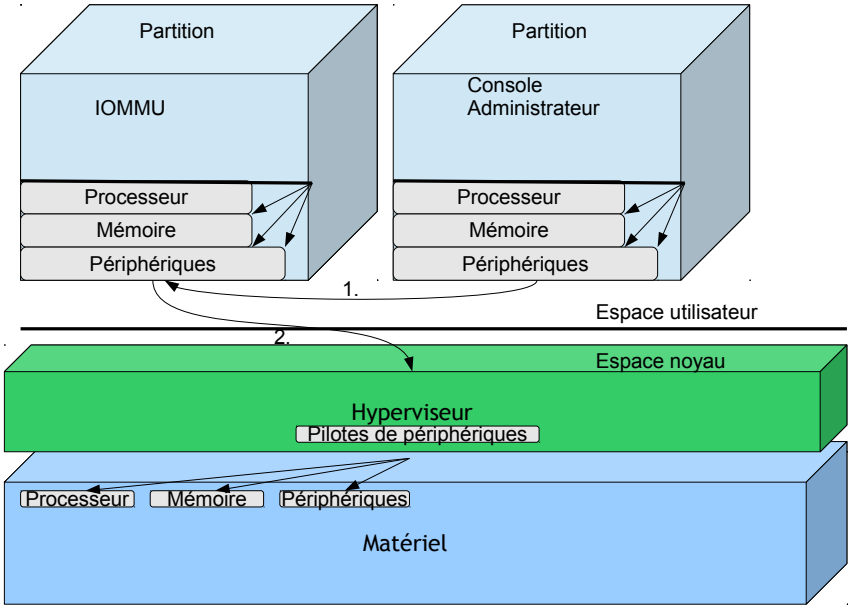

Fig. 2.9 – Une machine virtuelle sert de proxy pour les accès aux périphériques.

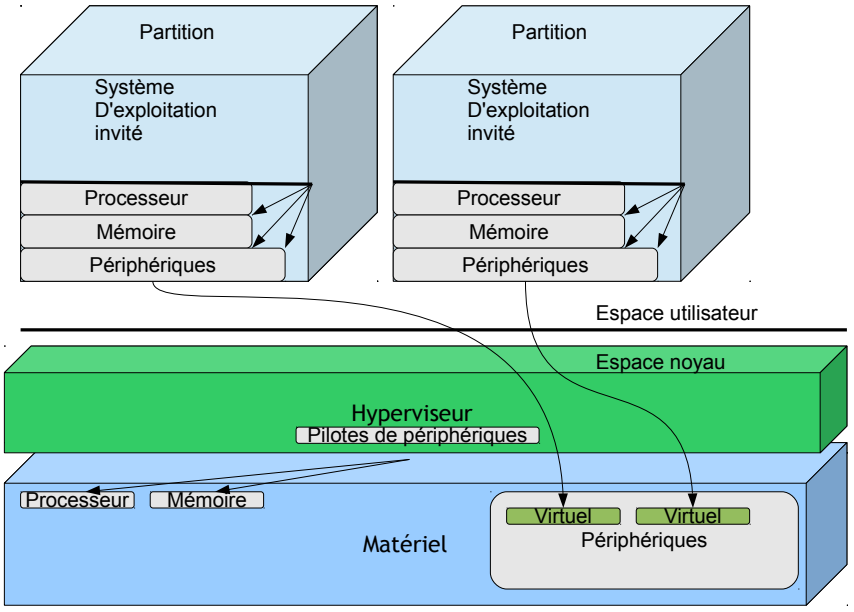

Fig. 2.10 – Deux machines virtuelles accèdent aux périphériques virtuelles.

Les systèmes Intel VT-d et AMD-VI mettent en oeuvre une unité de gestion de la mémoire

pour les entrées et sorties (IOMMU). Un hyperviseur peut utiliser celle-ci pour contrôler les requêtes DMA [7] et ainsi dédier un périphérique à une machine virtuelle. L'IOMMU peut être utilisée pour que chaque machine virtuelle possède un contrôleur réseau dédié. Il est aussi possible de permettre à une machine virtuelle d'accéder directement à une carte graphique afin qu'elle envoie des commandes OpenGL pour effectuer un rendu 3D.

2.3.2 Virtualisation pour les systèmes embarqués

Les objectifs de la virtualisation dans les systèmes embarqués sont plus complexes et différents de ceux exposés précédemment. Les systèmes embarqués nécessitent, contrairement aux serveurs ou aux ordinateurs classiques, de prendre en compte de nouvelles contraintes lors de la conception de l'hyperviseur (notamment les ressources limitées et le temps réel).

Les solutions de virtualisation spécifiques à ce domaine ont été introduites avant d'avoir de réels cas d'utilisations. La virtualisation pour les systèmes embarqués n'a pas encore atteint un stade de maturité aussi avancé que pour les ordinateurs classiques. Cela s'explique par le nombre réduit de développeurs de ce type de système. Néanmoins, des recherches universitaires sont actuellement menées sur la résolution de problématiques de virtualisation pour systèmes embarqués. Quelques cas d'utilisation de ce type de virtualisation ont déjà été développé :

— gestion des identités sur un téléphone mobile (profil personnel ou professionnel) [JFA13] ;
— isolation de la fonctionnalité principale des gadgets (par exemple fonction de navigation et lecteur de musique sur un GPS) ;
— protection du noyau des logiciels malveillants [ISM09] ;
— intégration de logiciel sous licences contaminantes [Hei09] ;
— isolation des services clients dans un satellite [DDAN12].

Liste d'hyperviseurs embarqués temps-réel

La virtualisation pour les systèmes embarqués progresse continuellement. Voici une liste non exhaustive de solutions, commerciales ou non :

OKL4 : cette solution est basée sur un micronoyau L4. Fonctionnant sur les processeurs ARM9 et ARM11, elle utilise la mémoire cache TLB (Translation Lookaside Buffer) qui permet d'accélérer la traduction des adresses virtuelles en adresses physiques. La mémoire cache doit être effacée à chaque changement de contexte afin que le processus du contexte courant ne puisse accéder à des zones mémoires de l'ancien contexte. Bien que réduisant les performances du processeur de l'architecture ARM, cela permet au processeur de marquer les entrées du TLB avec un contexte. Ceci limite le nombre d'effacements de la mémoire cache et les pertes de performance lors d'un changement de contexte [vSH07]. OKL4 permet de gérer des machines virtuelles temps-réel, mais seule la version 2.6.24 du noyau Linux a été portée sur ce parahyperviseur [8]. Cette solution est développée par l'entreprise OKLabs (rachat par General Dynamics en 2012) et certaines versions sont en parties libres.

7. Les requêtes DMA (pour Direct Memory Access) permettent, une fois initialisées, d'effectuer des opérations d'entrées/sorties entre deux périphériques sans que le processeur intervienne. Cela permet l'augmentation des performances d'un ordinateur car le processeur peut effectuer d'autres opérations pendant le transfert. Les périphériques communiquant par DMA possèdent un accès total à la mémoire d'une machine. Le DMA est utilisé dans l'aérospatial

8. Dans la suite, « parahyperviseur » désigne un hyperviseur utilisant la technologie de paravirtualisation.

CodeZero : Codezero est un système d'exploitation temps réel de type micronoyau reprenant l'architecture de L4. Il fonctionne sur architectures ARM et est conçu par la société B-Labs [Jon11].

Integrity Multivisor : cet hyperviseur est développé par Green Hills Software et fonctionne sur ARM. Très peu d'informations sont disponibles à propos de cette solution[KLE].

L4Android : L4Android est un portage du système Android dans un micronoyau L4 (qui peut servir de parahyperviseur) [LLL+11].

PikeOS : ce système d'exploitation est un micronoyau temps réel capable d'isoler d'autres systèmes d'exploitation. Il a été conçu pour fonctionner sur des systèmes critiques (avionique notamment) [KK12, BFB09]. Cette solution supporte les architectures x86, PowerPC et MIPS. Les systèmes d'exploitation invités ou les environnements d'exécution suivants sont capables de fonctionner sur cette solution : Linux, Posix Realtime, ARINC 653, OSEK OS, iTRON, SoftPLC, Ada et Real-Time Java. Cet hyperviseur est développé par la société SYSGO AG (rachat par Thales en 2012).

VirtualLogix : cette solution est un hyperviseur natif temps réel compatible ARM et x86. Cette solution est commercialisée par VirtualLogix (rachat par RedBend Software en 2010). Cet hyperviseur permet une gestion souple des pilotes de périphérique inspirés de Xen [AG09].

VMware Horizon Mobile : cette solution est développée et vendue par VMware (anciennement Trango Virtual Processors) [BBD+10]. Très peu d'informations sont disponibles.

Wind River Hypervisor : ce système est un hyperviseur natif pour x86 et PowerPC. Il permet la gestion de machines virtuelles temps-réel. Sont fournis notamment avec les systèmes d'exploitation de la même société (Wind River) : VxWorks et Wind River Linux [WHK+07].

Xen pour ARM : il existe plusieurs travaux d'adaptation de l'hyperviseur Xen pour système embarqué. Ils sont portés sur ARM. En voici deux : Ferstay [Fer06], Xi [XWLG11].

XtratuM : XtratuM est un hyperviseur temps réel natif qui effectue de la paravirtualisation. Cet hyperviseur est prévu pour fonctionner sur des processeurs LEON (architecture Sparc V8) utilisés dans les systèmes aérospatiaux embarqués (satellites) [MRPC10]. Ce système est développé par la société FentISS et distribué sous licence GPL.

Xvisor : Xvisor est un hyperviseur natif pour ARM et x86. Il est publié sous licence GPL depuis 2011 [PDSEK].

Nous pouvons également cité les travaux de Kiszka [Kis09] portant sur l'adaptation du noyau Linux en hyperviseur temps-réel.

Certaines de ces solutions ont été retenues pour le domaine aérospatial. C'est le cas d'XtratuM employé par la société AIRBUS dans le domaine spatial. C'est également le cas de PikeOS utilisé par la DGA (Délégation Général de l'Armement).

Les spécificités de la partie physique

L'architecture ARM n'est devenue virtualisable qu'à la version 7 [VH11]. Avec la dernière spécification de l'architecture, ARM a introduit de nouvelles instructions et un nouveau mode dédié aux hyperviseurs [Bra10] semblable aux extensions Intel VT-x et AMD-V.

La fonctionnalité TrustZone est dorénavant disponible sur certains processeurs ARM (depuis l'architecture ARMv6) et propose des fonctionnalités matérielles pour héberger dans un contexte spécifique et sécurisé des informations importantes. Cela permet de faire fonctionner parallèlement sur un même processeur ARM un système d'exploitation minimaliste (offrant des fonctionnalités pour la sécurité du système) et un système d'exploitation généraliste. Les travaux d'OpenVirtualization [9] présentent un hyperviseur fonctionnant dans la partie TrustZone des processeurs ARM.

Pour finir, certains processeurs x86 pour l'embarqué (gamme Atom) de la société Intel supportent la technologie Intel VT-x. Ces processeurs permettent de faire fonctionner l'ensemble des solutions de virtualisation existantes pour cette architecture.

Évaluation

La virtualisation des systèmes embarqués est un marché jeune et fortement concurrentiel. Cela entraîne une difficulté d'accès à ces produits et à leurs spécifications, malgré nos demandes.

Nous nous sommes basés sur les documents publics disponibles des sociétés afin d'évaluer les différents hyperviseurs présentés. Dans le tableau 2.1, nous classons les hyperviseurs pour systèmes embarqués avec les critères suivants :

— architecture processeur ;
— architecture de l'hyperviseur ;
— support du temps-réel ;
— systèmes d'exploitation invités disponibles ;
— disponibilité du produit.

2.3.3 Problématique

Concevoir et programmer un hyperviseur n'est pas forcément très difficile pour une personne ayant de bonnes connaissances en système d'exploitation. Cependant, cela reste une tâche longue et fastidieuse même si la tendance est à la création d'hyperviseurs minimalistes afin de réduire la taille de la base de confiance (TCB pour Trusted Computing Base). De plus, dans ce domaine, une erreur ou un manque de rigueur lors de la vérification de paramètres peuvent avoir des conséquences importantes et remettre complètement en cause la sécurité des machines virtuelles et de l'hyperviseur lui-même — voire d'une infrastructure complète si des machines virtuelles assurent des tâches critiques. Les hyperviseurs Xen et VMware ont déjà été l'objet à plusieurs reprises de problèmes de sécurité. Dans le domaine de l'embarqué, les conséquences de tels problèmes peuvent être d'autant plus graves que ces systèmes sont souvent utilisés en milieu critique.

Dans le cas des satellites, les opérateurs doivent faire confiance à certaines machines virtuelles dédiées aux systèmes de vol et aux périphériques. Une personne qui parviendrait à prendre le contrôle d'une machine virtuelle sensible en infectant au préalable une machine virtuelle invitée (ou client) sur laquelle fonctionne la pile logicielle des satellites pourrait ainsi créer des dégâts importants sur le système de son opérateur. Cela pourrait rendre totalement inopérant le système de contrôle du satellite. Il en va de même si deux partitions invitées réussissent à communiquer entre elles sans autorisation ou à perturber le bon fonctionnement du satellite.

9. http ://www.openvirtualization.org

Nom	architecture processeur	architecture de l'hyperviseur	support du temps réel	système d'exploitation invité si paravirtualisation	disponibilité de l'hyperviseur
OKL4	ARM9	micro-noyau	oui	Linux, Windows, Symbian Version 3.0 en GPL avec support commercial.	Support de la virtualisation limité dans la version GPL
VirtualLogix	ARM	Cortex-A7 et A15	hyperviseur natif	oui	Propriétaire
PikeOS	PowerPC, x86, MIPS, ARM, Sparc v8, V850, SH-4	micro-noyau (paravirtualisation)	oui	Linux, Android, muITRON, RTEMS, OSEK	Propriétaire
VMware Horizon Mobile	aucune information	aucune information	aucune information	aucune information	Propriétaire
Wind River Hypervisor	x86, PowerPC, ARM Cortex-A9	hyperviseur natif	oui		Propriétaire
B-Labs CodeZero	ARM Cortex-A5/A8/A9/A15	micro-noyau (paravirtualisation)	oui	Android, Linux	Propriétaire
Integrity Multivisor	ARM, x86, PowerPC	hyperviseur natif	oui	aucune information	Propriétaire
Xvisor	ARM, x86	hyperviseur natif	aucune information		Libre (GPLv2)
XtratuM	LEON2, LEON3, x86, ARM	paravirtualisation	oui	RTEMS, PartiKle, Lithos, Linux	Libre (GPLv3) avec support commercial
Xen pour ARM	ARM9,ARM11	paravirtualisation	non	aucun	Non maintenu ou non disponible
L4Android	ARM Cortex-A9	micro-noyau (paravirtualisation)	non	Android	Libre (GPLv2)

Table 2.1 – Tableau référençant les hyperviseurs embarqués

La virtualisation est une technologie à double tranchant qui nécessite une bonne analyse avant de l'employer. Cependant, il n'existe aujourd'hui aucun outil permettant de tester la fiabilité d'un hyperviseur et d'évaluer le degré de confiance qui peut lui être accordé [BBCL11]. L'aspect temps-réel est également fortement critique pour ce type de système.

2.4 Conclusion

Ce chapitre a démontré l'existence de la problématique multi-client, ainsi que celle de la généricité de gros systèmes. Cette observation ne se limite pas à la simple vision du monde spatiale. En effet, la problématique est également observé dans le monde aéronautique via l'ANR SOBAS, le monde naval avec les frégates multimissions, et le monde férroviaire avec les LGV. Le cadre particulier de cette thèse n'exclus pas la généralisation des observations aux aures domaines applicatifs tel que les systèmes embarqués temps réels imposants. Excluant de fait les micro et nano satellites ayant une mission unique.

L'arrivée des nouveaux systèmes hypervisés change la vision que nous avions des satellites. Il en existe plusieurs disponibles sur le marché ou en cours de développement mais des problèmes de sécurité persistent sur ce nouveau type de système. Le chapitre suivant permettra de mettre en avant les problèmes d'actes malveillants sur un système hypervisé tel que présenté dans ce chapitre.

Chapitre 3

Attaques sur hyperviseurs

Sommaire

3.1 Introduction

Dans ce chapitre, nous présentons en premier lieu les nouvelles contraintes apportées par l'hyperviseur et le changement de vision de la conception d'un satellite. L'impact de ces nouvelles technologies est développé avant de présenter la problématique des actes malveillants. Nous terminons ce chapitre et cette partie de thèse par la présentation d'un cas concret d'acte malveillant. Celui-ci montre la faisabilité d'une telle opération.

3.2 La contrainte temps-réel

La contrainte temps-réel a évolué avec l'arrivée des hyperviseurs, tel qu'XtratuM étudié précédement [MRPC10]. La mise en place d'un hyperviseur a permis de réduire fortement la taille des systèmes électroniques, et par conséquent la taille et la masse des satellites pour une même mission. La réduction de la taille du système d'un point de vue physique est du à la généralisation

de composants non spécifiques. Il a aussi permis une standardisation des systèmes électroniques et la possibilité de développer une plateforme unique pour l'ensemble des satellites en cours de développement. Ainsi, au lieu de développer une plateforme physique pendant 10 ans et d'en perdre les bénéfices une fois la mission envoyée, la plateforme physique reste identique, réduisant ainsi les coûts de développement et fabrication.

D'un point de vue logiciel, cette simplification de l'électronique oblige à une plus grande modularité. La virtualisation permettant de tester, développer, et implémenter sans avoir obligatoirement la partie électronique, les concepteurs peuvent développer le code indépendamment de la connaissance de celle-ci. D'un point de vue embarqué, l'ensemble des contraintes, (coût de production, masse, taille, vibration, température etc...) est donc fortement limité à un système unique robuste et multifonction. Il n'y a donc que des avantages à utiliser ce type de système dans le domaine spatial, si ce n'est en matière de consommation électrique et de complexification de la gestion du temps réel.

Les technologies d'alimentation et de gestion de l'énergie à bord du satellite ne sont pas développées dans cette thèse. Elles sont indépendantes de l'utilisation d'un hyperviseur ou tout autre système logiciel.

En revanche, le coté temporel est un point central de notre document. L'utilisation de l'hyperviseur a un impact direct sur l'aspect temps-réel. Le respect des tâches à effectuer ne prend maintenant plus en compte seulement le matériel et le code logiciel optimisé mais aussi la partie virtualisation avec un hyperviseur. Le standard ARINC 653 [Spe06] étudié précédemment nous stipule que le système de virtualisation doit gérer les partitions de manière prévisible et fixe. La norme ARINC [Spe06] ne stipule pas, en revanche, comment gérer à l'intérieur d'une partition l'aspect temps-réel. Il est plus difficile de déterminer clairement l'ordonnançabilité d'un système par rapport à un système bare-métal. En revanche, ceci apporte la possibilité de réalisé aisément des modifications en vols sur les satellites. Une reconfiguration des missions d'un satellite devient pleinement possible. Bien qu'actuellement limité à la modification du plan d'ordonnancement statique, il pourrait, à terme, voir une reconfiguration complète du logiciel en cours de vie.

3.2.1 L'impact des nouvelles technologies

Nous avons développé la complexité amenée par la couche de virtualisation. En effet, le temps pris par une tache pour s'exécuter est beaucoup plus complexe à estimer ou mesurer. Cette couche n'étant pas standardisée pour un système embarqué spécifique, le temps pris par la partie hypervision dépend entièrement de l'implémentation réelle de chaque hyperviseur. A cela s'ajoute des processeurs plus gros et complexes comparés aux systèmes bare-métal pour intégrer les dit hyperviseurs.

De plus, de nouvelles fonctionnalités se greffent afin de rendre plus performante la partie logicielle, devenue coeur central du satellite. Dans ces nouvelles technologies, nous voyons apparaître les systèmes multi-coeurs, tels que le Leon 4 [GGPZ14] dans le domaine spatial. La technologie multi-coeurs est aujourd'hui en plein essor mais non encore complètement maitrisée. Ce type de système peut répartir la charge de travail sur plusieurs coeurs et cela entraîne un coût de communication entre eux [KOW07]. D'un point de vue temporel, cela se traduit par une difficulté à estimer un temps d'exécution, comme le démontre Holmbacka dans son article [Hol10]. Cette difficulté également évoquée dans Zamorano [ZdlP13] est due à l'imbrication de l'ensemble du système logiciel

et matériel. Ainsi, il n'est plus possible de prédire clairement le temps d'exécution d'un système multi-cores car il dépend entièrement de la partie logicielle au-dessus définissant l'ordonnancement des tâches sur l'ensemble des cores et donc sur le coût de communication réel entre ceux-ci. A cela s'ajoute l'hyperviseur, qui prend un temps non négligeable mais inconnu dans la gestion des hypercalls et autre contrôle de partitions.

De tels systèmes ne pourraient fonctionner sans l'apport également de caches à plusieurs niveaux. Chaque processeur est équipé d'un cache, dit L1, au minimum afin de limiter les appels mémoires. Le cache est une autre technologie qui pose fortement problème dans la mesure du temps [YZ08]. Bien que cette technologie soit indispensable aux yeux des industriels pour augmenter les performances, cela ce traduit encore une fois par une complexité, voire une impossibilité à prédire le temps pris par une tâche ou une partition [SS07]. L'utilisation du cache entraîne donc une forme d'imprévisibilité du système [Pua06]. Le cache, permettant de limiter les accès mémoires, dépend de l'ordonnancement de celui-ci. Il est également lié à l'ensemble du système, puisque le code passant par celui-ci est généré en fonction de l'ordonnancement général du système et de la répartition de la charge sur l'ensemble des processeurs.

Nous détaillerons plus en détails la complexité de ce type de système et les faiblesses en matière de sécurité et de sûreté de fonctionnement que cela peut engendrer. L'ordonnancement du système est la clé de voûte qui permet de prédire les tâches qui seront effectuées, et leurs répartitions dans le système. C'est ainsi l'ordonnancement qui est en mesure de calculer le temps pris par une tâche pour s'exécuter.

3.3 L'ordonnanceur

L'ordonnanceur est le point central permettant la génération d'un système temps-réel ou non. Le standard ARINC 653 détermine l'ordonnancement de deux systèmes :

— l'ordonnancement de l'hyperviseur ;
— l'ordonnancement des tâches intégrées dans chacune des partitions.

3.3.1 L'ordonnancement de l'hyperviseur

L'ordonnancement de l'hyperviseur est fixe. Afin de limiter l'impact non prédictif de l'hyperviseur, le standard ARINC demande un plan d'ordonnancement fixe et statique. Nous avons développé précédemment ces particularités. Ainsi un plan d'ordonnancement fixe, tel que définit dans la norme, prend en compte des tâches périodiques. En l'occurrence, les tâches sont des machines virtuelles (partitions). Nous prenons l'exemple de l'ordonnancement sur système temps-réel avec un hyperviseur respectant le standard ARINC. Les schémas 3.1 et 3.2 proviennent de la documentation technique de l'hyperviseur XtratuM particulièrement adapté au domaine spatial :

Nous observons dans le cas 3.1, un plan d'ordonnancement périodique pouvant faire apparaître autant de tâches que souhaitée. Le plan d'ordonnancement 3.2 est statique et fixe et ne peut donc pas être modifié dans son contenu pendant le vol du satellite. En revanche, il est possible de changer le plan d'ordonnancement complet par un autre. Cette possibilité n'est offerte qu'à une partition dédiée aux systèmes de vol. Chaque ré-ordonnancement a un coût temporel. Il est donc possible de

```
<CyclicPlanTable>
    <Plan majorFrame="1s">
        <Slot duration="20ms" id="0" partitionId="0" start="0ms"/>
        <Slot duration="10ms" id="1" partitionId="1" start="20ms"/>
        <Slot duration="10ms" id="2" partitionId="0" start="30ms"/>
        <Slot duration="30ms" id="3" partitionId="2" start="40ms"/>
        <Slot duration="10ms" id="4" partitionId="1" start="70ms"/>
        <Slot duration="20ms" id="5" partitionId="0" start="100ms"/>
        <Slot duration="10ms" id="6" partitionId="1" start="120ms"/>
        <Slot duration="10ms" id="7" partitionId="0" start="130ms"/>
        <Slot duration="30ms" id="8" partitionId="2" start="140ms"/>
        <Slot duration="10ms" id="9" partitionId="1" start="170ms"/>
        <Slot duration="20ms" id="10" partitionId="0" start="180ms"/>
    </Plan>
</CyclicPlanTable>
```

Fig. 3.1 – Fichier XML contenant un exemple de plan d'ordonnancement d'XtratuM

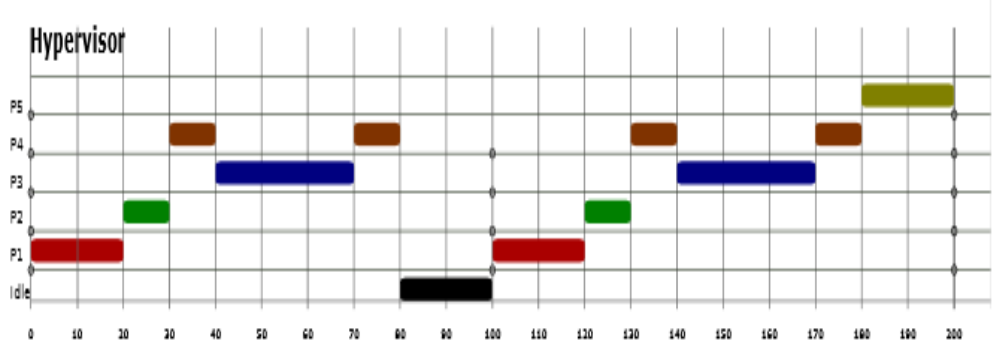

Fig. 3.2 – Vue graphique d'un exemple de plan d'ordonnancement de l'hyperviseur XtratuM

connaître dans le temps, la durée, le début et la fin de chaque partition. Avec un ordonnancement dynamique, il aurait été très difficile de le déterminer dans chaque partition.

3.3.2 L'ordonnancement des partitions

Chaque partition est potentiellement temps-réel. Avec ce plan fixe, il est maintenant possible d'avoir un ordonnancement des tâches contenu dans chacune des partitions de manières différentes. Ainsi une partition peut avoir un ordonnancement différent d'une autre.

Le standard ARINC [Spe06] limite tout de même les ordonnanceurs de partitions avec les propriétés suivantes :

— l'objectif majeur que l'ordonnanceur de partition doit respecter est la gestion perpétuelle de la concurrence entre les différentes tâches souhaitant utiliser les processeurs ;
— chaque tâche est définie avec une priorité fixe ;
— l'algorithme d'ordonnancement doit gérer les préemptions. Il doit également gérer les niveaux de priorités de chaque tâche et leurs états courants. Ainsi, si la tâche en cours d'exécution a un niveau de priorité plus élevé que celles en attente, il ne doit pas être préempté. Lors d'un ré-ordonnancement, l'ordonnanceur doit prendre la tâche avec le niveau de priorité le plus élevé. Si plusieurs d'entre eux ont un même niveau, la tâche ayant attendu son ordonnancement le plus longtemps est sélectionnée ;
— Les tâches périodiques et apériodiques sont supportées ;
— toutes les tâches contenues dans une partition n'utilisent que les ressources allouées à cette partition.

Cette description complète et détaillée montre la nécessité de ne pas avoir de doute sur le fonctionnement de l'ordonnancement à bord d'un satellite. C'est uniquement avec les deux ordonnancement (d'hyperviseur et de partitions) fixés et imbriqués que l'ordonnancement global du système est faisable. Voici un exemple 3.3 d'ordonnancement d'une partition comme défini dans ARINC 653 :

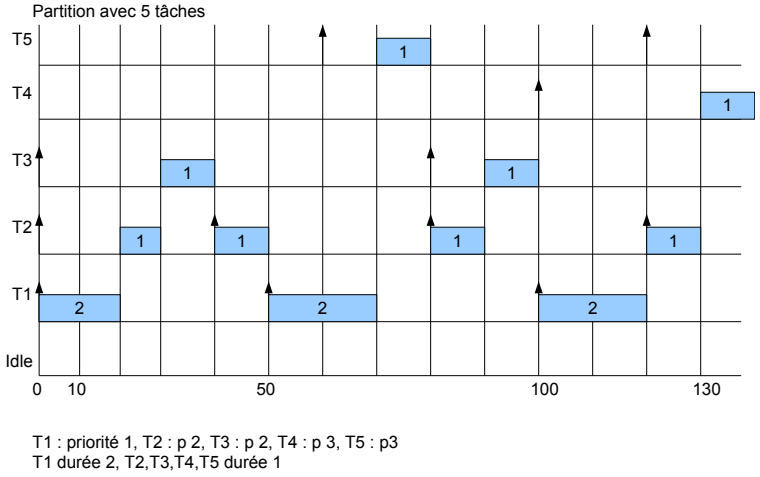

Fig. 3.3 – Vue graphique d'un exemple de plan d'ordonnancement d'une partition avec les contraintes de ARINC 653

L'ensemble de ces changements entraînent à des conséquences inévitables, positives comme négatives en matière de sûreté de fonctionnement.

3.4 Les conséquences en matière de sûreté de fonctionnement

Le projet ANR SOBAS a pour objectif de sécuriser les systèmes satellites et aéronautiques contre des actes malveillants portant atteinte à l'intégrité du système. Plus particulièrement, cette ANR, faisant suite à l'ANR SECPAR, se concentre sur des formes d'attaques particulières. En effet, la classe d'attaque dite d'infection n'est pas étudiée. Il est considéré que tout ou partie du système a déjà été compromis. Le procédé d'infection n'est donc pas prioritaire, en revanche les dégâts qu'une telle infection peut réaliser sont spécialement observés. C'est dans cette vision que ma thèse prend place. Mes connaissances dans les domaines de l'électronique et l'informatique, ainsi que leur point de jonction sont primordiaux, car la majeure partie de ces attaques ont lieu lors de l'intégration entre les couches matérielles et logicielles. Ces classes d'attaques sont nouvelles pour le monde industriel. Il maîtrise les attaques infectieuses, mais se retrouve affaibli par ces attaques utilisant majoritairement le partage des informations sur leurs systèmes avec l'extérieur. Mes connaissances antérieures, dans ce domaine, m'ont permis de développer un grand nombre d'attaques suivant des protocoles et connaissances propres aux cas déjà étudiés en réels. Une partie de la classification des attaques est disponible dans les documents publiques de l'ANR SOBAS.

La sûreté de fonctionnement est l'objectif principal des industriels, au côté de la performance. En effet, avoir un système très performant mais en panne dès la mise en orbite n'est pas vendeur. Cette sûreté de fonctionnement est le critère le plus fort permettant aux industriels de se démarquer entre eux et de faire face à l'arrivée des pays émergeant avec un coût de fabrication plus faible.

De nos jours, la sûreté de fonctionnement est très bien définie. Ainsi, la sûreté est spécifiée comme "l'aptitude d'une entité à satisfaire à une ou plusieurs fonctions requises dans des conditions données" par Alain Villemeur [Vil88], et comme "la propriété qui permet aux utilisateurs du système de placer une confiance justifiée dans le service qu'il leur délivre" par Jean-claude Laprie [ACBL96]. Dans notre domaine, il ressort quatre mots-clés qu'il s'agit de définir :

La fiabilité est la capacité d'un système à fonctionner correctement dans un intervalle de temps donné ;

La maintenabilité est la capacité de maintenir ou remettre en état un système ;

La disponibilité est la capacité à garantir que le service légitime va avoir accès au système sans en être empêché ;

La sécurité (*safety*) est la capacité d'un système à ne pas causer d'accident ou de dégâts pouvant entrainer destructions, blessures ou mort.

En français, nous traduisons généralement *safety* par sûreté de fonctionnement, pour la distinguer de la sécurité/innocuité, à ne pas confondre avec la sécurité telle que nous l'entendons dans ce document, en anglais *security* et correspondant à la gestion des actes malveillants. Les actes malveillants sont tout actes détournant un produit de son utilisation première et nuisant à son bon fonctionnement.

Plus concrètement, l'apport des technologies multi-cores, caches, ... doivent permettre l'évolution des performances des nouveaux systèmes satellites. En revanche, l'hyperviseur est une brique centrale de la sûreté de fonctionnement. Il doit pouvoir réaliser une isolation spatiale et temporelle

parfaite des différentes partitions. Par exemple, une partition qui a un problème ou une corruption de donnée, ne mettra pas en défaut l'ensemble du système. L'industriel peut donc s'assurer et garantir que son satellite répondra toujours aux exigences internationales [Smi68] et fournira un service pendant la période garantie par celui-ci.

L'hyperviseur garantit le contrôle des parties logicielles et leurs isolations, tandis que l'utilisation d'un ordonnancement statique garantit la fiabilité de la partie logicielle au dessus de lui.

La maintenabilité est assurée par la possibilité de changer le plan d'ordonnancement en cas de défaillance d'une machine virtuelle par exemple.

Le gage de qualité donné par la sûreté de fonctionnement est pris très au sérieux, car nous voyons apparaître sur les satellites, de petits blocs électroniques dit *emergency* (urgence) [VBS05]. Ces composants permettent une redondance d'une petite partie du satellite afin d'en prendre le contrôle en cas de défauts majeurs sur le système ou l'hyperviseur. Cela permet, au minimum la récupération et la ré-initialisation du satellite en répondant aux critères de maintenabilité et de disponibilité. Cela peut également se limiter à la gestion des communications.

La disponibilité est garantie également par le fonctionnement des systèmes de préemptions de l'hyperviseurs au niveau des bus électroniques et au niveau logiciel. Elle garantie aussi le partage de l'ensemble des ressources processeurs aux différentes machines virtuelles qui compose le système.

La sécurité dans l'espace se limite, dans la mesure du possible, à éviter des collisions avec des systèmes tiers, car aucun humain n'est directement en contact avec les satellites dans l'espace.

L'utilisation d'hyperviseurs dans le domaine spatial apporte donc certains avantages. La vérification d'un tel système n'est plus globale comme pour chaque satellite précédent, mais partitionné. En effet, la vérification lors du développement peut, en raison de l'isolation de l'hyperviseur, se faire par partition. Cela fait également économiser du temps lorsque plusieurs satellites d'un modèle proche sont réalisés. L'augmentation de la partie logicielle se fait au détriment de la partie électronique. Cela limite donc les redondances auparavant systématiques.

Le développement et l'optimisation s'en voient facilités et permettent l'appel des sociétés conceptrices à des sociétés tierces pour le développement des parties non critiques telles que les partitions clients. Cela permet même d'intégrer directement le code du client et le faire participer financièrement et techniquement au développement de celui-ci. Un satellite ne prend plus que 3 ans de développement en moyenne.

L'utilisation de sous-traitants facilite, en revanche, le risque d'actes malveillants. Ainsi, il est plus facile pour une tierce personne de réaliser, par exemple, des vols de données sur des systèmes concurrents.

Une attaque sur de tels systèmes répond à des protocoles définis. En effet, nous commençons par évaluer les points nouveaux ou les points critiques du système. Nous utilisons les classes d'attaques déjà existantes, référencées ou non auprès du grand public, pour tenter de dégrader le système, puis, en fonction des résultats obtenus, nous attaquons des parties moins critiques afin de mesurer le niveau de nuisance et de quantifier les attaques critiques ayant aboutis. En fonction des résultats, nous mettons en oeuvre des contre-mesures (modification du système, ou modification des procédures de production) afin de réduire ou de supprimer l'intérêt de telles attaques. Les cas étudiés sont des cas rencontrés ou rencontrables sur des systèmes embarqués tels que les nôtres. Leur représentativité est donc pleinement réelle et non théorique. Une fois cette phase effectuée, nous pouvons, le cas échéant, tenter de créer de nouvelles classes d'attaques encore non-existantes.

3.5 Les conséquences en matière de sécurité

Dans cette section, nous énumérerons une partie des attaques dites malveillantes et leurs conséquences avec différents exemples. Des attaques malveillantes sont détectées dès le début de l'histoire spatiale mais restent limitées aux Etats. Ainsi, par exemple, un système militaire doit être en mesure de rester sous contrôle de ses opérateurs sol malgré un brouillage offensif non naturel. Il existe une adaptation continuelle aux nouvelles menaces, comme le démontre le système 21 de Thalès développé en 2015 ou le travail effectué en 2000 par cette même société [HBR00]. Si cette menace est réelle et prise en compte malgré le coût de développement, c'est qu'il existe ce type d'attaque. De plus, aujourd'hui, la généralisation des systèmes de communication et l'accès aux différentes parties matérielles et logicielles par l'ensemble de la population a pour conséquence une généralisation des nuisances sur satellites.

L'avènement d'internet permet à l'ensemble de la population d'accéder à de nombreuses informations et connaissances. Ainsi, l'observation de failles et bogues s'est généralisée et leurs exploitations se sont standardisées. Le domaine spatial n'échappe pas à ce phénomène. En effet, l'ensemble des plans de fabrications mécaniques, électroniques et informatiques de nombreux satellites sont accessibles de nos jours avec les systèmes amateurs de type cubesat [KBP+10] (micro-satellites de même catégorie qu'ANAIS [10]). Voici quelques données informatives trouvées sur internet :

— le cubesat est devenu un standard [PSTA01] ;
— il en existe une liste non exhaustive [NPST02] ;
— avec entre autre, celui de tokyo tech [NKS+03] ;
— il existe aussi certains prototypes tel qu'Open Orbiter [SKN+13] ;
— il existe également des plans [LHT+09, M+09] ;

Nous pouvons observer qu'il y a des points communs sur l'ensemble de ces satellites. La base étant commune, s'il existe une faille (ou bogue) sur un de ceux-ci et qu'elle est révélée, alors potentiellement l'ensemble des satellites de ce type est compromis. Ainsi, il ne faut pas ouvrir le code à des tierces parties sans confiance, car une personne de mauvaise intention peut vouloir prendre le contrôle, utilisé à des fins différentes pour détruire ou simplement perturber l'ensemble des satellites de même gamme.

Certains scénarios d'attaques sont retenus par les industriels. La possibilité, dans le domaine civil, qu'un client tente d'en perturber un autre présent sur le même satellite afin de diminuer les capacités ou de perturber les services fournis par son concurrent est une hypothèse prise très au sérieux. De même, dans le monde militaire, des pays qui collaborent sur un même satellite peuvent vouloir se perturber en raison de conflits d'intérêt sur un évènement en cours, l'un empêchant l'autre de recueillir les renseignements nécessaires pour une action armée, par exemple. Les quelques exemples militaires de ce document sont en majorité tirés de documents dé-classifiés de l'armée américaine expliquant sa politique de sécurisation des systèmes embarqués et l'explosion de l'utilisation de drone, principalement pilotés par satellite, comme l'atteste ces documents [CYC+09, CYC+13].

L'ensemble de ces scénarios généralistes sont possibles mais se traduisent par des actions techniques. Dans la suite de ce document, nous ne développons que les parties techniques, et la recherche scientifique associée.

10. www.geminisat.com

Nous observerons d'abord trois catégories d'attaques et leurs conséquences. Puis nous explorerons les failles physiques, logicielles et temporelles. Ce document se veut non exhaustif sur les attaques et n'offre que quelques exemples dont une contribution chiffrée sur système réel. Chaque partie commence d'abord par une observation générale puis approfondit la question sur les satellites modernes utilisant des hyperviseurs.

3.5.1 Plate-forme physique

Les attaques physiques nécessitent des connaissances spécifiques dans les domaines de l'électronique et des architectures des différents systèmes utilisés. Il existe deux catégories de ce type d'attaque : les attaques invasives, et les non-invasives.

Les attaques invasives

Les attaques invasives consistent à détruire les composants par procédés physiques ou chimiques afin de récupérer les schémas de routages et les informations relatives au fonctionnement d'un système. Elles déclenchent la rétro-ingénierie d'un système existant afin de comprendre les mécanismes de sécurité interne. Les manipulations ne peuvent être réalisées que par des experts de ces domaines. Cette attaque à la particularité d'être destructive pour le système et fort peut discrète, puisque qu'elle demande du matériel coûteux, réservé quasi-exclusivement aux Etats [KK99].

Les attaques non-invasives

Les attaques non-invasives se concentrent également sur l'électronique mais sont des techniques non-destructives. En effet, l'objectif est de perturber ou d'observer certaines particularités physiques sans altérer son fonctionnement. En voici trois exemples :

Attaque par canaux cachés : elle consiste à observer un paramètre physique extérieur aux différents composants électroniques, par exemple un bus de donnée entre un processeur et une mémoire externe. Les paramètres observés sont les variations du temps, du courant [Ngu11], de la tension ou du champ magnétique. Après analyse des traces, un expert est en mesure d'observer le type de transmission réalisé et peut aller jusqu'à subtiliser des clés secrètes sur le système [Gui07].

Attaques par conditions anormales : cette attaque consiste à perturber les entrées d'un système. L'injection de nouveaux signaux peut se faire avec la variation de la tension, du courant ou de la fréquence du signal d'entrée afin d'en observer les conséquences [Can09]. Il est aussi possible de perturber la température du composant sur une zone précise afin d'observer, entre autre, les difficultés temporelles d'un composant à répondre au signal d'entrée [NSIC+09]. Toutefois il existe sur les systèmes des détecteurs de conditions anormales qui sont réalisées à des fins de sûreté de fonctionnement. Ces attaques deviennent donc difficiles à concevoir.

Attaque par injection de fautes : Les attaques par injection de fautes sont parmi les plus répandues. Le point commun entre les différents composants est leur composition. En effet, la majeure partie de ceux-ci se compose de silicium. L'attaque se base sur les propriétés physiques de celui-ci. Il change de comportement électrique dans certaines conditions

parfaitement déterminées et connues. Pour influencer le silicium, il est possible d'utiliser des rayonnements tels que les ultraviolets, rayons EM, rayons X, lumière blanche, etc. Cela entraîne l'altération du contenu d'une mémoire par exemple. Cette altération peut être temporaire ou définitive [FFBM06]. Ce type d'attaque est également faisable avec des sondes et émetteurs électromagnétiques, afin d'écouter la variation de courant et de tension dans les transistors [Deh11].

La standardisation des parties électroniques peut augmenter le nombre de ce type d'acte malveillant. Néanmoins, il existe des contre-mesures efficaces contre ces phénomènes. Sur les satellites, il est tout de même difficilement envisageable de voir ce type d'attaque. Le seul scénario possible de nos jours serait de précéder l'attaque physique par un vol de l'un d'eux dans l'espace à l'aide d'un drone ou pendant sa phase de fabrication. Cette vision reste hypothétique et futuriste, surtout d'un point de vue budgétaire. Aucune attaque de ce type n'a été publiquement référencée dans le domaine spatial. Nous n'avons pas tenté d'attaque de ce type sur les plateformes satellites à notre disposition pour cette raison.

3.5.2 Logiciel et transmission

Les attaques logicielles sont aujourd'hui en phase avec les attaques sur les systèmes de transmissions, car les transmissions sont gérées par le logiciel. Les actes malveillants logiciels ont pour objectif d'injecter au sein du code des données ou des applications malicieuses dans le but de contourner ou détourner les applications installées de leurs rôles initiaux. Aussi, des secrets peuvent être dévoilés (code de sécurité, activité d'une autre partition que la sienne etc...). Il existe ainsi deux catégories d'attaques : celles exploitants des failles algorithmiques dues à un non-respect des spécifications ou des failles non précisées dans les spécifications elles-mêmes, et celles chargeant directement des applications malicieuses.

Les attaques algorithmiques

L'objectif des attaques algorithmiques est de trouver une faille, souvent absente des spécifications ou mal interprétée et implémentée. Seule une expertise et de l'expérience peut susciter une attaque efficace. Néanmoins, il existe des principes basiques qui permettent de réaliser ce type d'attaque. C'est le cas du *Fuzzing* et du *direct protocol attacks* [VF10]. Le *Fuzzing* consiste à envoyer un maximum de commandes afin d'obtenir des réponses inattendues du système. Le but est de détecter automatiquement des failles dans le système qui pourront êtres exploités plus tard. Quant au *direct protocol attacks*, il consiste à envoyer des commandes incohérentes avec l'état courant de la machine. Par exemple lire une mémoire après son effacement...

Il existe des techniques plus complexes qui conduisent à réaliser régulièrement des tentatives d'accès aux mémoires, caches, bus et autres périphériques pour observer un retour non prévu. Il arrive parfois que certaines informations retournées concernent une autre partition que celle ayant reçu le résultat (dans le cas d'un système non hypervisé ou mal implémenté). Les risques prit en compte sont donc des tentatives d'accès aux données de façon préméditée pour une action malveillante.

Notre contribution

Lors de la production de cette thèse, au sein du projet SOBAS, nous avons développé en amont ce type d'attaque afin de vérifier l'implémentation d'un système aérospatial utilisant l'hyperviseur XtratuM [MRPC10]. Cette hyperviseur était intégré sur un monocore Léon 3 de Gaisler [11]. L'objectif de cette démarche est de vérifier l'implémentation correcte du standard face aux attaques malveillantes.

L'ensemble des primitives systèmes et des commandes processeurs a été passé en revue. La particularité du système testé est l'intégration d'une MMU (*Management Memory Unit* ou unité de gestion de la mémoire) sécurisée. Elle provient d'un projet appelé "ESA project AO5829 Securely Partitioning Spacecraft Computing Resources". Il existe 67 hypercalls dont certains sont exécutables uniquement en mode privilégié (mode réservé uniquement à l'industriel fabriquant le satellite). Certains sont également désactivés ou en cours d'implémentation finale.

L'ensemble des hypercalls évalués résistent très biens aux attaques *Fuzzing* et *direct protocol attacks*. L'ensemble des résultats ne pouvant être révélés, en voici la conclusion. Une seule faille possible fut mise en évidence sur un hypercall en cours de développement. En effet, un des paramètres de celui-ci n'était pas protégé par un test (mis en commentaire) et pouvait s'appeler lui même indéfiniment. L'hypercall fautif était indisponible sur la version final du produit. Cela prouve que les attaques basiques telles que celles présentées au-dessus sont prisent en compte dès la phase de développement.

Des attaques plus complexes ont été mises en œuvre pour tenter de perturber cache, MMU, bus de donnés et mémoires. L'hyperviseur, simpliste, résiste de fait en partitionnant la mémoire en autant de sous-parties qu'il y a de partitions de manière fixe et non modifiable. Il efface également l'ensemble du contenu du cache entre chaque passage de nouvelles partitions. Cela évite un certain nombre de vols de données. Le bus de donnée partage l'accès entre les périphériques avec un *token* (jeton). Le principe du jeton est que chaque demandeur d'accès au bus attend son tour. Une fois celui-ci arrivé, il reçoit le jeton d'un contrôleur de bus. Il peut donc communiquer seul sur le bus car il détient celui-ci. Après sa communication il rend le jeton au contrôleur. Ce principe permet un accès fixe et invisible aux yeux des autres partitions de ce qui ce passe sur le bus.

L'hyperviseur testé prend également correctement en compte la gestion d'hypercalls privilégiés ou non. Ainsi une partition client n'a pas d'accès même temporaire quand bien même celle-ci se fait passer pour une partition privilégiée.

Les attaques par téléchargement de codes malicieux

Les attaques par téléchargement de code malicieux font entrer la chaîne de transmission en compte. Il n'est pas possible de se brancher directement via un câble sur le satellite, il faut donc corrompre le système de transmission. L'attaque la plus basique et la plus répandue est l'attaque par déni de service. Cela consiste à envoyer des données hertziennes, correctes ou non, en grand nombre afin de saturer le système [WWBD08]. Ce type d'attaque, également décelable [12], ne nuit pas directement à l'intégrité même du satellite mais uniquement à son moyen de communication. Elle est non destructive.

Une attaque plus évoluée est la tentative de substitution [RH10]. Une antenne sol proche de l'antenne de communication normale tente de se faire passer pour l'antenne de commande normale

11. www.gaisler.com
12. http ://www.ioactive.com/pdfs/ IOActive SATCOM Security WhitePaper.pdf

et envoie des données erronées afin de faire télécharger un code malicieux au sein du satellite. De nombreuses recherches ont été réalisées au cours de ces dernières années pour sécuriser le système. Ainsi les codes privilégiés ne sont pas modifiables et des systèmes sont mis en place pour observer une défaillance ou un comportement non programmé des partitions client.

Notre contribution

Sur le système spatial utilisé par le projet SOBAS, nous étions en charge de réaliser une partition client déjà infectée par un code malicieux. Celui-ci a pu intervenir lors de la conception et être dissimulé au sein du code de fonctionnement normal, comme nous avons pu le démontrer, ou potentiellement être téléchargé à cause d'une faille dans l'implémentation d'une partition client par exemple. Nous avons démontré les limites d'une telle approche avec la nécessité d'associer des failles physiques aux failles logicielles afin de réellement perturber le système.

L'objectif d'une partition client est de perturber ces voisines ou de tenter de prendre le contrôle sur celles-ci. Deux attaques furent particulièrement réussies, dont l'une d'elle sera explicitée dans la suite du document. L'autre permettait la prise de contrôle exclusive du satellite par une partition client qui mettait en défaut l'hyperviseur. En effet, les tâches exécutées sur la partition client étaient non préemptives. L'hyperviseur ne pouvaient donc plus reprendre la main pour faire travailler les autres partitions. Le code impliqué fut intégralement retiré et nous nous somme assurés de son absence dans d'autres parties du code.

Il est toujours possible de trouver des failles sur les systèmes, mais il faut empêcher que l'une d'elle soit problématique ou qu'elles ne puissent êtres associées pour créer un problème. Toutes les attaques ne doivent pas être supprimées. Certaines, connues et identifiées peuvent service à la détection de tentatives d'intrusions. Nous avons implémenté certains de ces mécanismes. L'isolation spatiale et temporelle reste le meilleur atout de ces nouveaux satellites. Mais qu'en est-t-il réellement de l'isolation temporelle ?

3.5.3 Temporel

Les attaques temporelles regroupent toutes les attaques faisant intervenir le temps. L'objectif de ces attaques est le même que les autres mais se base aussi bien sur l'électronique que l'informatique. L'objectif premier est de déterminer le temps. Une fois ce paramètre fixé, les attaques consistent à observer bus, ordonnancement, temps d'exécution de la mémoire, du cache etc... L'observation peut permettre la compréhension du système et potentiellement aider à voler des clés secrètes.

Dans un système temps-réel comme les satellites, cette attaque est l'une des plus efficaces. En effet, il est facile et répandu de se défendre contre les autres types d'attaques, même par défaut, mais le fait d'être dépendant du temps rend la situation critique. Prenons un exemple simple. Imaginons une partition malveillante. Elle observe sa capacité à déplacer un périphérique tel qu'une antenne, et se rend compte qu'une partition voisine en a systématiquement besoin après elle. Pour perturber la partition suivante, il suffit, par exemple de mettre l'antenne en butée et obliger l'autre système à la repositionner en perdant un temps précieux à cette tâche.

Les attaques sur système temps-réel peuvent aussi avoir pour seul objectif de casser l'ordonnancement et faire rater des échéances à l'hyperviseur ou aux partitions au-dessus.

Mais les attaques temporelles peuvent êtres bien plus efficaces et aller jusqu'à voler ces informations. Il existe une attaque nommée *covert channel* [Mil87]. Cette attaque consiste à utiliser le

temps comme un système de communication. Nous avons pu approfondir cette attaque sur système réel.

Notre contribution

Nos travaux se sont basés sur l'hyperviseur XtratuM avec un processeur Léon3. A la suite des autres attaques déjà réalisées sur ce système dans le projet SOBAS, nous avons pu développer différentes attaques temporelles dont le *covert channel*. Dans notre cas, le *covert channel* consiste à faire communiquer deux partitions entre elles sans que cela passe par les communications "normales" et ne soit "détectable facilement". Pour rappel, deux partitions ne peuvent communiquer entre elles que si celles-ci, lors de la phase de développement du satellite, ont obtenues ce privilège fixe et non modifiable en vol. Si ce n'est pas le cas, elles doivent êtres isolées spatialement et temporellement. Pour que deux partitions puissent communiquer en toute "illégalité", il faut donc créer un nouveau canal de communication. Pour nous, ce sera le temps.

Nous prenons en compte deux scénarios pour nos tests. Le premier est la communication entre deux partitions malveillantes. Elles ont interdiction de parler entre elles par hypercalls et vont tenter de parler entre elles tout de même pour s'échanger des données en *fullduplex* (dans les deux sens). Le deuxième scénario proviendrait d'une faille lors du développement du satellite dans une partition saine, ou du téléchargement d'un bout de code malveillant sur cette partition. Celle-ci ne se rendrait pas compte de la fuite d'information qui serait collectée soigneusement par la partition réellement malveillante. Le canal de communication est uni-directionnel (*Halfduplex*). Ainsi la partition émettrice est dans un cas complice de la partition réceptrice et dans l'autre cas une victime.

Le premier cas correspond à deux partitions client et le deuxième cas fait intervenir le risque d'infection d'une partition privilégiée appartenant à l'industriel concepteur.

L'objectif de l'attaque

L'objectif de l'attaque se résume en 3 points :

1. déterminer le temps d'exécution de la partition malveillante et calibrer son horloge ;
2. tenter de faire déborder temporellement de son temps d'exécution normal la partition victime ou complice ;
3. récupérer cette information avec la partition malveillante.

L'expérimentation

La première phase d'expérimentation est de fixer une référence temporelle. Nous avions pour cela deux possibilités :

— Une logicielle avec un hypercall nommer "XM_get_time" qui indique en microseconde le temps exécuté depuis la dernière ré-initialisation du processeur. L'interêt de cet hypercall, outre son existence, est d'être accessible par toutes les partitions du système ;
— Une physique avec l'utilisation de l'horloge électrique utilisé par le processeur cadencé à 50MHz soit une précision de l'ordre de 20ns.

Le fait de pouvoir avoir deux valeurs temporelles pour une partition client malveillante est un avantage certain. Il permet de calibrer et vérifier son attaque dans le temps. Lors des expérimentations, une troisième valeur a été prise afin de vérifier à l'aide d'un oscilloscope la cohérence des valeurs reçues. En l'absence de fonctions permettant d'indiquer le temps, il reste possible de générer soi-même celle-ci.

La deuxième phase est de faire déborder le temps d'une partition. Ainsi, nous n'avons pas eu besoin d'installer de machine virtuelle spécifique. Seul un petit logiciel C suffit. Nous avons ensuite testé temporellement l'ensemble des hypercalls et instructions logicielles afin de déterminer une instruction qui prendrait plus de temps que prévu. Les hypercalls ont été correctement implémentés et ils ne débordent pas de l'enveloppe de temps qui leur est allouée dans la documentation. En revanche, une instruction processeur, laissée volontairement accessible par tout type de partition permet de faire une remise à zéro du cache. Cette instruction ne prend qu'un cycle processeur, mais en prend un par ligne de cache à effacer (il y en a 512 de 32 octets). Ainsi, il existe une impossibilité d'accès au cache durant cette opération, soit 2x5.120 microsecondes.

Il est en théorie facile de voir une impossibilité d'accès du cache mais l'hyperviseur réinitialise le cache avant chaque partition. Il n'est donc pas possible de déterminer un fait volontaire de la partition de communiquer ou d'observer le phénomène normale de l'hyperviseur. En revanche, l'hyperviseur souhaitant systématiquement vider le cache doit attendre que celui-ci soit disponible pour réaliser son instruction. Ainsi, l'hyperviseur attend de pouvoir reprendre la main un petit peu plus longtemps.

Le décalage créé par l'attente de l'hyperviseur est visible temporellement, car ce décalage est supérieur à 2x5 microsecondes.

La troisième étape est de réaliser le canal de communication tel que montré sur 3.4. Dans l'expérience numéro 1, les deux partitions sont malveillantes, ainsi le code permettant l'observation du temps n'est pas dissimulé. Cela a pour but de démontrer uniquement la faisabilité. A des fins de réalisme, une troisième partition est rajoutée afin de créer un plan d'ordonnancement de machines virtuelles réalistes.

L'expérimentation 1

Les deux partitions malveillantes tentent dans un premier temps de comprendre leur ordonnancement sans perturbation volontaire de leur part. Après avoir appris et détecté ce plan par mesure de temps, elles se synchronisent en perturbant le cache à la fin de chacune de leur échéance. Le canal de communication est ainsi crée, et les deux partitions connaissent réciproquement leurs décalages respectifs sur le cache.

La fréquence du processeur étant de 50MHz, nous devinons que le taux de transfert de données est fortement limité dans le temps. Ainsi, seul 1 bit utile est exécuté pour chaque terminaison d'une partition malveillante. (Pas de perturbation du cache = '0', perturbation du cache = '1' par exemple). Le taux de transfert est en l'état ridicule, donc négligeable.

L'objectif suivant est d'augmenter le débit. L'intérêt est d'avoir deux mesures du temps observables, ainsi, nous pouvons augmenter la précision à 40ns près.

La précision est revue à 60ns afin de garantir une bonne lecture des données lors de notre exploitation physique. Cela veut dire que le temps occasionné par le décalage de l'instruction de la mémoire cache est observable par pas de 60ns avec un pas maximal de 2x5 microsecondes. Il y a

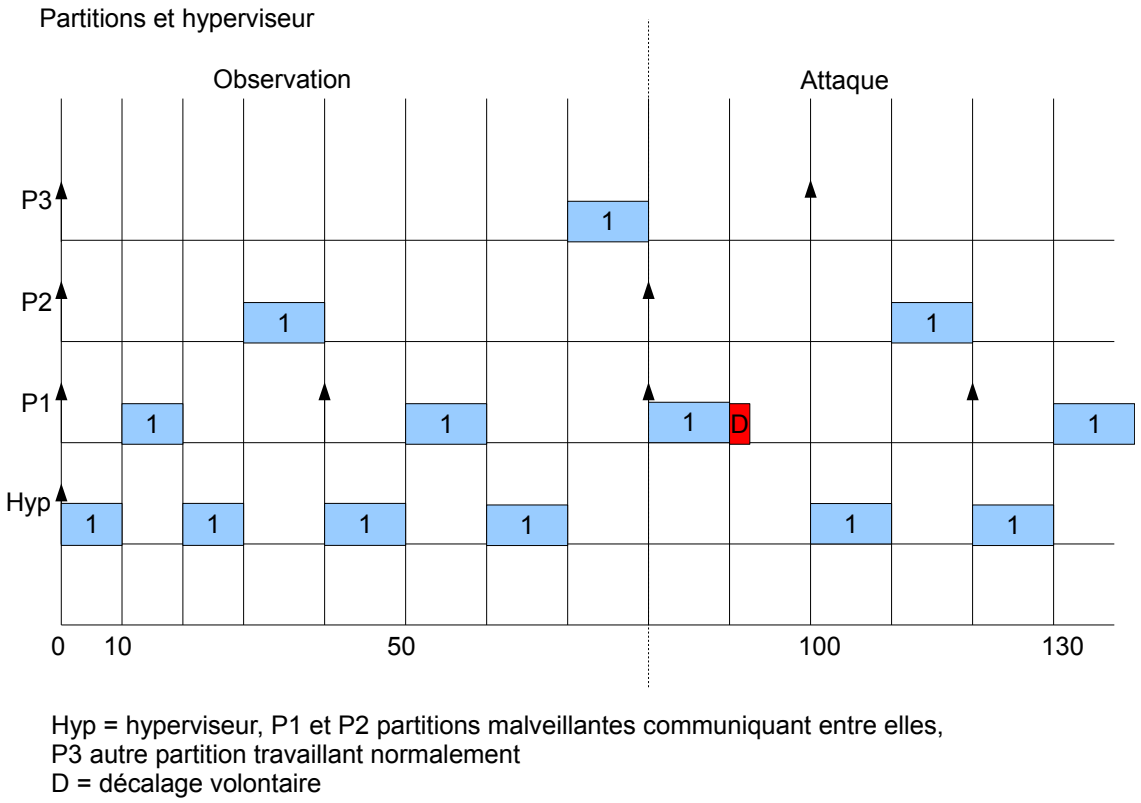

Fig. 3.4 – Observation puis attaque de la partition 1 en communiquant avec la partition 2

donc 2x85 valeurs possibles par envoi d'information (en plus de celle ou il n'y a pas de décalage). Il est donc possible de communiquer 2x6 bits utiles complets à la fin de chaque exécution d'une partition. Soit les 62 combinaisons ASCII de l'alphabet minuscule, majuscule et les nombres de 0 à 9. Il est donc maintenant possible à chaque changement de partition de faire passer une information claire entre les deux partitions malveillantes.

Le débit, par rapport au type de système, est très important. Ainsi, pour un tel système, une journée représente le vol de 14.688 Méga-Octets de données utiles, car la partition donne les informations à chaque seconde. Le débit pourrait être fortement augmenté en cas d'amélioration de la précision de l'observation, ou d'un ordonnancement plus clément envers la partition malveillante fournissant les données car l'ensemble des mémoires disponibles sur le système (hyperviseur compris) est en comparaison de 16 Méga-Octets. La totalité des données satellites pourrait (dans le cas où la partition infectée aurait accès à la totalité des mémoires, ce qui n'est jamais le cas) être volée en une journée au plus.

Les données clients sont fortement limités à leurs besoins, ainsi nous tentons d'infecter une partition de vol avec des informations et privilèges étendus.

L'expérimentation 2

L'expérience numéro 2 est parvenue à dissimuler le code dans une partition privilégiée, ou le canal de communication est unidirectionnel. La technique reste la même, seule la partition malveillante mesure le temps afin de récupérer les bits utiles. Le débit trouvé précédemment ne semble pas important mais il est considéré comme suffisamment sérieux pour être surveillé et

modifié. Ainsi la documentation technique du constructeur confirme ce type d'attaque et a mis en place des systèmes pour en limiter le risque. Le plus simple, mais aussi le plus coûteux en temps est l'utilisation d'un délai d'attente obligatoire au début de chaque démarrage de l'hyperviseur afin d'empêcher ce type de lecture. Le taux réel de communication dépend fortement du plan d'ordonnancement et de la répétition des partitions concernées.

L'une de nos plus grandes difficultés est de pouvoir mesurer efficacement le temps de chaque hypercall sans avoir accès à l'ensemble du code source du système. Nous pouvons conclure que ce type d'attaque est réalisable et peut mettre en défaut un satellite. La connaissance du temps est donc primordiale sur ce type de système.

3.6 Conclusion

Ce chapitre nous a permis de mieux comprendre les évolutions du domaine des satellites, et d'appréhender la mise en place des hyperviseurs. Nous avons pu conclure sur les modifications qu'ils apportent et plus particulièrement sur la sécurité de ceux-ci en observant une attaque réalisée sur système réel. Celle-ci met en défaut l'ensemble du système et permet de voler de nombreuses données rapidement. Ainsi, la plus grande faille ne vient pas de l'implémentation mais d'un mauvaise appréhension du temps. Le chapitre suivant aura pour but d'expliquer ce qu'il est nécessaire de connaître pour bien évaluer le pire temps d'exécution.

Deuxième partie

La mesure du temps d'exécution

Chapitre 4

Détection dynamique adaptée au problème industriel

Sommaire

Dans ce chapitre nous présentons l'état de l'art spécifique à la première contribution majeure. Nous développons dans cette première contribution les moyens techniques et scientifiques possibles pour mesurer le pire temps d'exécution d'une tâche dans un système où le code source n'est pas accessible (par obligation ou souhait). Nous étudions plus précisément dans ce chapitre les deux techniques qui permettent de s'approcher du pire temps d'exécution et nous les comparons en fonction de notre problématique. Nous concluons par la justification de notre choix.

4.1 Introduction

Nous avons observé dans le chapitre précédent (chapitre 3) qu'il est possible de mettre en défaut un système temps-réel en utilisant le débordement temporel des tâches pour déstabiliser l'ordonnancement et faire passer de l'information cachée. Ce débordement n'est possible qu'en cas d'erreur d'estimation du pire temps d'exécution d'une tâche, c'est-à-dire du temps maximal prit par une tâche pour s'exécuter complètement (en prenant en compte tout les retards possibles tels que le délais des bus de communications etc...). Ce pire temps est aussi appelé WCET pour *Worst Case Execution Time* en anglais.

La description complète du problème relatif au WCET est apportée entre autre par Wilhelm [WEE+]. Celui-ci explique les influences possibles d'un système réel sur le WCET et la difficulté à estimer celui-ci. L'exécution d'une tâche est assujettie à la taille de son code à effectuer mais aussi aux branchements et autres demandes extérieures d'informations. Par exemple, si le système souhaite accéder à un périphérique avant de l'enregistrer dans une mémoire, alors le WCET devra prendre en compte le temps pris par le bus de communication pour accéder au périphérique ainsi que le temps de réponse de celui-ci. Ce sera également le cas pour la mémoire. Le temps de réponse du périphérique, de la mémoire ou du bus dépend, à l'instant t, de leurs taux d'occupations respectifs.

Il n'est pas possible de connaître le WCET réel d'un système car celui-ci est trop complexe à déterminer. En effet, il existe de nombreux paramètres à prendre en compte et l'espace de recherche est trop important pour être sûr que le cas observé ou calculé soit exactement le WCET. Ainsi nous parlons d'estimation ou de mesure du WCET lorsque nous parlons de chercher le WCET. Un des effets recherché se résume à trouver la valeur la plus proche possible du pire temps d'exécution réel par observation ou calcul.

Quelles solutions existent pour observer le pire temps d'exécution ?

Puschner en apporte certaines en 1989 [PU89]. Il existe deux grandes familles de techniques. La première permet une surestimation du WCET et est appelée analyse statique. La deuxième est une mesure sur le système réel, et s'appelle analyse dynamique. Cette dernière offre un minorant au WCET réel. Toute la difficulté est de trouver une valeur proche de la valeur réelle car une surestimation trop importante engendrerait un échec potentiel des tests de faisabilités et un surdimensionnement des ressources matérielles nécessaires. Une sous-estimation permettrait le type d'attaque précédemment observées au chapitre 3 et nuirait à la sûreté de fonctionnement puisque cela mettrait naturellement en défaut l'ordonnanceur. La figure 4.1 montre l'écart entre les valeurs estimées ou calculées et le WCET réel. Elle indique plus précisément l'intérêt des deux techniques (statique et dynamique). Plus l'algorithme est évolué (et complexe) et plus les valeurs obtenues sont proches du WCET réel.

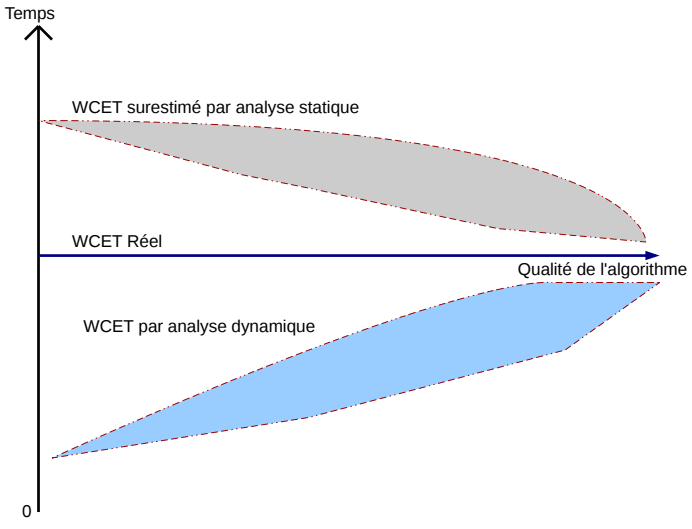

Fig. 4.1 – Observation de la différence entre WCET réel, estimé et calculé

Dans la section suivante, nous observons de manière plus détaillée les deux analyses proposées, ainsi qu'un ensemble de techniques utilisant simultanément les deux analyses. Cette observation permet, après rappel de notre problématique, de conclure sur la plus adaptée.

4.2 Statique versus dynamique

Nous détaillons dans cette section l'analyse statique, puis la dynamique et nous finissons par les techniques hybrides.

4.2.1 Le statique

L'analyse statique est une technique qui donne le moyen de calculer le pire temps d'exécution en observant les chemins de code possibles du système. L'objectif est de vérifier par calcul le temps possiblement pris par une fonction ou une tâche pour s'exécuter [IC]. Ainsi, il n'y a pas d'exécution lors de la première phase d'analyse du programme. Cette analyse conduit à l'obtention du WCET estimé.

Cette technique ce décompose en trois grandes parties :

L'analyse de flot qui détermine les chemins d'exécutions possibles en observant le code source ;

L'analyse de bas niveau qui offre le temps d'exécution d'une séquence d'instruction sans branchement sur un matériel précis (nous parlons ici de bloc élémentaire d'instructions du programme) ;

Le calcul qui cherche le pire chemin d'exécution possible et calcul sa longueur : chaque bloc élémentaire est associé à un poids dans le graph, valant son temps d'exécution.

Cette technique est la seule approche permettant de dépasser un niveau de sécurité supérieur à EAL4 selon les critères communs de sécurité des systèmes. En revanche cette analyse nécessite non seulement que l'ensemble des boucles soient bornées, mais aussi que celles-ci soient connues. Il est généralement demandé aux développeurs de l'indiquer dans le code source. Une difficulté supplémentaire est de déterminer les chemins faisables ou non dans le code. Il est donc plus aisé de travailler avec le code source directement. Chaque chemin non-faisable pris en compte réduit la précision de la mesure et donc la validité du résultat obtenu.

L'analyse de flot est complexe et nécessite une personne formée à la compréhension du domaine de l'analyse statique, car il faut déterminer par annotation : des constantes, des symboles, et les chemins infaisables ect... Plus précisément, il n'est pas nécessaire d'être spécialiste dans le domaine de l'analyse statique. Néanmoins, l'utilisation des outils d'analyse de code ajoute des contraintes, qui ne sont généralement pas comprises par les développeurs n'ayant jamais eu à calculer de WCET. Dans un programme standard, nous ne devons pas mettre d'annotations pour borner les boucles. Nous pouvons directement utiliser des pointeurs de fonctions sans indiquer l'espace des fonctions pointées, tout comme il n'est pas nécessaire de préciser au compilateur que certains chemins d'exécutions sont infaisables.

La phase de calcul peut utiliser une technique à base d'arbre avec analyse des blocs de bases. L'interface d'Heptane, un logiciel d'analyse statique développé par l'IRISA pour la mesure du WCET, nous le montre en figure 4.2.

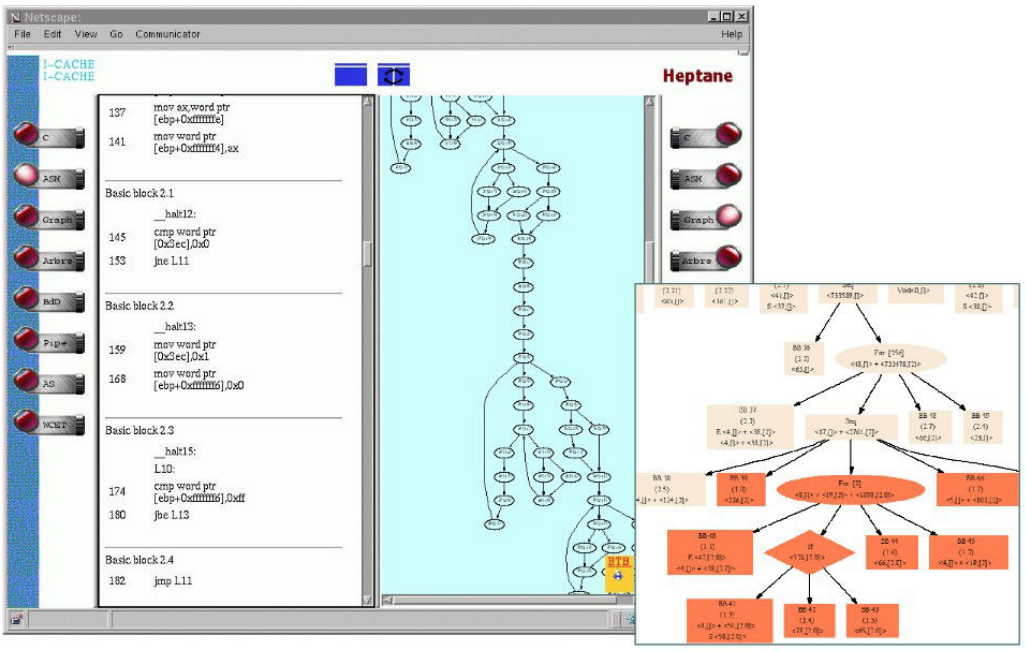

Fig. 4.2 – Une image prise en fonctionnement du logiciel Heptane

Les blocs de bases sont les plus petites séquences d'instructions processeurs exécutables sans interruption. Ces interruptions sont de type conditionnel ("if") ou par branchement. Une fois déterminé, les blocs de bases sont calculé de manière récursive afin de trouver les plus longs chemins temporels. Cette analyse permet une corrélation aisée entre le code et le système testé mais ne prend pas en compte certaines optimisations réalisées par les compilateurs. Il existe donc d'autres techniques de calculs.

Les différentes techniques d'analyses statiques sont implémentées, en majorité par des centres de recherches (français) tels que le logiciel Heptane (IRISA) [CP01] développé par l'institut de recherche de Rennes et visible à la figure 4.2, ou OTAWA (IRIT) [HP06] développé par l'institut de recherche de Toulouse.

La particularité de l'analyse statique, due à l'obtention d'un majorant du WCET rend possible la garantie des valeurs obtenues. C'est-à-dire que d'un point de vue sûreté de fonctionnement, nous sommes sûrs que la valeur réelle du WCET est inférieure à la borne estimée [KWHM]. En considérant l'expertise nécessaire au déploiement de cette technique, sans erreur, alors nous obtenons un niveau de confiance élevé dans les résultats obtenus.

Cette expertise est néanmoins difficile à mettre en œuvre sur des systèmes complexes avec caches, multi-cores etc... à cause du trop grand nombre de chemins d'exécutions valides et d'états possibles du matériel utilisé. De très nombreux travaux d'améliorations s'axent sur ce domaine, comme le rappelle l'article [WEE+] écrit par les nombreux chercheurs du domaine, à majorité d'analyse statique. Les méthodes hybrides, qui font appel à l'analyse dynamique pour améliorer les

résultats estimés sont une des principales avancées du domaine. Cela permet d'approcher au plus près du système réel, donc du WCET réel. Néanmoins cette technique subit l'impact du matériel et du logiciel sur les processeurs et plus particulièrement sur le WCET qui en découle [CP03].

4.2.2 Le dynamique

L'analyse dynamique est une technique réalisant l'observation d'un système réel lors de son fonctionnement. Cela consiste à utiliser des jeux de données d'entrées et de mesurer le temps d'exécution de celui-ci. La génération de jeux de données est réalisée par l'utilisateur, "à la main", ou de manière automatique. Soit l'ensemble des jeux d'entrées est effectué, soit il faut pouvoir définir un jeu d'entrée qui conduit de façon certaine au temps d'exécution le plus long.

Un outil idéal pourrait exécuter un programme pour chaque ensemble d'entrées possibles mais ceci n'est pas toujours possible. En pratique, la complexité du système et des programmes à analyser conduit à un nombre d'entrées possibles proche de l'infini. Seule la connaissance d'un sous-ensemble d'entrées pouvant amener au WCET suffit pourtant à calculer un temps d'exécution proche du WCET.

Cette méthode directe consiste à exécuter un programme logiciel sur un système réel puis à observer le temps d'exécution de celui-ci. La mesure du temps peut s'effectuer aussi bien au moyen de systèmes physiques externes (analyseurs logiques ou oscilloscope) ou internes (horloges processeurs). Elle peut également être effectuée de façon logicielle avec des horloges internes, moins précises et basées souvent sur l'horloge processeur. Il est également possible, lors du développement, lorsqu'il n'y a pas de système physique, par exemple, de n'avoir aucune horloge disponible et de devoir utiliser un simulateur. Le simulateur doit reproduire fidèlement un système matériel artificiel pour effectuer les mesures. La précision des valeurs trouvées par simulateurs est relative et demande un coût de développement certain.

Indépendamment du choix du système réel ou d'un simulateur, il faut réaliser un code exécutable et un jeu d'entrées pour le programme. Pour choisir le jeu de tests d'entrées, il existe trois possibilités :

— mesurer l'ensemble des jeux de tests d'entrées. Cela n'est possible qu'avec de petits programmes et systèmes à cause du nombre de cas possibles ;
— demander à l'utilisateur de fournir un jeu de tests qu'il considère pertinent, avec le risque qu'il se trompe ou soit corrompu ;
— de générer un jeu de tests de façon automatique, afin de déterminer le WCET. C'est sur cette dernière que nous nous concentrons.

Les méthodes dynamiques permettent donc, sans connaissances spécifiques du code source, de déterminer le temps d'exécution réel d'un système et de s'approcher du WCET. Elles sont régulièrement utilisées par les industriels pour valider des concepts, systèmes, ou sorties de production des produits. La technique fut néanmoins critiquée face au statique, car les résultats peuvent être néfaste, dans le cas de jeux de tests mal dimensionnés [BZK11]. Si l'analyse dynamique n'a pas de "preuve" que le temps observé soit le WCET, elle permet tout de même d'avoir un ordre d'idée des résultats et un oeil critique sur le système [Har07]. La solution de l'analyse dynamique peut être améliorée en associant les deux techniques : analyse statique et dynamique.

4.2.3 Les méthodes hybrides

L'association des deux techniques précédentes se nomme méthode hybride. L'objectif est de remplacer une partie de l'analyse statique en y intégrant l'analyse dynamique. La partie modifiée est celle du calcul. En effet, les blocs de bases calculés sont remplacés par des blocs de bases mesurés sur plate-forme réelle ou simulateur. Cela permet une estimation plus fiable et plus réaliste.

[WRKP05] proposent une approche hybride mixant analyse statique et dynamique pour calculer le WCET. L'analyse dynamique est utilisée pour mesurer sur plate-forme réelle le temps d'exécution du programme, et l'ensemble des jeux d'entrées qui le définisse est fourni par l'analyse statique qui, au regard du code source, limite les chemins à observer et amène potentiellement le WCET majorant au plus proche du WCET réel.

4.2.4 Conclusion

De nos jours, l'analyse statique est de plus en plus utilisée par les industriels, et plus particulièrement lors des phases de conception. Elle peut également intervenir lors de la phase de test et de validation des systèmes. Face aux défis des systèmes complexes, elle doit être associée aux techniques d'analyses dynamiques afin de garder une cohérence et améliorer ses résultats [KWRP]. Un article compare les deux techniques et montre les avantages et désavantages de chacune d'elles [MW98]. Nous voyons ici l'intérêt d'une association de ces deux méthodes. Il est donc possible de garantir un temps d'exécution sur un système temps réel critique et de se protéger, de fait, de défauts de fonctionnements tels que celui présenté au chapitre 3. Néanmoins, cette solution n'est pas magique. En effet, elle demande une grande expertise du système et reste complexe à mettre en œuvre dans l'industrie. Nous rappelons cette difficulté dans la section suivante.

4.3 Notre vision sur l'analyse dynamique et son intérêt face au statique

Dans cette section, nous développons la problématique de notre thèse et son implication sur la technique d'analyse que nous allons mettre en œuvre et améliorer pour déterminer le WCET d'un système complexe. Nous concluons sur la méthode adoptée.

4.3.1 Notre problématique

Le domaine des satellites, et plus généralement de l'industrie des systèmes embarqués, a évolué, comme nous avons pu le constater dans les deux premiers chapitres. L'évolution des méthodes de fabrication a permis de prédire le temps d'exécution d'une fonction à l'aide d'analyses statique, et de vérifier ces valeurs à l'aide d'analyse dynamiques ou hybrides. Il reste pourtant des difficultés lors de l'exploitation des produits finaux. Notre objectif est de comprendre pourquoi et d'améliorer les techniques précédentes en analysant et en résolvant certains verrous scientifiques.

Notre objectif n'est pas de déterminer quelle technique est la meilleur entre l'analyse statique et l'analyse dynamique mais d'améliorer au moins l'une d'elle afin de prendre en compte la complexité grandissante des systèmes embarqués. Les contraintes imposées par l'industriel sont autant techniques que scientifiques, commerciales ou financières.

L'exploitation des satellites a montré la volonté des entreprises d'externaliser la production vers des sociétés sous-traitantes. Cette extériorisation d'une partie de la fabrication est possible avec l'arrivée des hyperviseurs, comme nous l'avons étudié au chapitre 2. Cela a un impact sur la recherche du WCET. En effet, l'analyse statique effectuée ne prend pas en compte le développement d'autres parties du système par les autres sous-traitants. Au mieux, un simulateur est mis en place pour offrir plus de réalisme aux résultats. Il existe néanmoins de possibles failles dans ce processus et il arrive que certaines évaluations statiques soient faussées par un contexte matériel et/ou logiciel changeant.

Des incidents sont encore répertoriés malgré la mise en place en amont de l'analyse statique, il faut donc pouvoir contrôler les résultats trouvés de manière automatique sur ces systèmes complexes.

Dans cette thèse nous nous plaçons à deux niveaux. Au niveau de la société concevant le produit final, en l'occurrence un satellite, et au niveau du client réceptionnant ce satellite.

Le premier niveau implique que l'industriel vérifie la cohérence des résultats et détecte une action possiblement malveillante provenant d'un sous-traitant de façon automatique. La difficulté intervient dans la phase d'intégration. L'ensemble des codes des sous-traitants sont analysés sur système réel pour déceler des erreurs de calculs (surestimation ou sous-estimation) du WCET. Le temps d'analyse est importante. De plus, que l'impossibilité, pour des raisons de coûts et de temps, il est difficile de mettre un expert à la collecte de ce type d'information afin qu'il reprenne à nouveau le code source de chacun des sous-traitants pour réaliser statiquement l'analyse du WCET (sauf en cas d'erreur avérée).

Au niveau du client, la problématique est quasi-identique. La différence est le non-accès au code source. Comment un client peut-il, par mesure, faire confiance à l'industriel et vérifier que le code répond aux contraintes de sûreté de fonctionnement et de sécurité ?

La contrainte de non-accès au code source (associée à la volonté d'éviter une modification longue et fastidieuse du binaire) est la plus importante car la totalité de l'analyse statique se base dessus. Dans notre scénario de thèse, la méthode statique ne pourrait intervenir qu'en cas d'observation d'un problème réel dans les valeurs observées. Celle-ci serait complémentaire à l'analyse dynamique réalisée en amont (possiblement via les méthodes hybrides). En effet, l'analyse dynamique peut déterminer des valeurs de temps d'exécution en boite noire, c'est-à-dire sans accès au code source ou au binaire du système. Celui-ci répondrait donc, à première vue, à notre problème. De plus, contrairement à l'analyse dynamique, l'analyse statique qui considère le pire chemin d'exécution est sujette au problème de paramétrage des instructions élémentaires. Celles-ci n'ont pas d'incidence lors de la recherche des chemins mais lors de l'approximation de mesure du temps. Le statique teste le pire chemin mais le score de temps doit être comparé au dynamique pour être au plus près du réel. Nous prenons également en compte que ce n'est pas la même personne qui crée le logiciel et le manipule.

La méthode dynamique peut se résumer à la résolution d'un problème d'optimisation mathématique. En effet, nous cherchons à développer une optimisation du meilleur chemin, c'est-à-dire du chemin d'exécution réel qui induit le temps d'exécution le plus important pour une fonction donnée.

4.3.2 Problème d'optimisation

Nous n'avons donc pas, en théorie, directement accès au code source, mais au produit fini. L'analyse statique n'est donc pas, de fait, possible. Or l'analyse dynamique correspond à ce type de problème comme l'explique l'article suivant [VLW+12]. Mais comment contrôler l'existence d'erreur ou non dans la mesure du pire temps d'exécution dans ce cas ?

Les méthodes utilisées pour la recherche dynamique peuvent être empiriques [DD08]. Cela peut être le cas chez certains industriels. Une des solutions retenues habituellement est celle de la marge d'erreur. Ainsi, le WCET est déterminé tel qu'équivalent au temps d'exécution maximal trouver par méthode dynamique ajouté de 200% de marge d'erreur (représenté en figure 4.3). Sans garantie statique, cette technique hasardeuse et dangereuse donne soit un minorant, soit un majorant du WCET mais quasiment jamais le WCET réel. Cette technique ne permet pas non plus de résoudre notre problème et vérifier l'exactitude des valeurs précédemment trouvés par analyse statique. En effet, les jeux d'entrées ne sont pas toujours fournis par un spécialiste du système et les impacts des influences et phénomènes internes inhérents aux systèmes complexes ne sont pas pris en compte. Cette solution n'est donc pas fiable et apporte peu d'intérêt.

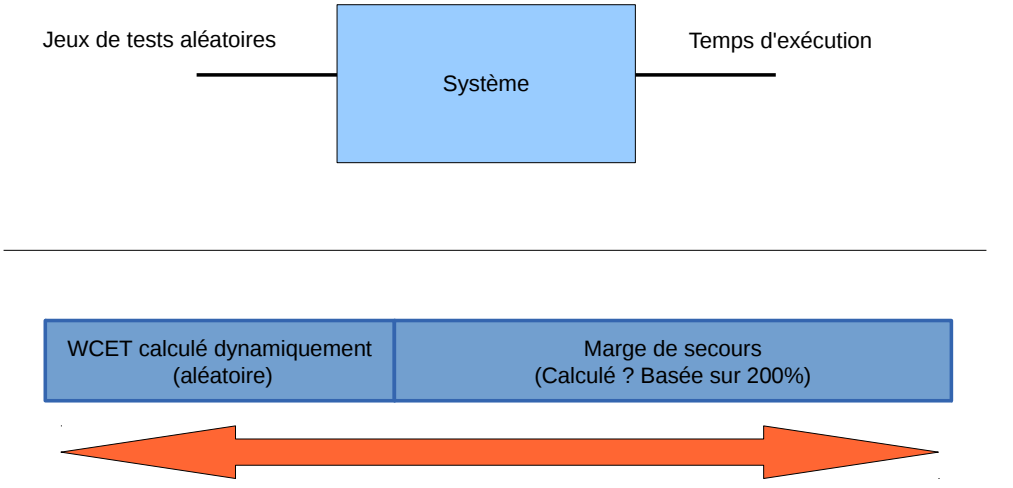

Fig. 4.3 – Mesures effectuées dans certaines industries pour vérifier le WCET

Les jeux d'entrées ont été par la suite amélioré par l'utilisation de benchmarks [GBEL10]. Ce sont des codes permettant de tester certaines caractéristiques d'un système afin d'en mesurer le temps (pour notre cas). Ceux-ci doivent, en théorie, limiter le nombre de chemins possibles en représentant chacune des étapes de "vie" d'un programme, telles que la compression, l'utilisation de boucle etc... Mais ces techniques se limitent à la connaissance qu'ont les experts réalisant les benchmarks.

Le but est de rendre cette démarche automatique. Elle nécessite de se tourner vers des techniques

dites heuristiques, dont l'utilisation sans ajout d'une marge d'erreur permettant de garantir la découverte d'un minorant du WCET ou le WCET lui-même. Dans un cas concret, cela revient à étudier de manière automatique et en boite noire les influences réalisées par le cache, la MMU, les effets multiprocesseurs etc... en plus des périphériques habituels tels que la mémoire.

Où est la difficulté ?

Pour chaque état du système, il y a un ou plusieurs chemins pouvant y amener. Les états possibles et par incidence les chemins qui y amènent sont proches de l'infini. Cela stipule qu'il est impossible de réaliser soi-même l'ensemble des chemins dans un temps fini. Or ce sont des systèmes produits par des industriels, et le temps de validation de ceux-ci doit être le plus court possible. Il faut donc limiter les chemins explorés au minimum et trouver les plus à même d'offrir un WCET. Il semble alors nécessaire de se reposer sur l'expérience d'un expert mais ceci n'est pas toujours efficace car chaque expert à son domaine et ne peut connaître que partiellement les autres. Les systèmes étant de plus en plus complexe, cela reste difficile à entrevoir.

En revanche, l'application d'une méthode d'optimisation s'inscrit particulièrement dans la problématique [MCS04]. En effet, nous cherchons mathématiquement à limiter l'espace de recherche et à optimiser celui-ci. C'est le cas de nombreux domaines. Ils utilisent tous des méthodes heuristiques, (figure 4.4) et plus particulièrement méta-heuristiques [ABHPW10]. La particularité des méta-heuristiques est de chercher, non pas une, mais plusieurs solutions répondant au problème. Cela permet, dans notre cas, de ne pas trouver le pire chemin mais l'ensemble des pires chemins possibles. Ne rechercher qu'un seul résultat nous obligerait à refaire l'ensemble des tests à chaque correction ou modification sur ce chemin. Le résultat serait également trop sélectif.

Une des solutions heuristiques utilisées est l'emploi d'algorithmes génétiques [Gol89]. Mais la modélisation de notre problème peut-elle justifier leur utilisation ?

4.4 Conclusion

Dans ce chapitre, nous avons étudié plus en détail le problème de notre première contribution, à savoir déterminer le pire temps d'exécution d'un ensemble de code sur système embarqué complexe. Nous sommes arrivés à la conclusion qu'il est nécessaire d'améliorer l'analyse dynamique afin de renforcer le contrôle effectué par les industriels et clients sur les systèmes satellites de conception moderne. Ces travaux s'inscrivent dans l'optique, à terme, d'améliorer également les systèmes hybrides utilisant aussi l'analyse statique. La contrainte majeure qui nous a amené à cette conclusion est l'absence de connaissance du code source et la possibilité de réaliser les mesures directement sur système réel. Dans le chapitre suivant nous déterminerons la méthode heuristique utilisée pour répondre à notre problème.

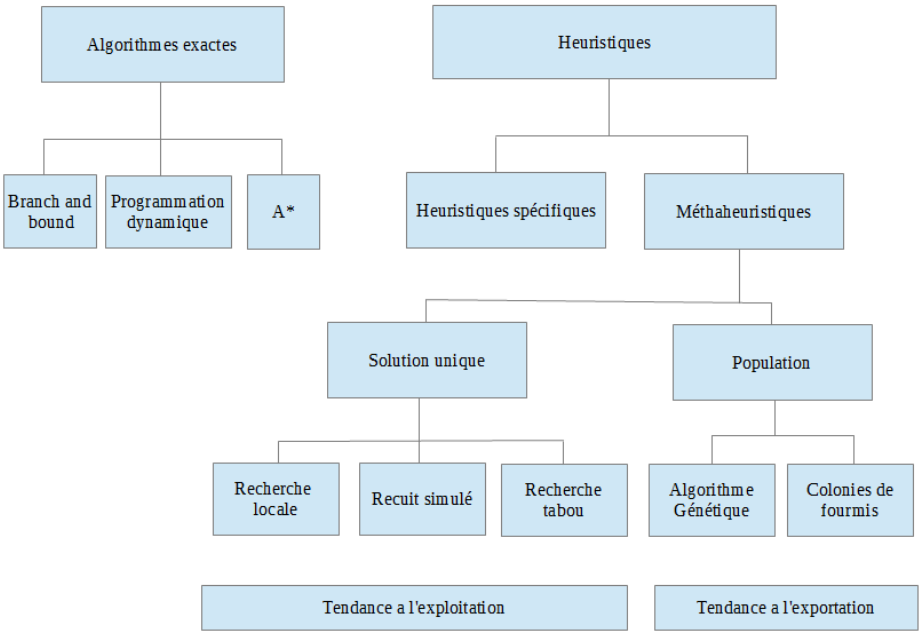

Fig. 4.4 – Les méthodes heuristiques

Chapitre 5

Modélisation et algorithme génétique

Sommaire

5.1 Présentation générale

Dans ce chapitre, nous développons la modélisation du problème de mesure du pire temps d'exécution d'une fonction dans un système complexe, sans accès au code source. Nous définissons, par la suite, notre approche et détaillons la solution dynamique retenue. Nous concluons sur l'intérêt de notre apport et préparons les résultats expérimentaux qui seront présentés au chapitre suivant.

5.2 Modélisation

Nous commençons par modéliser mathématiquement notre problématique, afin de définir quelle heuristique peut répondre au problème fixé. Nous déterminons dans un premier temps, la fonction objectif. Cette fonction résume le problème à étudier. Dans notre cas, la fonction objectif est de trouver le **temps d'exécution maximal** d'une fonction par rapport aux paramètres et au

contexte d'exécution du système. Par paramètre, nous entendons paramètre de la fonction testée, et par contexte, l'état du système. Le contexte sera plus détaillé dans la section suivante.

Les variables de décision, permettant de définir celles qui ont une influence sur notre choix sont donc :

— le ou les paramètres de la fonction à tester ;
— le contexte d'exécution du système.

Suite à cette base, nous y incluons un certain nombre de contraintes :

— Il existe un nombre fixe n de fonctions f_i possibles sur le système avec $i < n$. Chacune de ces fonctions f_i peut prendre un ensemble de paramètres qui influencent, eux aussi, l'évolution du système ;
— Chaque élément de base (que nous appellerons gène par la suite) représente une des n fonctions ;
— Il doit y avoir une successions de K élément de bases f_i ainsi que leurs paramètres associés, appelés C, (que nous appellerons chromosome par la suite) permettant de résoudre au mieux le problème ;
— Il existe un nombre maximal de fonctions exécutables consécutivement $< K$ (liée aux limites des systèmes réels et aux technologies utilisées) ;
— Pour chaque fonction il existe un nombre j de paramètres p avec, pour chacun d'eux, un intervalle de valeurs compris dans N (naturel) ;
— Il existe des contraintes de variables liées entre elles. Une variable liée est une variable utilisant un ou des paramètres p conçus ou modifiés par l'exécution préalable d'une autre fonction et générant une dépendance obligatoire entre elles (les variables liées sont de type fonctionnelles ou de données, telle que les fonctions d'initialisations ou les fonctions de création/destruction de tâches, en considérant qu'une tâche ne peut être détruite si elle n'existe pas). Ce problème relève plus d'une difficulté d'implémentation que d'un problème algorithmique ;
— L'évaluation de la séquence des fonctions (gènes) qui compose un chromosome donné est toujours envisagée sur un système dans l'état initial (déterministe), qui est l'état initial étant l'état obtenu par le système à la fin de son démarrage normal (*boot*) et en attente de la première fonction à exécuter ;
— Il existe X résultats composés de C successions d'éléments de bases, répondant au mieux au problème.

L'ensemble de ces contraintes sont la modélisation mathématique du problème cité au dessus. Ainsi le non-accès au code source est représenté par le fait qu'il n'est possible d'appeler que des fonctions du système avec leurs paramètres respectifs. Celles-ci sont bornées et non-nulles. Pour simplifier le discours nous considérons que la dernière fonction à tester est celle que nous souhaitons évaluer. Ce n'est pas un paramètre de notre algorithme mais une simplification du discours, amenant à une meilleure lisibilité.

Nous pouvons observer dans la définition des contraintes que nous prenons en compte l'existence d'un système réel avec des bornes maximales de fonctionnement liées aux technologies utilisées et à leurs limites physiques. Nous avons également pris en compte la nécessité de pouvoir rejouer l'ensemble des jeux de tests afin de rendre reproductible les résultats que nous obtenons [Joh01].

C'est le cas de l'avant dernière contrainte qui nous permet d'avoir toujours un état initial neutre au début de chaque test.

L'ensemble de la modélisation nous prouve que le problème est bien un problème d'optimisation, plus précisément, un problème méta-heuristique. La dernière contrainte nous permet de conclure sur l'intérêt qu'un algorithme du type algorithme génétique, offrant une population de résultat, répond au problème d'optimisation. En effet, l'espace de recherche est très vaste et particulièrement lié au nombre n de fonctions du système étudié. Il est borné par les limites technologiques du système testé et par l'ensemble des paramètres p de chaque fonction testée. Nous pouvons donc définir la forme de l'algorithme avec l'ensemble de ces contraintes.

Cela implique, tout de même, une vision nouvelle du contexte d'exécution d'un système.

5.3 Contexte : une nouvelle vue du fonctionnement d'un système

Nous définissons un état logiciel et physique d'un système à un instant t comme un contexte d'exécution (ou contexte). Voici la composition de celui-ci :

L'état physique : états de la mémoire, des périphériques, de la MMU, des caches, bus de données... ;

L'état logiciel : états internes de l'hyperviseur, et des partitions.

Le contexte est obtenu par l'exécution de fonctions successives amenant à un état précis, tel que représenté en figure 5.1. Dans la suite de notre thèse, les fonctions successives seront les fonctions de bases du processeur, c'est-à-dire les primitives systèmes. Celles-ci sont les plus simples et basiques du système. Néanmoins, le contexte peut être généré par n'importe quel type de fonction, donc pas seulement et spécifiquement les primitives.

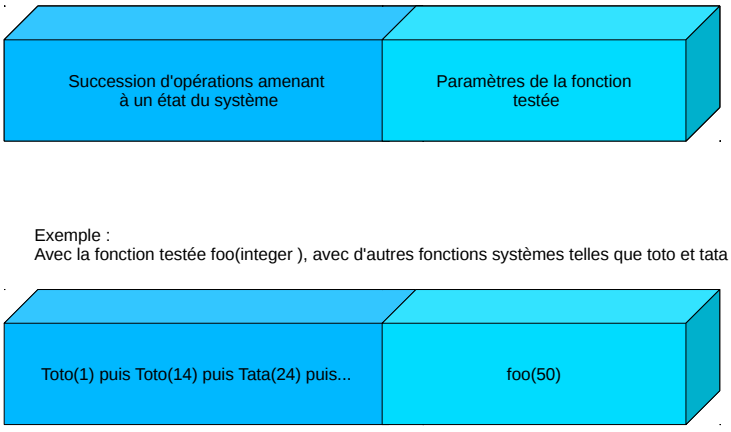

Fig. 5.1 – Vue d'un contexte suivi de la fonction à tester.

Le nombre de paramètres inclus dans un contexte (aussi bien matériel que logiciel) est potentiellement très important. Les paramètres correspondent aux paramètres des fonctions successivement exécutées. Leur énumération exhaustive est difficile, car dépendante des fonctions et de leurs nombres. L'influence du choix de ces paramètres est pourtant significative sur le temps d'exécution de la fonction testée. Dans un cas concret d'application, le temps d'exécution d'une fonction est influencé et dépend de l'état matériel et logiciel au moment de son exécution.

Dans le cas des hyperviseurs traités ici, le contexte d'exécution serait une succession de fonctions hyperviseurs et/ou processeurs. Celles-ci amèneraient à un état interne de l'hyperviseur. Pour des raisons de reproductibilité et de vitesse de calcul, les systèmes sont limités aux appels processeurs de bases, car une fonction hyperviseur est un ensemble d'appel à des fonctions de bases.

Les questions sont : *Comment intégrer cette nouvelle vision du contexte dans un algorithme méta-heuristique ? et plus particulièrement, peut-on utiliser un algorithme génétique ?* Nous tenterons de répondre à cette question dans la section suivante après avoir développé la particularité d'un algorithme génétique de manière générale.

5.4 Algorithme génétique

Nous avons choisi l'algorithme génétique [Dip] pour être en mesure d'intégrer le contexte d'exécution. En effet, celui-ci est déjà prédominant dans le domaine de la recherche automatique de tests [WSJE97] mais aussi dans le domaine de la recherche du WCET [Gro03]. L'algorithme génétique exploite une population de solutions partielles, et cherche à optimiser de grands espaces de recherches. Il est donc adapté à notre problématique.

L'algorithme génétique à été proposé par Goldberg [Gol89]. Cet algorithme évolutionnaire se base sur le principe de sélection des espèces définit par Darwin. Cette théorie de l'évolution étant principalement d'ordre biologique, nous y retrouvons des termes associés, ainsi l'ensemble des solutions qui peuvent résoudre le problème s'appelle population.

Les algorithmes génétiques sont fortement utilisés dans l'analyse dynamique de systèmes informatiques [Bou06, GÓ5, RHP99]. Une bonne définition de ceux-ci est donnée dans [WEE+]. Ainsi il est utilisé dans l'ordonnancement de systèmes robustes et permet de trouver le meilleur plan d'ordonnancement possible sur ces systèmes critiques. Il est aussi utilisé en économie pour essayer d'optimiser la courbe de la bourse par exemple [Val01].

Plus près de nos problématiques, celui-ci est également utilisé dans l'aéronautique [DG] à des fins de gestion du trafique aérien.

L'algorithme génétique a donc aussi été étudié dans le domaine du WCET [Gol89]. Celui-ci est encore amélioré de nos jours [Mar12].

Un algorithme génétique agit donc sur une population de chromosomes comme suit :

1. chaque chromosome représente un état possible du système ;
2. durant l'exécution, les chromosomes sont sélectionnés à l'aide d'un *fitness*, puis recombinés afin de créer de nouveaux chromosomes par croisement et/ou par mutation ;
3. une grande valeur de *fitness* offre plus de chance à un chromosome d'être sélectionné, puis d'être recombiné avec d'autres chromosomes à fort *fitness* ;

4. le processus recommence jusqu'à satisfaction d'un critère d'arrêt.

Le principe de fonctionnement général d'un algorithme génétique est visible à la figure 5.2.

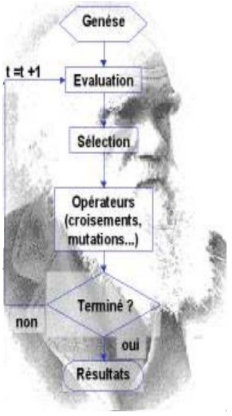

Fig. 5.2 – Principe de fonctionnement de l'algorithme génétique

L'algorithme génétique utilise des chromosomes contenus dans une population et la fait évoluer afin d'approcher au mieux le problème d'optimisation mathématique. Pour cela il existe plusieurs phases représentées en figure 5.2 et détaillées ici :

— Génération d'une population initiale : créé des chromosomes pouvant répondre au problème d'optimisation ;
— évaluation : détermine la pertinence de chaque chromosome pour répondre au problème mathématique ;
— Sélection : élimine les chromosomes les moins pertinents. Chaque chromosome est exécuté sur système réel. Le problème à optimiser (le WCET) est recherché, pour cela nous exécutons chaque chromosome et mesurons le temps d'exécution de la dernière primitive système. Un *fitness*, représentant l'intérêt du chromosome à répondre au problème d'optimisation, est fixé (pour nous il représente la valeur du temps d'exécution mesuré en réel) ;
— Croisement et mutation : associe les chromosomes entres eux. Les chromosomes sont sélectionnés préalablement par un tournoi en prenant deux paires d'individus choisis aléatoirement dans la population. L'individu ayant le meilleur *fitness* dans chaque paire survit. Les survivants sont sélectionnés pour générer de nouveaux chromosomes se rapprochant du problème à optimiser (croisement des deux chromosomes sélectionnés, avec une probabilité $p_{croisement}$ que cela se réalise, en créant deux fils) et créé une déviance (mutation en modifiant potentiellement chaque gène en prenant aléatoirement dans chaque chromosome un nouveau gène avec la probabilité que l'opération arrive, définie par $p_{mutation}$) ;
— Critère d'arrêt : fixe la règle spécifiant quand l'algorithme doit arrêter la recherche du chromosome optimal. L'algorithme s'arrête s'il n'y a pas d'évolution du fitness maximal de la population depuis un certain nombre d'itération. Si la première condition n'est pas vérifiée, l'algorithme s'arrête au bout d'un temps fixé par l'utilisateur.

Cet algorithme est déjà connu et il en existe de nombreuses variantes. Celui-ci a pourtant été minimisé avec l'arrivée de l'analyse statique dans le domaine de la recherche. Néanmoins, de manière générale, la recherche génétique et biologique avance et nous avons de nouvelles visions de celle-ci [ANLBHM00]. Ces solutions doivent êtres apportées à la recherche du WCET et peuvent nous permettre d'avancer vers des algorithmes génétiques plus pertinents et efficaces au sein d'un système complexe tel que le nôtre.

Afin de mieux comprendre ce qui peut être évolué et amélioré dans cette algorithme génétique, et où se trouve le verrou scientifique, nous allons observer l'approche actuelle des industriels dans le domaine. Nous verrons ensuite nos évolutions tendant à répondre à notre problématique.

5.4.1 Première approche : techniques industrielles existantes pour le contrôle des caractéristiques du satellite

Les industriels effectuent des tests d'évaluations du WCET en utilisant des *benchmarks*. Ceux-ci, propriétaires, ont été développés spécifiquement par certains experts pour tendre vers des exécutions "normales" ou "classiques" d'un satellite. Ainsi, les *benchmarks* de type *Mälardalen* [GBEL10] qui évaluent le WCET ne permettent pas d'observer une velléité d'acte malveillant ou un bogue particulier. Celui-ci se limite à offrir un temps d'exécution théorique à effectuer une tâche, sans garantie que cette valeur soit un pire temps d'exécution.

Les *benchmarks* de *Mälardalen* sont souvent utilisés mais principalement pour des systèmes simples et bare-métals. Les *benchmarks* sur systèmes complexes tels que les nôtres demanderaient un investissement certain, sans pour autant arriver à des conclusions sûres concernant la valeur du WCET d'un système. En revanche, pour des raisons de performances, ils sont très utiles.

Afin de voir l'impact d'un système complexe et de comprendre que les testent réalisés via *benchmarks* ne sont pas optimaux dans notre cas d'étude, nous avons choisi de les évaluer sur un système embarqué intermédiaire. Celui-ci est un MBDED [13] LPC1768 contenant un ARM Cortex-M3. C'est la plate-forme d'exécution matérielle. Pour sa part, la cible logicielle pour l'évaluation de notre proposition est construite sur FreeRTOS [Bar10] et d'une tâche effectuant des appels aux primitives du RTOS. Par rapport à l'objectif d'étudier un hyperviseur, cette solution matérielle et logicielle a l'avantage d'être peu coûteuse mais néanmoins de valider notre approche. Elle est également reproductible et est un système bien plus répandu que les systèmes spécifiques au domaine spatial.

Les primitives de FreeRTOS étudiées sont un sous-ensemble de 13 primitives :

Taches : create, delete, setPriority, suspend, resume ;

Ordonnanceur : getState ;

Semaphores : createBinary, createCounting, createMutex, createRecursiveMutex, delete ;

Mémoire : malloc, free.

Celles-ci sont utilisées par les *benchmarks*. Les *benchmarks* de *Mälardalen* [GBEL10] étudiés, visibles dans le tableau 5.1, sont sur de petits systèmes allant du Renesas H8300 au ARM9. Cette suite de *benchmarks* permet l'expérimentation des algorithmes de mesure du WCET dynamique. Dans ce papier, nous ne présentons les résultats que sur un sous-ensemble des *benchmarks* choisies pour mettre en évidence les propriétés des algorithmes génétiques et leurs défauts. Nous développons

13. www.mbed.com

plus particulièrement deux algorithmes génétiques et un algorithme aléatoire sur la plate-forme étudiée.

Techniquement, une connexion relie le MBED via USB à l'ordinateur réalisant les calculs de l'algorithme génétique. Un ordinateur exécute une version de l'algorithme génétique de Gross [Gro03] et demande via USB l'exécution de *benchmarks* avec les paramètres et contextes générés. Après deux exécutions dudit *benchmark*, la réponse du MBED est donnée sous la forme d'une trame comprenant le rappel du benchmark utilisé, la couverture de code dans le cas de notre apport explicité par la suite, ainsi que le temps d'exécution mesuré en interne par le MBED. La reproductibilité se limite donc aux capacités du MBED et aux fonctions testées.

Benchmark	S	L	N	A	B	U	Octets	LOC
adpcm		X					26852	879
compress		X	X	X			13411	508
duff	X	X				X	2374	86
matmult	X	X	X	X			3737	163
edn	X	X	X	X	X		10563	285

Table 5.1 – Tableau des *benchmarks* de *Mälardalen* utilisés

Légende : S = programme sans dépendance de flot ou de variable externe. L = contient des boucles. N = contient des boucles imbriqués. A = utilise des tableaux et/ou matrices. B = utilise des bits d'opérations. U = contient du code non structuré. Octets = taille du fichier de code source. LOC = lignes dans le code source.

L'évaluation sur la plate-forme physique est découpée en trois étapes :

— l'étude d'un algorithme de recherche aléatoire n'influençant que les paramètres de la fonction testée ;
— l'application de l'algorithme génétique de Gross tel que décrit en 2003, que nous appellerons SA pour Sans Apport dans la suite du document. L'algorithme génétique de Gross [Gro03] est la base des algorithmes génétiques que nous testons sur cette plate-forme. Celle-ci nous a permis de tester et valider nos apports détaillés par la suite ;
— la comparaison avec un apport de notre part sur celui-ci, que nous appellerons AA pour Avec Apport dans la suite du document.

L'algorithme aléatoire ne cherche pas à optimiser le problème et offre des valeurs pseudo-aléatoires à la fonction testée, ici un benchmark. L'algorithme génétique de Gross, lui, optimise le code et cherche à maximiser le temps d'exécution du système en trouvant des paramètres optimaux. L'algorithme génétique fonctionne tel que décrit dans 5.4.

Notre apport

Notre apport vient de l'idée du multi-critère. Nous avons testé l'intérêt de modifier l'algorithme génétique pour y intégrer un nouveau paramètre, c'est-à-dire celui de la couverture de code.

L'idée vient de travaux récents sur l'amélioration de la recherche du WCET par algorithme génétique. Une nouvelle technique basée sur la capture de l'état d'exécution par méthode dynamique durant l'évaluation est proposée pour améliorer la probabilité de découvrir des populations de chromosomes intéressants. Bate et Khan ont proposés [BK10] d'utiliser le multi-critère (erreur

de cache et nombre d'itération des boucles logicielles) combiné avec le temps d'exécution dans la fonction de *fitness* de l'algorithme génétique.

L'algorithme génétique modifié ici utilise la couverture de code, en supplément du temps d'exécution dans la fonction de *fitness*. L'idée est d'étendre l'espace de recherche en maximisant les chemins avec un fort impact sur le temps d'exécution.

Pargas *et al.* [PHP99] a proposé de guider l'optimisation avec la couverture de code et plus précisément la couverture de branche en utilisant un graphe de dépendance de contrôle. De plus, Whitten [Whi98] a utilisé l'algorithme génétique pour trouver un sous-ensemble de tests améliorés par la couverture de code qui emploie un *fitness* basé sur le temps d'exécution et l'observation des branches d'exécutions.

En outre, Bueno et Jino [BJ00] ont étudiés l'algorithme génétique pour identifier les chemins inacessibles d'un programme via une fonction de *fitness* basée sur le contrôle des informations de flots et de données. Les résultats sur l'analyse des *benchmarks* démontrent que l'algorithme génétique de base de Gross [Gro03] a obtenu 100% de la valeur maximale recherchée avec seulement 10% de branches infaisables comparativement à l'algorithme aléatoire qui n'a trouver que 70% de la valeur finale. Ces résultats peuvent être encourageant sur un système simple.

D'un point de vue électronique, Smith et al. [SBF97] propose de générer des tests qui vérifient l'implémentation électronique d'un microprocesseur en VHDL (langage de description électronique). Il a trouvé que, comparativement à une méthode aléatoire, la distribution offerte par l'algorithme génétique est stable et que le résultat obtenu est significativement supérieur à l'approche aléatoire.

A propos de l'utilisation de la couverture de code dans l'algorithme génétique pour la recherche du WCET d'une fonction logicielle, Tlili *et al.* ont proposé de distribuer l'algorithme avec une population présentant une couverture de code de haut niveau [TSWW06]. Mais cette technique n'utilise la couverture de code qu'au début de l'algorithme génétique, plus précisément dans la population initiale. Ceci ne permet que d'améliorer celui-ci au démarrage. Cette absence dans l'itération de l'algorithme génétique limite son impact.

Nous avons décidé, dans cette thèse, d'évoluer l'algorithme génétique de Gross [Gro03] avec la couverture de code dans la fonction de *fitness*. Plus précisément, nous décomposons le code en n blocs de bases avec $i \in [0, n[$. Un bloc de base, communément utilisé dans l'analyse statique, est défini ici comme une unité élémentaire de code avec un seul et unique point d'entrée et une sortie unique. Pour un chromosome C contenu dans la population P, chaque bloc de base est associé à un nombre (*integer* en anglais) $\Delta_i(c)$ représentant le nombre d'exécution de chaque bloc de base i durant l'évaluation du chromosome c. Cette couverture de code est reliée à chaque chromosome c au coté du temps d'exécution mesuré $t(c)$. L'intérêt de chaque chromosome est basé sur la fonction de *fitness* suivante :

$$cost(c) = \alpha * \frac{t(c)}{\max_{j \in P}\ t(j)} + \beta * \frac{\sum_{i=0}^{n} \Delta_i(c)}{\max_{j \in P}\ \sum_{i=0}^{n} \Delta_i(j)}$$

Cette technique ne répond pas exactement à notre problème puisque nous utilisons l'accès au code source. Néanmoins nous avons implémenté notre approche en modifiant un compilateur,

démarche plus aisée que la réécriture du fichier binaire. Ceci dit, nous pourrions tout de même modifier le fichier binaire et obtenir un résultat parfaitement identique. L'analyse du binaire est un cas envisageable dans le monde industriel. Cette modification coûteuse en temps et en hommes, spécifique à un matériel, n'a pas été réalisé dans cette thèse afin de se concentrer sur d'autres problématiques.

Nous constatons un biais expérimentale ou l'ensemble des chromosomes s'approchent, en temps d'execution, de la meme solution et les plus mauvais chromosomes représentent les moins bonnes variation du chromosome le plus elevé. Pour éviter cela, nous voulons maintenir des chromosomes, qui meme si ils sont moins bon, sont tres differents de part leur chemins d'exécutions. On ne cherche pas a garder celui qui obtiens la plus grande couverture de code mais celui qui crée une meilleur diversité.

Nous cherchons à optimiser le temps d'exécution mais la couverture de code permet une meilleure répartition des chromosomes observés dans l'espace de recherche. La première approche consistant à privilégier les grandes couvertures de code ne marchant en effet que pour le ou les bloques de bases qui ont le même temps d'exécution. Un grand bloque, dont le temps d'exécution est plus grand qu'un ensemble de petits bloques aurait été supprimé alors qu'il correspondait mieux à l'obtimisation du WCET mesuré. Cette approche est abandonné au profit d'un calcul de distance entre les couvertures de code par la suite. En effet, la couverture de code représente une forme primitive de contexte d'exécution d'un système. Une grande diversité de ceux-ci améliore grandement la chance de visiter une plus grande partie du système. Elle ne représente pas totalement le contexte d'exécution du système, mais permeet dans un cas ou le temps d'exécution est représenté de facon continu en fonction du contexte d'exécution, de converger plus rapidement vers un résultat. Dans le cas discret, il garantira une plus grande diversité de la population et limitera la probabilité d'obtenir un minimum local. Le paramètre a observé doit rester le temps d'exécution. Alhpa devant donc être supérieur ou égale à Beta.

L'objectif de la couverture de code n'est pas d'obtenir un chemin d'exécution plus grand, ou avec seulement un plus grand nombre de boucle effectués mais d'obtenir une plus grande diversité des chromosomes effectués. Ainsi nous calculons une distance entre les couvertures de codes des chromosomes plutot que de privilégier les chromosomes uniquement avec la taille de leur couverture de code.

Le couverture de code montre ces limites. En effet, la couverture de code ne nous permet pas de nous approcher d'un WCET mais nous permet de garantir que nous avons visité un maximum de chemins possibles afin de penser que nous nous sommes éloigné d'un WCET mesuré local. Cette vision de la couverture de code est une première approche de la forme du contexte d'exécution. Celle-ci donnera par la suite naissance à la question de l'intérêt de prendre un dump mémoire comme contexte d'exécution puis de nous amener à la réfléxion que nous présenterons par la suite. La couverture de code est une amélioration, qui, au vu des expérimentations, apporte une plus value restreinte à un cadre d'utilisation particulié et limité à de petits systèmes. Nénanmoins cette démarche scientifique nous a permis, de par les résultats encourageant de mieux appréhender la conception d'un contexte d'exécution.

5.4.2 Nos évaluations des techniques industrielles avec plate-forme complexe : contexte

Dans les *benchmarks* étudiés et visible dans le tableau 5.1 nous évaluons chaque chromosome deux fois. La première pour déterminer le temps d'exécution (sans couverture de code afin d'avoir le temps mesuré le plus correcte possible). La deuxième fois pour observer dynamiquement la couverture de code. Ces deux étapes sont nécessaires car l'observation de la couverture de code influencerait et modifierait les résultats temporels en exécutant son propre code.

Les deux algorithmes génétiques que nous étudions sont exécutés dix fois pour chacun des *benchmarks* que nous observons. Pour chacune des mesures, nous attendons le critère d'arrêt de l'algorithme génétique, c'est à dire lorsqu'il n'y a plus de changement temporel depuis un grand nombre d'itération (50).

Nous avons sélectionné un premier *benchmark* pour expliquer en détail les résultats temporels. Le *benchmark* "adpcm" a été choisi car il ne contient que des boucles et est extrêmement simple. Les résultats des différents algorithmes devraient donc être très proches. Le *benchmark* "compress" réalise quant à lui une opération bien connue dans les systèmes embarqués spatiaux afin de limiter la communication entre la terre et les satellites. Cet algorithme de compression montrera donc un cas proche du réel en incluant des matrices. Le programme "duff" prend quant à lui du code non structuré et offrira une différence accrue entre les algorithmes. "Matmult" accentuera les calculs matriciels et "edn" permettra l'apport de bits d'opérations.

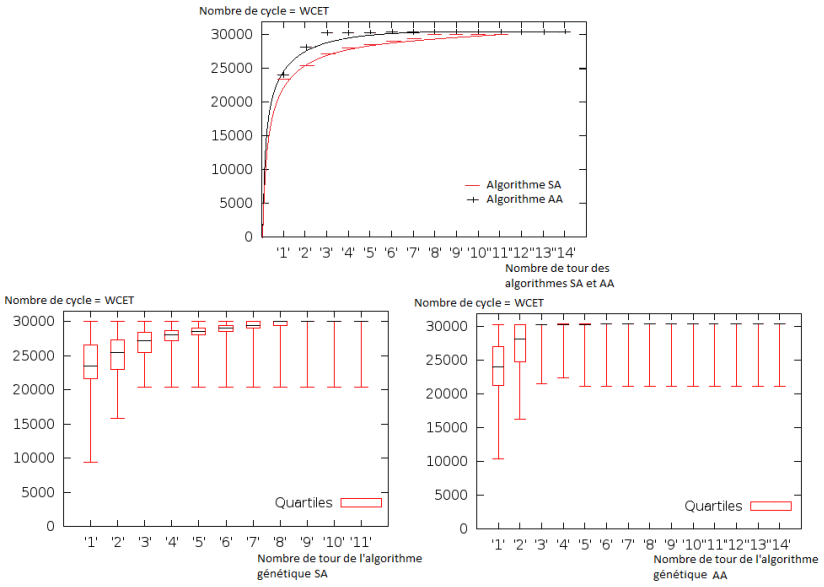

Fig. 5.3 – En haut : les 2 algorithmes sur ADPCM. A gauche : l'algorithme de Gross sur ADPCM en détail. A droite : notre algorithme sur ADPCM en détail.

La figure en haut du graphique 5.3 présente le WCET trouvé par les deux algorithmes génétiques sur le *benchmark* "adpcm". Les deux algorithmes ont été initialisés avec la même population initial, c'est-à-dire des chromosomes avec les mêmes paramètres. Nous pouvons observer que notre algorithme génétique utilisant la couverture de code trouve un meilleur résultat du pire temps

d'exécution et ce, plus rapidement que l'algorithme génétique de Gross. A la 4ème itération notre algorithme a déjà trouvé le résultat, alors qu'il faut attendre la 11ème itération pour voir un résultat identique pour l'algorithme de Gross. Nous pouvons également observer qu'après la 4ème itération de notre algorithme génétique, notre algorithme reste proche mais en-dessous de la valeur maximale du WCET mesurable. En effet, nous connaissons le WCET réel grâce aux concepteurs de celui-ci qui est disponible dans [Tan06].

Les Figures à gauche et droite du graphique 5.3 représentent respectivement l'algorithme de Gross et notre algorithme génétique utilisant la couverture de code dans le *fitness* pour le *benchmark* "adpcm". Nous y observons la répartition de leurs populations respectives avec les valeurs maximales, minimales, moyennes, ainsi que les bornes à 10 et 90%. C'est-à-dire que nous observons l'ensemble des valeurs de temps d'exécutions de l'ensemble des chromosomes que composent leurs populations. Les résultats confirment qu'une grande partie de la population générée par notre algorithme génétique est très proche de la valeur du WCET réel [Tan06] comparativement à l'algorithme de Gross. Nous constatons aussi la vitesse de convergence plus importante de notre algorithme avec des valeurs moyennes et minimales plus élevées.

Les résultats présentés dans la figure 5.4 représentent, quant à eux, le WCET mesuré des cinq *benchmarks* de *Mälardalen*. Ceux-ci sont exécutés dix fois sur les deux algorithmes génétiques.

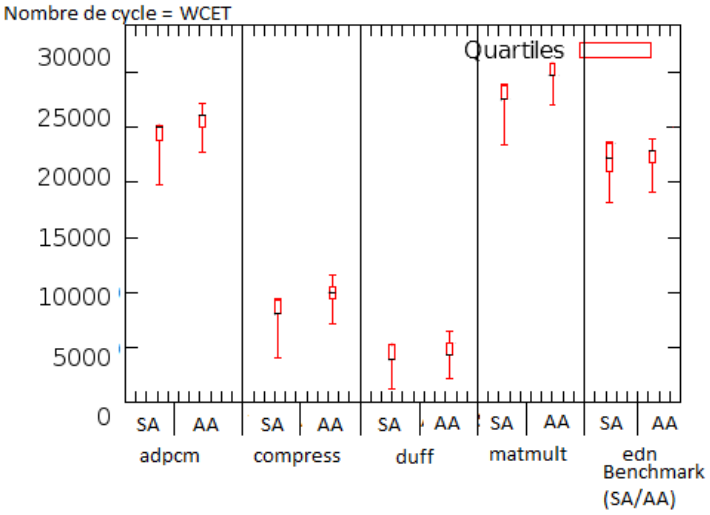

Fig. 5.4 – WCET final des algorithmes SA et AA sur les cinq *benchmarks* de *Mälardalen* utilisés

Nous observons les valeurs maximales, minimales, moyennes, ainsi que les bornes à 10 et 90% finales de l'ensemble des dix exécutions de chaque *benchmarks*. C'est-à-dire que nous observons les dix populations finales obtenues après exécution du critère d'arrêt de chaque algorithme génétique. L'ensemble de ces cinq *benchmarks* montre un intérêt certain à l'utilisation du critère de couverture de code. Le temps de convergence pour notre algorithme (non présenté ici) est inférieur à celui de Gross. Néanmoins, les valeurs restent proches. En revanche, l'implémentation de la couverture de code est non seulement complexe, mais aussi coûteuse.

Les valeurs de l'algorithme aléatoire ne sont pas présentés sur les graphes car les valeurs obtenues sont inférieures, en moyenne, de 10 à 30% par rapport aux algorithmes génétiques. Il n'y

a pas d'information majeure outre le fait que les algorithmes génétiques, de par leur fonction d'optimisation, obtiennent de meilleurs résultats et ce plus rapidement.

Conclusion

Pour conclure, cet apport montre que les algorithmes génétiques disponibles (celui de Gross est un des mieux documentés et a permis d'être repris pour les expérimentations) ne sont pas optimaux et peuvent être améliorés. Qui plus est, les nouvelles méthodes d'optimisation multi-critère sont réalisables. Néanmoins, pour notre problème satellitaire, notre solution ne peut être acceptable car elle prend en compte l'accès au code source, ce que nous ne souhaitons pas. L'idée du multi-critère avec la couverture de code reste donc viable (comme nous l'avons présenté à la conférence [GJP14]) mais ne correspond pas exactement aux évolutions souhaitées bien que la couverture de code représente à un instant t l'état du système. Les *benchmarks* de *Mälardalen* ne sont pas une fin en soi et peuvent être écartés au profit de recherches automatiques représentant mieux l'état du système. Il est cependant important de noter que notre premier apport est complémentaire au second, présenté dans le paragraphe suivant. Notre proposition limite grandement le risque de converger vers des maximums locaux et peut apporter cet avantage dans les travaux qui suivrons.

Nous avons montré précédemment que les *benchmarks* sont victimes du contexte du système et ne sont plus facilement applicables sur les systèmes complexes tels que ceux équipés par des systèmes multi-coeurs, des caches, MMU, etc... Concevoir un *benchmark* aujourd'hui sur de tels systèmes est particulièrement complexe et les algorithmes génétiques actuels ou légèrement évolués ne pourraient répondre correctement à nos attentes.

Nous choisissons donc de représenter l'état du système d'une autre manière afin de prendre en compte l'aspect boite noire.

Fort de nos premières analyses, et afin de mieux répondre à l'objectif que nous nous sommes fixés, nous nous posons plusieurs questions visant à améliorer notre réflexion. La première vise à répondre à la possibilité de réalisation concrète ainsi qu'à la technique qui donne les moyens d'atteindre notre objectif. Les questions suivantes définissent les difficultés techniques que nous pouvons rencontrer lors de la phase d'implémentation. Voici les questions que nous nous posons et qui seront détaillées tout au long de l'étude de notre deuxième apport :

1. Pouvons-nous définir un chromosome incluant le contexte ?
2. Qu'est ce qu'un espace de recherche et des états atteignables ? Comment les prendre en compte ?
3. Comment constituer une base de donnée d'états du système ? Est-ce utile ?
4. Comment concevoir une nouvelle population avec un algorithme prenant en compte le contexte ?
5. Comment ne pas modifier ou influencer notre système lors d'une mesure ?
6. Est-il possible de réaliser une parallélisation des mesures de temps ?
7. Comment générer la population initiale ?
8. Fort de ces questions, pouvons-nous mieux définir ce qu'est un système complexe ?
9. Pouvons-nous garantir une convergence finie de notre algorithme ?

5.4.3 Deuxième approche : prise en compte du contexte : notre apport

Notre vision de la représentation du contexte est différente afin de respecter l'ensemble des contraintes fixées. L'état d'un système à un instant donné peut être représenté par un ensemble de valeurs/variables. C'est ce que nous avons précédemment présenté. Notre nouvelle vision du contexte n'est pas de prendre comme information utile l'état du système à un instant précis mais l'ensemble des fonctions exécutées qui l'y amène, car Chaque exécution de fonction qui crée un nouvel état intermédiaire du système amène à l'état final.

La meilleure vision qui permet d'observer l'état d'un système à instant t est la succession d'étapes rendant possible l'arrivée à cet état. La succession d'étapes est la succession de fonctions logicielles depuis le démarrage du système ou sa ré-initialisation.

Le principe général de l'algorithme génétique voulu n'obtient pas de différence majeure par rapport à ceux déjà existants. Néanmoins, nous y apportons des problématiques nouvelles provenant de nos observations chez les industriels qui affinent notre choix :

— être entièrement reproductible ;
— n'accéder qu'aux primitives systèmes ;
— restreindre l'espace de recherche aux états atteignables ;
— modifier et n'influencer le système qu'au minimum pendant les mesures ;
— être parallélisable pour augmenter la vitesse de calcul ;
— gérer les problématiques d'un système complexe ;
— permettre une convergence dans un temps fini d'un résultat sur le WCET mesuré.

Cette nouvelle vision doit donc permettre l'intégration de nouveautés dans l'algorithme génétique telles que le multi-critère. Elle doit surtout prendre en compte la reproduction possible des réalisations [Joh01] (conserver cette propriété serait plus juste) et utiliser uniquement des fonctions de bases du système (primitives).

Impact sur notre algorithme génétique

La forme des chromosomes de celui-ci reprennent l'apparence générale du chromosome présenté figure 5.5. Mais il utilise uniquement une succession de primitives, la dernière primitive exécutée étant celle évaluée. L'état de départ de chaque chromosome est l'état initial du système. C'est-à-dire l'état obtenu par le système à la fin de son démarrage normal (*boot*) et en attente de la première fonction à exécuter. Un exemple est donné figure 5.6.

1.Pouvons nous définir un chromosome incluant le contexte ?

Afin de concevoir un chromosome répondant à l'intégration du contexte, nous soulevons deux problèmes scientifiques.

Le premier problème est que pour réaliser une succession de primitives, dans un cas réel, il n'est pas possible de prendre cette succession aléatoirement, car certaines primitives sont liées entre elles. De fait, il existe une dépendance entre les primitives. Il n'est pas possible de restituer, par exemple, un bloc mémoire qui n'a pas été alloué, ni possible de retirer de la mémoire une tâche qui n'existe pas. Pour mieux prendre en compte ces états improbables, inatteignables ou non-cohérents, nous ne générons pas aléatoirement la succession de primitive système. En revanche, il est possible de construire un ensemble de succession ou chaque nouvelle primitive exécutée dépend de l'ensemble des primitives déjà réalisées. Ce cas s'applique également pour la primitive à tester. Si la primitive

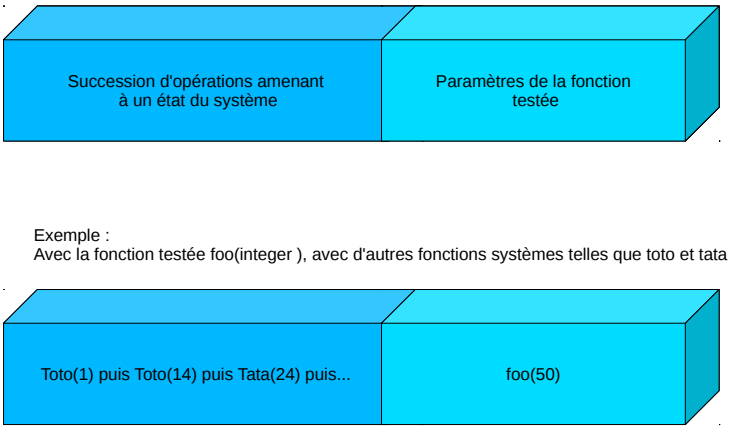

Fig. 5.5 – Vu d'un contexte suivi de la fonction à tester.

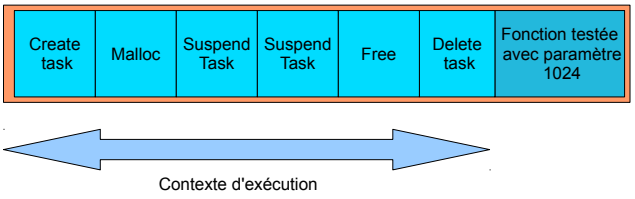

Fig. 5.6 – Vu d'un contexte généré par un ensemble de primitives suivi de la fonction à tester.

testée doit supprimer une tâche, il faut, de fait, qu'une tâche ait été créée et qu'elle existe déjà dans le chromosome généré. Notre idée est de générer des états cohérents, qui encodent dans le chromosome la suite des opérations/primitives y menant et permettant la sélection d'une variable liée uniquement si son complémentaire obligatoire existe précédemment.

Le second problème est l'existence pour chaque primitive de paramètres. Il faut définir pour chacun d'eux des valeurs réalisables et exécutables. Ces valeurs doivent se trouver dans un intervalle donné par la norme constructeur, ou correspondre, dans le cas des pointeurs par exemple, à une référence en mémoire centrale. Les paramètres, ne peuvent pas êtres choisis, à nouveau , de manière strictement aléatoire mais définis en fonction des autres paramètres précédemment sélectionnés dans la succession de primitives.

Nous proposons de prendre en considération, dans le cas des pointeurs pour une variable liée par exemple, non pas la valeur en tant que telle mais la $i_{ème}$*(modulo le nombre d'instruction)* valeur de paramètre résultant d'une opération antérieure (retournée par une fonction appelée, comme *create task* par exemple, qui retourne un pointeur vers une tâche).

Un chromosome C est donc représenté par une succession de primitives, incluant leurs paramètres respectifs, et se terminant toujours par la primitive à tester ainsi que par les valeurs de paramètres de cette dernière. Un chromosome C est composé d'une suite de primitives avec des paramètres :

$C = (p1(P1), p2(P2), p3(P3)...pk(Pk)pe(Pe))$ avec k nombre maximal d'appels de primitives avant la primitive évaluée et pe représentant la primitive à évaluer avec ses paramètres Pe. Chaque primitive appartient à l'ensemble des primitives du système mais également à l'ensemble des primitives acceptées et dépendantes des variables liées : $pi \in (malloc, free...)$ Chaque paramètre appartient à son espace acceptable, définit de la même manière : $Pi \in acceptable(p)$

Une limitation de nos travaux est le nombre d'états atteignables. Celui-ci dépend de la taille du chromosome, et n'est potentiellement pas suffisant pour atteindre des états pertinents. En effet, certains WCET sont obtenus grâce à contexte d'exécution élevé, donc mesurer un temps d'exécution avec un chromosome de taille faible serait donc incohérent et non productif.

2.Qu'est ce qu'un espace de recherche et des états atteignables ? Comment les prendre en compte ?

Le nouvel algorithme doit également restreindre l'espace de recherche aux états atteignables. L'espace de recherche est l'ensemble des états du système. Aussi bien les cas possibles (états réalisables par exécution successive de fonctions) que les cas impossibles (états du système ne pouvant être atteint par une succession d'exécution de fonctions, comme par exemple lorsqu'il existe deux branchements conditionnels. Si l'un de ces branchements est vrai, l'autre doit être faux et inversement, alors le cas impossible serait celui où les deux sont simultanément vrais, car aucune fonction ne pourrait réaliser un tel état du système). Ils représentent une infinité d'états du système. Ce que nous souhaitons, pour des raisons de rapidité mais aussi de sûreté de fonctionnement de notre système lors de nos tests, c'est de se limiter aux cas réalisables (les cas possibles) et donc ne mettant pas en défaut le système testé. C'est une problématique de disponibilité du système d'évaluation.

Ainsi, afin de garantir la stabilité du matériel pendant l'exécution de l'algorithme, notre proposition est d'exécuter les fonctions à évaluer dans leur espace normal de travail et de conserver les contextes obtenus du système. Un grand nombre de contextes est donc produit en exécutant les primitives testées (tout en respectant les bornes imposées par les concepteurs) afin de générer les points atteignables. L'algorithme de génération d'états atteignables, pourra, de plus, privilégier la sélection de contextes ($e1$) fortement dissemblables, ceci afin d'avoir un espace de recherche le plus diversifié possible. Un espace de recherche restreint nous amènerait automatiquement à un "maximum local", c'est-à-dire que les chromosomes d'une population tendraient à une même valeur inférieure au WCET, avec un chemin plus faible que le WCET réel.

Cet ensemble de contexte $e1$ finalement généré constituera une "base de donnée" d'états possibles pour le système lors de l'évaluation du temps d'exécution des primitives.

Un des apports de cette démarche est de rendre reproductible les pires temps d'exécutions observés [Joh01]. Si, par exemple, une primitive repose sur l'exploitation d'un générateur de nombres

pseudo-aléatoires, les variables globales constituant l'état interne du générateur pseudo-aléatoire vont êtres sauvegardées dans le contexte. Il suffira de restaurer ces variables globales pour reproduire le temps d'exécution.

3.Comment constituer une base de donnée d'états du système ? Est-ce utile ?

Une base de donnée qui permet la récupération des informations utiles générées lors de l'exécution est composée, non seulement, de l'ensemble des primitives exécutées depuis le démarrage (état initial) ou la ré-initialisation du système, mais également de la récupération de l'état des mémoires du système. En effet, la répartition des primitives et données sauvegardées lors de l'exécution dans la mémoire a également une influence sur le temps. Notre vision de la constitution d'une base de donnée qui permet la récupération des informations complètes d'exécution ne se limiterait pas uniquement à ces deux paramètres mais également aux conflits d'ordonnancement, de multi-coeur etc... si nous n'étions pas dans un système spatial. En effet, notre système à des particularités bien utiles. Il est statique et fortement prévisible. Ceci est du à la nécessité de connaître le temps pris par chaque fonction en train de s'exécuter et par la volonté de performance. L'apport du côté statique du système est d'avoir une réaction identique, quelle que soit l'exécution, sur les autres paramètres que la mémoire.

Impact sur notre algorithme génétique

Nous sauvegardons l'ensemble des états finaux obtenus par l'exécution des chromosomes de la population en cours d'exécution. Nous conservons également les chemins parcourus, c'est-à-dire les successions de primitives et paramètres exécutés. Nous sauvegardons aussi les graines utilisées par les générateurs de nombres pseudo-aléatoires. Ceci garantit la reproduction possible de l'ensemble de nos évaluations. Cela a un impact sur la composition intérieure de notre algorithme génétique, et plus particulièrement sur les étapes de génération d'une nouvelle population.

4. Comment concevoir une nouvelle population avec un algorithme prenant en compte le contexte ?

Il est nécessaire de modifier l'algorithme génétique dans son ensemble afin de conserver la propriété de chromosomes C valides et exécutables sur système réel, tout en rendant possible l'évolution du contexte du système. Nous modifions pour cela les étapes de l'algorithme. Celles qui ne sont pas modifiées conservent les propriétés définies dans l'algorithme de Gross [Gro03]. Nous présentons les modifications en deux parties. La première est l'impact des variables liées et des états atteignables ; et la deuxième est la solution apportée.

Sélection :

Impact : Chaque chromosome doit être exécuté sur système réel sans jamais le rendre indisponible et doit fournir le temps d'exécution de la fonction testée ;

Solution : Un *fitness*, représentant l'intérêt du chromosome à répondre au problème d'optimisation (WCET), est fixé en récupérant la valeur retournée par le système et dans le cas d'un chromosome non exécutable, celui-ci prend la valeur 0.

Croisement :

Impact : Le croisement ne peut être au bit près ou une simple modification de la liste des primitives ;

Solution : Nous croisons, pour chacun des fils créés, les $i_{ème}$ primitives de chacun des deux chromosomes sélectionnés par tournoi. Il existe deux possibilités :

— Lorsque les $i_{ème}$ primitives des deux parents sont identiques, nous transformons les paramètres P en binaires et donnons avec une probabilité $P_{croisement}$ le bit d'un des deux chromosomes parents. Dans certains cas particuliers comme les pointeurs, par exemple, une valeur sera générée et sera définie comme la $X_{ème}$ valeur de ce type. Elle correspond à un pointeur uniquement généré par ce type de fonction et conservé dans la liste des pointeurs générés par ce type de fonction.
— Lorsque les primitives des parents sont différentes, nous prenons la primitive de l'un des deux avec une probabilité $P_{croisement}$ en conservant les paramètres de celle-ci.

Nous réalisons cette opération jusqu'à obtenir une taille de population égale à la population initiale.

Mutation :

Impact : La mutation doit être capable d'influencer l'état du système et d'influer aussi bien sur les primitives que leurs paramètres ;

Solution : Nous mutons, pour chaque chromosome avec une probabilité $P_{mutation}$, les primitives et les paramètres. Le procédé se réalise en deux étapes :
— Pour chaque primitive, il y a une probabilité $P_{mutation}$ que celle si soit changée par une primitive choisie aléatoirement. La primitive choisie aléatoirement respecte la contrainte des variables liées.
— Chaque paramètre de chaque primitive est transformé en binaire et il existe une probabilité $P_{mutation}$ que chaque bit change de valeur. Le résultat obtenu est une valeur représentant le paramètre.

Les paramètres énoncés au-dessus ont été publiés dans notre deuxième contribution [GJP15].

5.Comment ne pas modifier ou influencer un système lors d'une mesure ?

Dans notre cas, cela n'est pas possible. Nous devons, comme indiqué précédemment, utiliser des horloges logicielles basées sur le démarrage du système par exemple. Il y a donc un impact sur le temps pris par cette fonction pour s'exécuter, mais cette valeur est mesurable et peut être supprimée. En revanche, son impact sur les bus et caches est bien réel. Dans le cas d'un système fortement optimisé, car cela pourrait mettre en défaut l'ordonnanceur. Mais les marges de sécurité minimales obligatoires sont et doivent restées supérieures au coût de notre observation.

Ainsi, nous avons bien une influence de l'observation avec un impact sur les caches et bus, ainsi que sur l'emplacement de nos données en mémoire. Pour neutraliser le problème de la place en mémoire et ne pas changer le système, il est possible d'ajouter une mémoire externe et de rendre ainsi nul son impact.

La mesure du WCET (et plus généralement l'algorithme génétique) doit donc être exécutée hors du système étudié.

6.Est-il possible de réaliser une parallélisation des mesures de temps ?

La parallélisation est un avantage certain des algorithmes génétiques. Elle est rendue possible par l'indépendance des chromosomes entre eux. Ainsi il peut y avoir autant de systèmes qu'il y a de chromosomes dans la population. Cela réduit d'autant le temps de calcul. Nous rejoignons également la vision d'un système central exécutant l'algorithme génétique et envoyant des ordres

aux systèmes testés. En effet, les systèmes testés ne reçoivent qu'un ensemble de primitives à exécuter et ne retournent que le temps mesuré de la dernière primitive envoyée (qui se trouve être la primitive en cours d'évaluation) ainsi que l'état de la mémoire. Ceci ne provoque pas plus d'influence et représente donc un avantage certain. Une optimisation permet même de partir d'un état non-initial grâce à la reprise du plan mémoire d'une ancienne succession d'opérations pour agrandir fortement la rapidité d'exécution sur système réel.

Impact sur notre algorithme génétique

L'algorithme génétique permet donc la parallélisation et est exécuté sur un système indépendant des machines en cours d'évaluation. Cette machine indépendante n'envoie aux systèmes testés qu'un ensemble de primitives à exécuter, partant de la population initiale.

7.Génération de la population initiale

La population initiale contient un ensemble de chromosome C. Ces chromosomes finissent tous par la primitive à tester avec ses paramètres propres. La population initiale est générée en 4 étapes :

Population initiale :

— Les primitives sont choisies aléatoirement dans les n fonctions systèmes pour chaque chromosome C ;

— Une nouvelle primitive aléatoire est sélectionnée en remplacement jusqu'à répondre aux exigences des fonctions liées, dans le cas où l'une d'elle est liée à une autre encore non existante ;

— Chacun des j paramètres P de chaque primitive p est choisi dans l'intervalle qui lui est propre ;

— Le résultat est une population de chromosome C générée pseudo-aléatoirement et répondant à l'ensemble des contraintes des variables liées. De plus, les C peuvent être exécutés sur le système réel.

8.Fort de ces questions, pouvons-nous mieux définir ce qu'est un système complexe ?

Dans cette vision des choses, le système complexe n'est représenté que par une succession d'actions. La complexité en détails du système n'est pas connu car il est boite noir. C'est à l'algorithme de voir les influences de chaque paramètre pour tenter d'influencer au mieux le système. Le système complexe n'est donc géré que par la succession des primitives et de leurs paramètres respectifs. Ceci répond donc pleinement, en théorie, à notre problématique.

9.Convergence finie ?

Pour obtenir une convergence de l'algorithme dans un temps fini, quoi qu'il advienne, il faut modifier l'algorithme et plus précisément le critère d'arrêt.

Impact sur notre algorithme génétique : La modification du critère permet la garantie d'une résolution en un temps fini. Le critère d'arrêt est composé de deux conditions :

— la population de chromosome n'évolue plus depuis un certain temps ;

— l'algorithme a effectué un nombre maximal d'itérations sans stabilisation de la population.

La solution est d'arrêter l'algorithme au bout d'un temps maximal afin de s'assurer qu'il termine un jour. Le résultat obtenu est bien moins intéressant qu'en cas de convergence, car cela stipule qu'il était encore possible d'augmenter le WCET. Cela permet en revanche de modifier l'état initial

de la population de départ pour une nouvelle génération permettant de "creuser" vers un meilleur WCET avec plus de précision ou simplement d'observer un bogue.

La convergence de la population vers une valeur est intéressante mais reste insuffisante si elle n'est exécutée qu'une seule fois. En effet, il est possible que la population se soit dirigée vers un "maximum local". Cela signifierait que ce n'est pas un WCET mais un temps d'exécution maximal d'un sous-ensemble du code étudié. Il nous faut donc exécuter plusieurs fois l'algorithme génétique, avec des populations initiales différentes afin de réaliser des statistiques viables.

5.5 Conclusion

Ce chapitre nous a permis de modéliser notre problème et de comprendre concrètement ce qui devait être mis en place pour y répondre au mieux. Nous avons présenté, par la suite, l'algorithme répondant au mieux à cette problématique. C'est-à-dire l'algorithme génétique. Notre premier apport concerne nos travaux initiaux sur le multi-critère montrant la faisabilité et le réalisme d'une telle proposition dans le cas d'une problématique de recherche du WCET. Cet apport de multi-critère à une plus-value et un impact certain, car il limite le risque de convergence vers des maximums locaux. Notre proposition 1 est complémentaire à notre seconde proposition qui conclue ce chapitre par un apport majeur, celui d'un nouvel algorithme génétique prenant en compte l'ensemble des contraintes fixées.

Nos hypothèses regroupent les paramètres suivants : "Boîte noire" limitant l'accès au code source ; nécessité d'exécution sur système réel dans l'optique d'une intervention en fin de chaine de production industrielle ; faible coût de développement et de formation ; faible spécialisation du personnel utilisant l'algorithme ; gain en temps d'exécution sans présence humaine.

Nous proposons les solutions suivantes : utilisation de primitives dont la documentation technique est clairement définie et accessible à tous, avec exécution dynamique permettant de garantir un fonctionnement continu et réel du système ; capacité de reprendre les boggues et de détecter les erreurs d'implémentation automatique ; utilisation d'un algorithme ayant déjà fait ses preuves et étant parfaitement connu du domaine scientique, ce qui implique un coût faible de développement et de formation et permet la parallélisation des calculs nécessaires à l'analyse de systèmes complexes demandant d'importantes ressources de calculs.

Nous présenterons dans le chapitre suivant une première évaluation de notre proposition afin de valider pleinement son fonctionnement.

Chapitre 6

Cas d'étude

Sommaire

6.1 Présentation générale

Dans ce chapitre, nous développons un cas concret d'étude de notre algorithme génétique avec l'observation d'un contexte d'exécution précis via notre algorithme génétique. Une fois présenté le contexte à observer, nous présentons la plate-forme étudiée. Nous observons ensuite la réaction et les performances de notre apport présenté au chapitre précédent, pour rappel, celui d'un algorithme génétique en mesure de gérer le contexte d'exécution d'un système en boite noire. Nous le testons précisément sur une fonction dont nous connaissons les impacts sur le système. Par ailleurs, au chapitre suivant, nous prenons un sous-ensemble des fonctions de bases du système étudié pour valider pleinement notre algorithme génétique et conclure notre contribution sur la mesure du pire temps d'exécution.

6.2 Cas de la fragmentation mémoire

Nous souhaitons mettre à l'épreuve notre idée, et pour cela, étudier un contexte d'exécution qui n'est pas repéré et géré par les autres algorithmes disponibles. Notre choix s'est porté sur un phénomène connu de fonctionnement de la mémoire. Celui-ci est appelé fragmentation mémoire [JW98].

La mémoire est un système bien connu qui peut, en fonction de son utilisation, avoir un impact sur les fonctions en cours d'exécution. La "vie normale" d'une mémoire commence par un état initial où le bloc mémoire est complet, vide et entièrement disponible. C'est le cas de l'exemple présenté à la figure 6.1 qui a une capacité mémoire de 1024 octets.

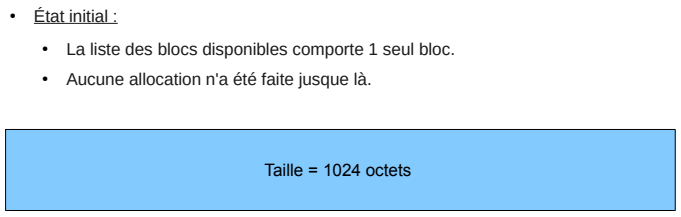

Fig. 6.1 – Vu d'un bloc mémoire vide

La mémoire peut ensuite recevoir des allocations. Les allocations mémoires sont des demandes d'utilisation d'une partie de celles-ci au profit de programmes s'exécutant sur un système. Elles allouent donc un espace réservé à la demande. Cette espace n'est plus disponible pour les autres programmes. C'est le cas de notre exemple figure 6.2. Nous observons deux allocations mémoires consécutives de 200 et 300 octets respectivement. Il ne reste donc plus, dans notre exemple, que 524 octets disponibles.

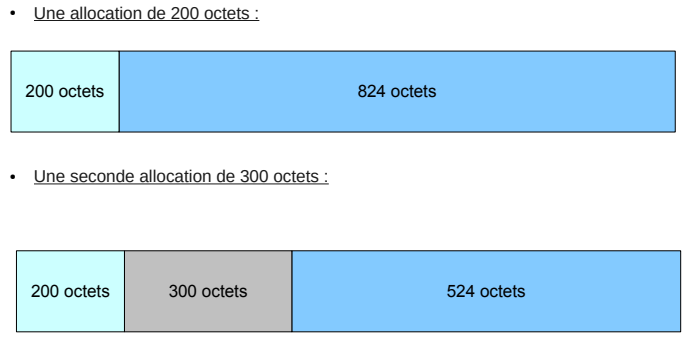

Fig. 6.2 – Vu de deux allocations mémoires successives

Une allocation ayant été faite, il est possible de la libérer, c'est-à-dire de rendre la partie mémoire

réservée pour que d'autres programmes puissent l'utiliser. Nous voyons cela sur la figure 6.3. Dans cet exemple, nous libérons en premier l'allocation de 200 octets effectuée en premier. Cela crée donc deux blocs vides non-contigus dans la mémoire, respectivement un de 200 octets et un de 524 octets. Puis nous libérons le bloc de 300 octets, ce qui permet à la mémoire de réunir l'ensemble des blocs vides contigus. Cette étape se nomme fusion. Dans notre exemple, cela correspond à la récupération de l'ensemble de la mémoire, soit 1024 octets libres.

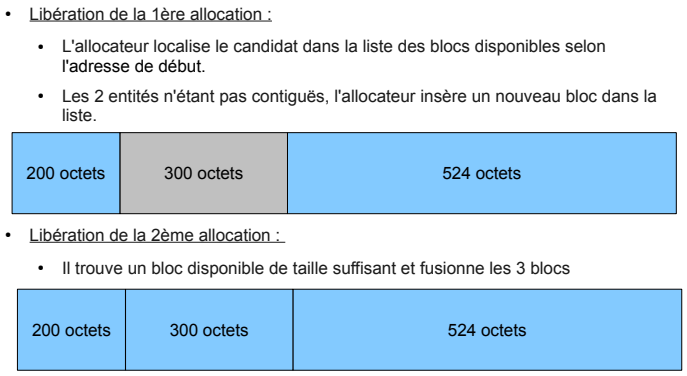

Fig. 6.3 – Vu d'une fragmentation mémoire suivie d'une fusion

Le principe du fonctionnement de la mémoire est relativement simple. Mais prenons un cas extrême de fonctionnement, tel que des allocations et des libérations successives visibles dans la figure 6.4. Dans cet exemple, nous observons que notre mémoire de 1024 octets a été, dans un premier temps, entièrement allouée par des petits blocs de 20 octets chacun, sauf pour le dernier avec 30 octets disponibles. Puis un bloc sur deux a été libéré afin de créer une succession de blocs pleins et vides et ce, jusqu'au dernier bloc (vide) de 30 octets. Ceci ne pose pas de souci en fonctionnement normal, néanmoins, dans le cas où un programme souhaiterait allouer une partie de la mémoire supérieure à 30 octets, il ne le pourrait pas sans remanier la mémoire. En revanche, s'il souhaite intégrer un bloc compris entre 21 et 30 octets, il est possible pour lui d'intégrer la dernière case vide.

L'impact temporel se trouve à cette étape. En effet, le système n'a pas d'oeil pour observer cette case vide et utilise un principe de pointeur. Il pointe sur la première case vide de 20 octets au début de la mémoire et lui demande sa taille. Ne satisfaisant pas aux conditions, le pointeur passe à la suivante et ainsi de suite jusqu'au dernier bloc. Le passage de case en case du pointeur à un coût certain et non négligeable dans un système. Nous le démontrerons par la suite.

Cette fragmentation mémoire n'est pas liée aux paramètres de la fonction en cours d'exécution mais à l'état de la mémoire à un instant t, dépendant entièrement de son contexte d'exécution. Il n'est pas possible de reproduire ce phénomène avec les algorithmes disponibles. Seule notre proposition peut générer ce phénomène grâce à la création d'un contexte d'exécution.

Le contexte pris en compte durant les expérimentations est donc l'état de la mémoire allouée par le RTOS. Son impact sur le temps d'exécution des primitives, et plus particulièrement l'influence

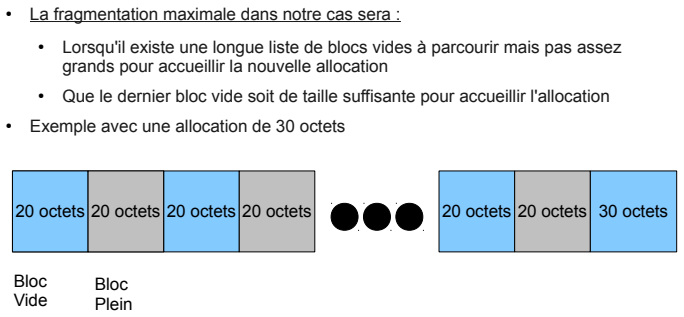

Fig. 6.4 – Vu d'une fragmentation mémoire maximale

temporelle de la fragmentation mémoire est observable.

Le RTOS conserve la liste des blocs libres qu'il doit parcourir à chaque fois qu'il désire effectuer une allocation, ralentissant d'autant l'exécution de la primitive `malloc` servant à l'allocation mémoire.

Mais est-il possible, dans un premier temps, d'observer cette contention sur un système réel, puis d'étudier notre algorithme sur ce phénomène afin de voir s'il détecte et prend en compte automatiquement celui-ci ? Nous verrons en premier lieu la plate-forme étudiée, puis notre algorithme génétique.

Le cas de la fragmentation mémoire est un cas typique de l'influence du contexte du système, et est régulièrement mis en cause ou sous-estimé dans les systèmes embarqués critiques. C'est en effet dans ce type de système que la fragmentation mémoire a le plus d'effets, l'ordonnancement pouvant échouer si la fragmentation a été mal mesurée. Un de mes projets étudiants sur les drones automobiles l'a démontré en 2011. Il est, de plus, difficile à mesurer, en particulier sur les satellites. Des cas réels démontrent qu'il existe encore aujourd'hui des problèmes de sûreté de fonctionnement liés à cette fragmentation mémoire. L'ensemble des problèmes reste souvent transparent aux clients exploitants le satellite mais nécessite des correctifs en "cours de vie".

6.3 Présentation de la plate-forme étudiée

Nous avons choisi d'étudier un système relativement complexe, mais surtout utilisable par tous et facile à trouver dans le commerce afin que les testes ne soient pas sur système propriétaire, et donc limités à leurs plate-formes. La contention mémoire étudiée dans un premier temps arrive sur la plupart des systèmes. Notre choix s'est orienté vers un système bas coût permettant la mise en œuvre de la parallélisation.

Il s'agit du système MBED [14] muni d'un OS FreeRtos [Bar10]. Voici les caractéristiques tech-

14. www.MBED.com

niques globales de cette plate-forme matérielle :

— MBED (http ://developer.mbed.org/platforms/mbed-LPC1768/) ;
— NXP LPC1768 MCU ;
— High performance ARM® Cortex^{TM}-M3 Core ;
— 96MHz, 32KB RAM, 512KB FLASH ;
— Ethernet, USB Host/Device, 2xSPI, 2xI2C, 3xUART, CAN, 6xPWM, 6xADC, GPIO.

Voici celles de l'OS FreeRtos :

— FreeRTOS (http ://www.freertos.org/) ;
— Noyau temps-réel configurable pour petits systèmes embarqués ;
— Gestion de tâches ;
— Ordonnanceur préemptif ;
— Gestion mémoire ;
— Gestions des queues, évènements etc...

Ce choix a été influencé par les travaux scientifiques dans le domaine. En effet, une évaluation de FreeRtos a déjà été effectuée sur une autre plate-forme (MSP430) dans [SR12].

Pour rappel, nous décidons d'étudier un sous-ensemble de 13 primitives de l'OS que voici :

Taches : create, delete, setPriority, suspend, resume ;

Ordonnanceur : getState ;

Semaphores : createBinary, createCounting, createMutex, createRecursiveMutex, delete ;

Mémoire : malloc, free.

FreeRtos a également l'avantage d'avoir été étudié sur un système temps réel dur [IMTSB11] mais aussi récemment avec des *Benchmarks* [XLvdS+08]. Les performances du système y sont développées dans cette thèse [XLvdS+08]. Nous avons donc un ensemble de valeurs nous permettant de nous comparer à la communauté scientifique par la suite.

La figure 6.5 représente le branchement effectué pour l'évaluation de notre plate-forme. Celle-ci se compose d'un ordinateur central ou fonctionne notre algorithme génétique ainsi que d'un ensemble de 20 MBED en parallèle. Nous pouvons donc étudier simultanément 20 chromosomes. La liaison se fait avec le protocole UDP. Ce protocole UDP a été éprouvé pour permettre une transmission rapide des informations et une perte de paquets très faible. Chaque MBED est représenté par un *Thread* au niveau de l'ordinateur réalisant l'algorithme génétique.

Nous développons également notre propre test sur la plate-forme mise en place pour référencer nos travaux avec une base. Celle-ci est décrite dans la section suivante présentant la primitive d'allocation mémoire en détail, celle du *malloc*.

6.4 Étude approfondie de la primitive malloc

La primitive *malloc* est une fonction de base du système permettant l'allocation dynamique de mémoire. Celle-ci peut, pendant l'exécution d'un programme, réserver une partie de la mémoire

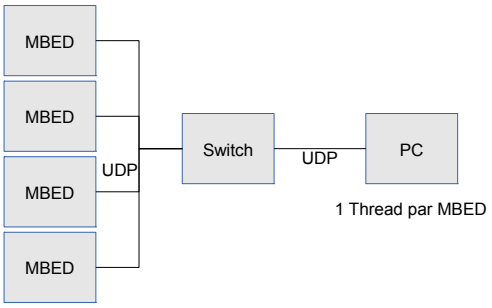

Fig. 6.5 – Vu d'ensemble de la plateforme d'expérimentation

globale. Son un temps d'exécution comprend le temps de réservation de la mémoire et le temps de recherche d'un emplacement suffisamment grand pour y installer sa zone à réserver.

Le système utilisé offre une observation du temps à la microseconde. Nous avons étudié, dans un premier temps, les caractéristiques de la fonction *malloc* sur notre plate-forme d'expérimentation sans algorithme spécifique. Le temps d'exécution de la primitive *malloc* est de 42ms lorsque aucun bloc mémoire n'est alloué. Dans le cas d'une fragmentation importante comme décrite ci-dessus et visible figure 6.4, le temps d'exécution est de 60ms (valeur moyenne prise pour 50 exécutions de la primitive). Il existe donc une influence non négligeable du contexte du système sur l'exécution de la primitive testée. Celle-ci équivaut à 14ms supplémentaires. Ce temps est totalement indépendant de la fonction elle-même, ou plus particulièrement de ces paramètres. L'impact est donc non-négligeable.

C'est l'impact de la fragmentation mémoire que nous allons essayer de retrouver et de caractériser avec notre algorithme génétique. Nous observons donc son impact sur notre algorithme incluant le contexte d'exécution. La première étape de notre évaluation avec l'algorithme est de calibrer celui-ci puis d'observer son fonctionnement et ses résultats.

6.4.1 Paramètres de l'algorithme génétique

Il existe trois paramètres à étudier, afin de calibrer notre algorithme génétique :

— la taille de la population ;
— le taux de mutation ;
— le taux de croisement.

En effet, la taille de la population donne les moyens de parcourir de grands espaces de recherche. Plus sa valeur est élevée, plus grand est l'espace de recherche visité. Le taux de croisement permet d'améliorer dans une petite zone de l'espace de recherche le WCET. Le taux de mutation peut faire sortir potentiellement un chromosome ou un ensemble de chromosomes d'un WCET local.

Les autres caractéristiques de notre algorithme sont basées sur les recherches approfondies de Gross [Gro03].

Nous prenons, dans un premier temps, une taille de génome (nombre de primitives exécutées avant la primitive testée) entre 25 et 200 primitives. Le WCET est observé en fin d'exécution de l'algorithme en fonction de la taille de la population pour l'ensemble des valeurs de génomes.

Le taux de mutation, le taux de croisement et la taille de la population ont été définis par expérimentation sur de nombreux essais. La relation entre ces 3 paramètres reste complexe. Les expérimentations ont démontré qu'il était nécessaire de prendre une taille de population importante afin de pouvoir explorer un espace de recherche plus grand et plus diversifié. Une taille de population plus faible que celle séléctionnée ne permet pas l'obtention d'un WCET estimé autre qu'un WCET mesuré local. Cette taille minimum de population implique un taux de croisement (0.2) assez élevé et un taux de mutation assez élevé (0.01). Dans la littérature, les valeurs que nous trouvons sont assez faibles (proche des valeurs obtenues par Gross). Néanmoins les expérimentations démontrent qu'un taux de croisement plus faible ne permet pas une convergence différente que celle d'un WCET mesuré local. Un taux de croisement plus élevé ne permet pas d'obtenir de meilleurs résultats. Le taux de croisement, par expérimentation, influence sur la diversité de la population et sa capacité à sortir d'un WCET mesuré local. Un taux faible offre un WCET mesuré local faible et un trop élevé ne permet pas une convergence de l'algorithme génétique rapide. Le taux de mutation correspond a une valeur obtenant les meilleurs résultats. Ces trois variables sont liées et s'influent entre elles.

Les courbes présentées figure 6.6 démontrent pour une population faible, l'intérêt d'un taux de croisement élevé, et pour une population de taille importante, l'effet négatif d'un taux trop élevé. Il est donc préférable de prendre une valeur du taux de recombinaison intermédiaire. Nous prendrons pour la suite un taux de 0.2. Pour observer le phénomène de l'influence du contexte sur les primitives, la taille de la population optimale n'est pas fixée afin d'observer son impact. L'ensemble des courbes ne nous a pas permis de définir une valeur optimale, car il existe un lien entre la taille du génome et celle de la population, lien que nous cherchons à déterminer.

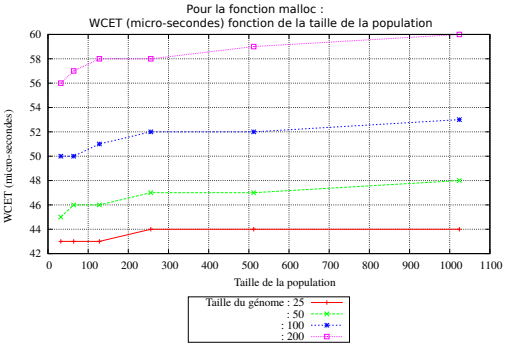

Fig. 6.6 – WCET(taille pop) pour tournoi de 4, recombinaison de 0.2, mutation de 0.01, primitive malloc

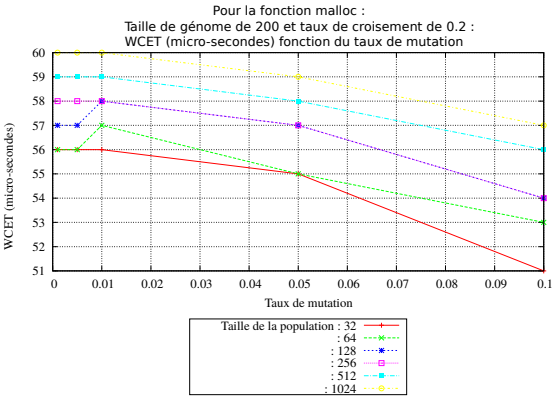

Fig. 6.7 – WCET (taux de mutation) pour tournoi de 4, recombinaison de 0.2, mutation de 0.01, primitive malloc

Le phénomène d'obtention d'un maximum local est parfaitement observable sur la courbe présentée ci-dessus, figure 6.6. Nous observons que, quelle que soit la taille de la population, le pire temps d'exécution (WCET) n'évolue plus à partir d'un certain seuil. Seule la taille du génome influe fortement sur la valeur du WCET. Cette influence est justifiable par le fait que 25 primitives

créent peu de fragmentation mémoire comparativement à 200. Il est normal que cette valeur se stabilise car il n'y a plus de fragmentation mémoire possible avec la taille de chaque génome donné. Une taille de population minimum de 500 est donc préférable.

Afin de confirmer notre première observation, nous cherchons à déterminer le taux de mutation nécessaire pour l'ensemble des populations précédemment testées. Ce taux influe sur la diversité génétique, et plus particulièrement sur la capacité de chacun des chromosomes à parcourir l'espace de recherche, c'est-à-dire de sortir d'un maximum local et d'offrir une grande diversité génétique. Cette valeur est faible dans l'ensemble des travaux précédents [Gro03], car un taux trop élevé de mutation à la particularité de stopper la convergence de l'algorithme génétique et d'avoir un effet négatif sur celle-ci.

L'observation du WCET figure 6.7, en fonction du taux de mutation démontre les propos de l'état de l'art, et confirme l'effet négatif d'un taux de mutation trop élevé quelle que soit la taille de la population. Un taux faible (mais non nul) de mutation est préférable. La valeur choisie pour la suite est de 0.01.

Le dernier paramètre à fixer est le taux de recombinaison. Pour décider de la taille de la population que nous utilisons, nous continuons à observer celle-ci, figure 6.8. Le taux de recombinaison a une influence sur la vitesse de convergence et donc permet à l'algorithme de converger trop vite. Cela a pour conséquence d'obtenir un maximum local ou trouver le WCET optimal (en fonction de la taille de la population). Un taux trop élevé doit, pour une population importante, impacter sur la diversité génétique de la même manière qu'un taux de mutation trop grand. Nous nous attendons à une valeur intermédiaire.

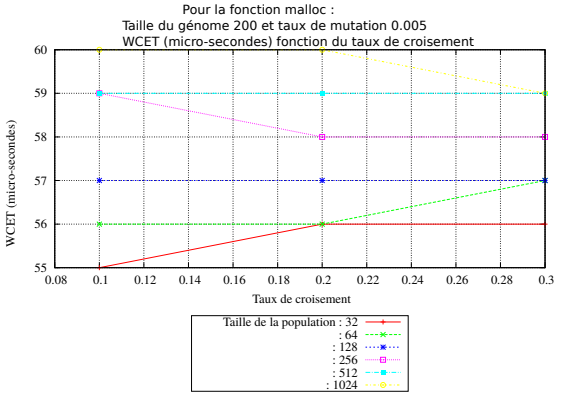

Fig. 6.8 – WCET (taux de croisement) pour tournoi de 4, recombinaison de 0.2, mutation de 0.01, pour la fonction malloc

6.4.2 Présentation des courbes analysant l'influence du contexte sur la fonction malloc

Les travaux précédents nous ont permis de calibrer et paramétrer le système réel ainsi que l'algorithme génétique. Nous pouvons à présent observer, dans le cas de la primitive *malloc*, l'influence du contexte système sur celle-ci. Nous étudions d'abord la taille du génome avec le temps

d'exécution obtenu. Plus la taille du génome est important, et plus il est possible de créer une contention mémoire sur le système réel. Nous nous attendons donc à une évolution croissante du WCET en fonction de la taille du génome, et ce quelle que soit la taille de la population.

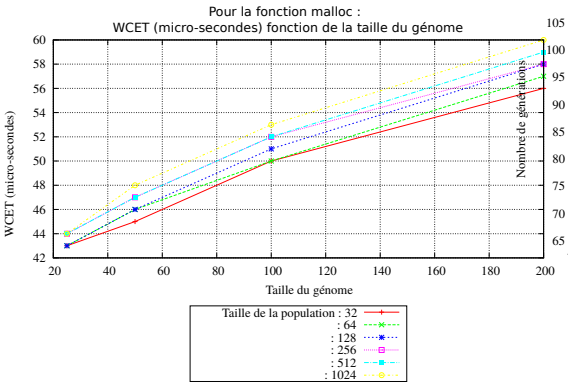

Fig. 6.9 – WCET (taille contexte) pour tournoi de 4 recombinaison de 0.2 mutation de 0.01 *malloc*

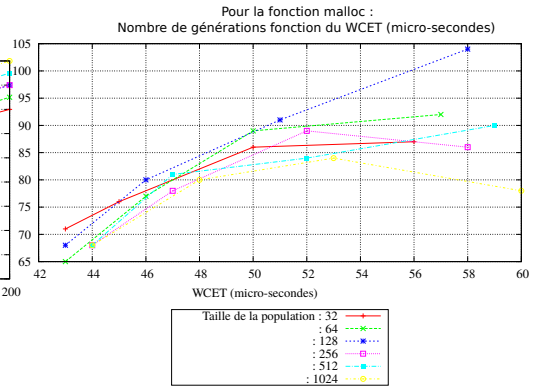

Fig. 6.10 – Nombre de génération de l'algorithme (WCET) pour un tournoi de 4, une recombinaison de 0.2, une mutation de 0.01 pour la fonction *malloc*

La courbe représentant le WCET en fonction de la taille du génome, figure 6.9, montre l'impact clair de la contention mémoire sur le WCET de la primitive *malloc*. Son temps d'exécution est bien fonction de la fragmentation mémoire.

Nous observons enfin le nombre de génération de l'algorithme génétique en fonction du WCET, figure 6.10, afin de déterminer si pour une taille de génome (et une taille de population) différente il existe un impact significatif sur le temps d'exécution de notre algorithme génétique.

Nous observons, figure 6.10, que la valeur du WCET que nous obtenons est maximale quelle que soit la taille de la population et de génome pour un nombre de génération de l'algorithme génétique assez important. La vitesse de convergence est très rapide pour une population de faible taille sans diversité génétique et sur une grande population avec une forte diversité génétique. Les cas intermédiaires demandent un plus grand nombre de génération de l'algorithme pour obtenir leur WCET. Nous pouvons conclure que le nombre de génération de l'algorithme génétique permet d'identifier rapidement si les paramètres de l'algorithme génétique et de la taille du génome sont correctes ou non. Il est préférable d'avoir une population importante pour avoir une vitesse de convergence rapide et un résultat de WCET important, soit une population de 1024 chromosomes et une taille de génome de 200.

Nous avons réussi à démontrer l'influence du contexte sur la primitive *malloc* mais également à déterminer l'évolution de l'algorithme génétique face à ce problème. Il détecte correctement l'impact sur le WCET de la primitive et converge vers celle-ci. Les paramètres à observer sont clairement définis.

6.4.3 Comparaison aux autres algorithmes

Nous avons également évalué la fonction *malloc* à l'aide d'un algorithme génétique de Gross [Gro03] ainsi que celui d'un système aléatoire.

Le système aléatoire a effectué 10 000 tentatives offrant un résultat équivalent à 43 microsecondes au total. L'algorithme génétique de Gross offre une valeur quasi identique. Cela s'explique par le fait que ces deux logiciels n'utilisent que les paramètres de la fonction testée. Les paramètres n'influencent que peu le temps d'exécution de celle-ci. ll est donc normal que ces logiciels ne puissent réussir à fragmenter la mémoire et donc trouver une valeur autre que la valeur initiale.

Nous observons ici l'intérêt de notre algorithme afin de déterminer le WCET mesuré. Notre algorithme génétique ne sera pas comparé aux autres algorithmes dans la suite du document car les autres algorithmes n'observent pas le contexte. Nous nous intéresserons à l'explication du contexte avec notre seul algorithme. En revanche, nous utiliserons les autres algorithmes pour montrer l'influence des paramètres de chaque primitive évaluée.

6.5 Conclusion

Ce chapitre, prémisse d'un plus imposant, montre l'influence du contexte d'exécution ainsi que la capacité de notre algorithme à l'observer et à l'utiliser pour optimiser le WCET mesuré. Le chapitre a permis de conclure sur des informations utiles de notre algorithme mais aussi sur un seul phénomène et une seule primitive. La population initiale est générée à l'aide d'un algorithme pseudo-aléatoire pour lequel nous collectons la graine de génération nous permettant de rejouer les tests autant de fois que nécessaire. Les expérimentations ont démontré que les générations aléatoires ont toutes permis la construction d'un effet de contention mémoire. Seul le temps d'exécution et le nombre de génération de l'algorithme génétique diffère. En effet, toutes les graines aléatoires ont généré un taux de fragmentation maximal mesurable, grâce à des valeurs de taux de croisement, de mutation et de taille de population bien choisis. Est-il capable, avec les mêmes paramètres d'initialisations, d'obtenir des résultats semblables, quelles que soit les autres primitives ? Est-il capable de découvrir l'ensemble des influences du contexte d'exécution ? Nous tenterons de répondre à ces questions au chapitre suivant, qui reprend les primitives du système et les évalue à l'aide de notre algorithme génétique.

Chapitre 7

Expérimentation

Sommaire

7.1 Introduction

Fort du constat de l'impact du contexte sur le temps d'exécution d'une primitive, nous étudions dans ce chapitre les autres primitives ainsi que l'impact du contexte sur leurs temps d'exécution. Pour cela, nous mettrons en œuvre notre algorithme génétique qui prend en compte le dit contexte. Après avoir rappelé l'analyse et la méthodologie mise en oeuvre pour la mesure avec d'autres algorithmes, nous comparons, dans la section suivante, les résultats obtenus avec l'algorithme génétique de Gross et un algorithme aléatoire, afin d'observer l'influence des paramètres sur le temps d'exécution de la fonction et ainsi dissocier les deux. Les primitives choisies dans FreeRTOS correspondent à des codes exécutés pour lequel la recherche scientifique démontre que celles-ci ont une influence sur le contexte d'exécution mesurable et pour lequel nous avons des mesures et valeurs auxquels nous comparer. Ce chapitre permettra d'introduire les perspectives et conclusions finales de cet apport majeur qu'est la mesure du temps d'exécution dynamique dans un système complexe.

7.2 Les primitives et leurs impacts

Dans cette section, nous étudions les primitives et l'impact du contexte du système et des paramètres des primitives sur le temps d'exécution. Nous détaillons les primitives de la façon suivante :

— rappel sur le rôle de la fonction ;
— rappel des valeurs utilisées pour l'algorithme génétique ;
— observation du WCET en fonction de la taille de la population ;
— observation du WCET en fonction de la taille du contexte ;
— observation du nombre de génération en fonction de la taille de la population ;
— analyse et conclusion sur les résultats obtenus.

Nous suivrons ces étapes pour l'ensemble des primitives suivantes :

— *create task* ;
— *get scheduler state* ;
— *create binary semaphore* ;
— *create counting semaphore* ;
— *create mutex* ;
— *create recursive mutex*.

La plate-forme utilisée est la même que celle présentée au chapitre précédent. Ainsi nous partons de l'état initial de FreeRTOS sur le système MBED pour engendrer des contextes d'exécutions suivis de la primitive à tester.

7.2.1 create task

La primitive *create task* permet la création dynamique de tâches. C'est-à-dire qu'elle alloue en mémoire deux emplacements pour y mettre les états globaux et variables qui suivrons. Celle-ci est automatiquement ajoutée à la liste des tâches prêtes à être exécutées.

Cette primitive prend six paramètres et en retourne un. Le premier est un pointeur sur le code à exécuter dans la future tâche. Le second est un nom facilitant le débogue. Le troisième correspond à la taille mémoire que nous souhaitons allouer pour la tâche. Le quatrième est un pointeur qui servira par la suite de paramètre à la future tâche. Le cinquième est le niveau de privilège donné à la tâche (temps-réel). Et le dernier est un référencement de la tâche générée. Le paramètre de retour est une confirmation ou non de la génération de la dite tâche.

Avant de mettre en œuvre notre algorithme génétique, nous voyons que cette primitive sollicitera principalement la mémoire. Nous nous attendons à un phénomène proche de celui observé par la primitive *malloc*.

Nous étudions la primitive *create task* avec les paramètres de l'algorithme génétique suivant :

— tournoi de 4 ;
— taux de croisement 0.2 ;
— taux de mutation 0.01 ;
— 10 réalisations complètent de l'algorithme génétique jusqu'au critère d'arrêt.

Nous étudions dans un premier temps la figure 7.1, avec l'algorithme génétique, le pire temps d'exécution mesuré en fonction de la population. Cette courbe permet de voir le pire temps d'exécution mesuré en fonction des différentes tailles de génomes et de voir son influence en fonction de l'état du système.

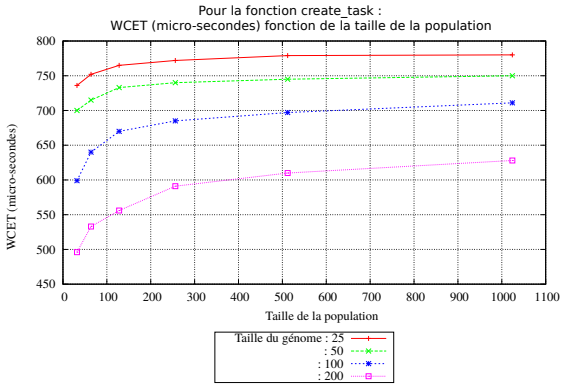

Fig. 7.1 – WCET (taille de la population) pour tournoi de 4, recombinaison de 0.2, mutation de 0.01, pour la fonction create task

Les valeurs observées sont très importantes. En effet, l'algorithme détermine des chemins d'exécutions permettant d'atteindre une durée de 770 micro-secondes. Plus il y a de fonctions intermédiaires, c'est-à-dire de fonctions concevant le contexte, et plus le temps d'exécution trouvé au final est conséquent. La courbe présentée a des similitudes avec les courbes observées par l'analyse de la primitive *malloc*. La différence est le temps d'exécution pris par cette primitive système. Ayant déjà déterminé l'impact de la mémoire sur le système, nous savons que celui-ci n'est pas aussi important que l'écart de temps observé sur cette courbe. Le temps d'exécution à l'état initial du système pour cette primitive est mesuré à 450 micro-secondes soit un écart de 320 micro-secondes avec le WCET mesuré. L'écart de temps réalisé pour l'allocation mémoire, avec cette même valeur, est d'environ 60 micro-secondes pour chacune des deux allocations dans le pire des cas, soit 120 micro-secondes maximum. Il reste donc 200 micro-secondes non encore expliquées. La taille de la population, elle, fait stagner rapidement le pire temps-d'exécution à une valeur fixe. Pour cette primitive nous observons la valeur retenue de 1024, pour la taille de la population, reste cohérente, car elle obtient le meilleur résultat.

Afin de mieux comprendre pourquoi il existe un écart de temps d'environ 200 micro-secondes non encore déterminé, nous observons la figure 7.2. Celle-ci représente le pire temps d'exécution mesuré en fonction de la taille du génome, c'est-à-dire en fonction du nombre de primitives exécutées précédemment à la primitive testée. Cette dernière augmente, dans le cas de la fragmentation mémoire.

Il se produit pourtant l'inverse sur cette figure 7.2. Plus le contexte d'exécution est important et moins la primitive ne prend de temps à s'exécuter. Cela veut dire que, contrairement aux conclusions amenées par la première courbe, l'augmentation du temps d'exécution ne dépend pas de la fragmentation mémoire. Celle-ci n'a en effet qu'un impact fortement limité avec quelques primitives et augmente avec un nombre d'accès mémoire plus important.

L'hypothèse se porte pourtant bien sur la mémoire. En effet, la primitive de création de tâche ne fait que s'écrire dans une liste et accéder à la mémoire. C'est sur cet accès mémoire que nous

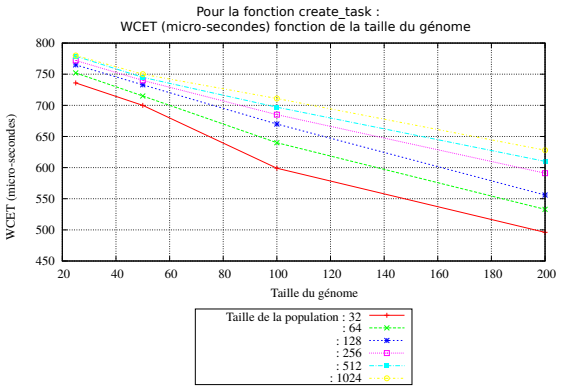

Fig. 7.2 – WCET (taille du génome) pour tournoi de 4, recombinaison de 0.2, mutation de 0.01, pour la fonction create task

étudions le système. Nous voyons dans le code qu'il existe une différence entre le *malloc* et l'appel a la mémoire des autres primitives. Le site de FreeRTOS [15] l'explique par la volonté de rendre déterministe le temps d'exécution des fonctions. Celui-ci est fait pour initialiser l'emplacement de la mémoire au début de l'exécution. Il s'avère que notre algorithme génétique tente de faire des créations de tâches avec de petites et grandes valeurs d'espace mémoire au début du programme. Les grands espaces mémoire demandés sont initialisés à zéro. Ainsi ces blocs prennent du temps à s'exécuter. Ce n'est pas le cas des tâches mémoires que nous réalisons par la suite (lors de grands génomes). Celles-ci ont uniquement de petits espaces mémoires à allouer ou sont simplement rejetées faute de place libre en mémoire, car celle restante est plus faible. Ainsi le temps d'exécution est logiquement plus faible après une succession de nombreuses primitives. Cette hypothèse est validée par les nombreuses explications disponibles sur le site de FreeRTOS.

Notre algorithme arrive donc à détecter ce phénomène. Il s'avère plus gênant et problématique que la fragmentation mémoire, elle-même détectée sur ce cas, mais faible en comparaison.

L'analyse de la troisième courbe, figure 7.3, permet d'observer le nombre de générations nécessaire à l'algorithme génétique pour trouver son résultat final.

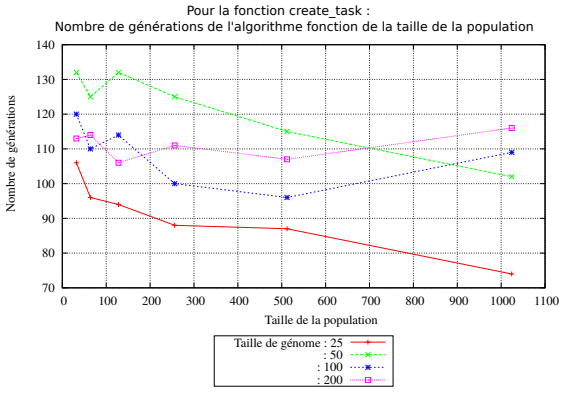

Fig. 7.3 – Nombre de générations (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create task

Cette courbe, représentant le nombre de génération de l'algorithme génétique en fonction de la

15. www.freertos.org

taille de la population, montre la difficulté pour lui de trouver une valeur de WCET avec une petite population. S'il y a trop peu de chromosomes, le WCET mesuré augmente à chaque itération et ainsi l'algorithme prend plus de temps à s'exécuter qu'avec une population plus importante. L'espace de recherche est plus faible pour les chromosomes avec un génome restreint. Nous observons la diminution du nombre de génération de l'algorithme plus la population est grande. En effet, celui-ci trouve plus rapidement le WCET mesuré. Ce n'est pas le cas des chromosomes avec un grand nombre de génomes. Le nombre de génération du programme ré-augmente avec une population élevée. En effet, le nombre de génération est plus important car l'algorithme ne tombe pas dans un WCET local, en raison de la diversité de la population.

Cette primitive nous a permis d'explorer une particularité de notre système et de voir qu'au-delà de ce que nous pourrions déterminer, l'implémentation peut offrir des particularités temporelles. L'initialisation de grands espaces mémoires a donc plus d'influence sur le temps que la fragmentation mémoire elle-même.

7.2.2 get scheduler state

Nous testons une autre primitive qui, cette fois, n'appelle pas la mémoire en écriture mais offre un résultat dépendant de la liste des tâches disponibles et de l'état de l'ordonnanceur. Cette fonction retourne à l'état dans lequel l'ordonnanceur se trouve, c'est-à-dire détermine s'il est démarré, suspendu, en fonctionnement ou arrêté. Nous nous attendons à un temps constant, ou peu variable. La fonction testée dépend peu du contexte d'exécution.

Nous observons la figure 7.4 qui nous présente le temps d'exécution maximal observé en fonction de la taille de la population. Nous constatons un WCET mesuré complètement fixe, quelle que soit la taille de la population ou la taille du contexte. En effet, la demande à un registre en lecture simple, est réalisée dans un temps constant et égal à 27 micro-secondes. Cette valeur est équivalente à celle obtenue à l'état initial du système.

Sur le graphique, nous voyons ainsi que la taille de la population ne joue pas de rôle dans le résultat trouvé. L'algorithme n'a pu maximiser la valeur car cette fonction, d'après son code fortement prédictible, ne dépend pas de paramètres.

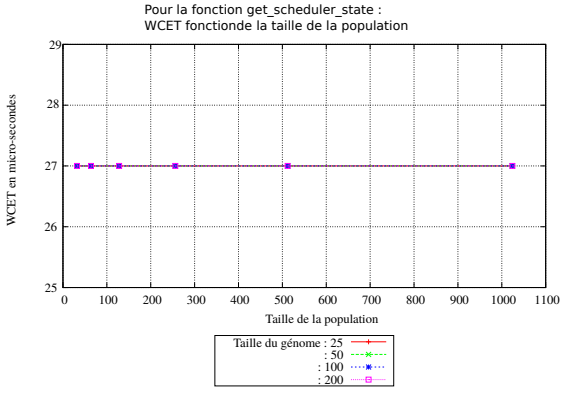

Fig. 7.4 – WCET (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction get scheduler state

L'absence d'influence du contexte du système sur l'exécution de la fonction *get scheduler state* est également observable dans la figure 7.5. Cette figure représente le WCET en fonction de la

taille du génome. Chaque courbe représente la taille de la population. A l'aide de ce graphique, nous voyons que le contexte d'exécution, quelle que soit sa valeur, n'a aucun impact sur le WCET. C'est une fonction temps réel dure, comme l'a souhaité le développeur de FreeRTOS.

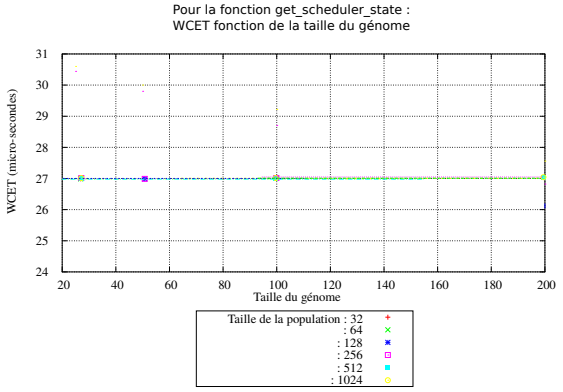

Fig. 7.5 – WCET (taille du génome) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction get scheduler state

La dernière figure représente le nombre de génération de l'algorithme en fonction de la population et confirme, encore une fois, que cette fonction fortement basique a une évolution constante en fonction de la taille de la population et des primitives exécutables au préalable. L'algorithme génétique, ne pouvant faire évoluer la valeur maximale du WCET, s'arrête systématiquement au bout du critère d'arrêt fixé a 50 générations de l'algorithme, sans évolution de la valeur du temps d'exécution. Outre le fait que cela prouve que notre algorithme est correctement implémenté, cela démontre également que certaines fonctions sont parfaitement indépendantes du contexte. Elles peuvent également être indépendantes des paramètres.

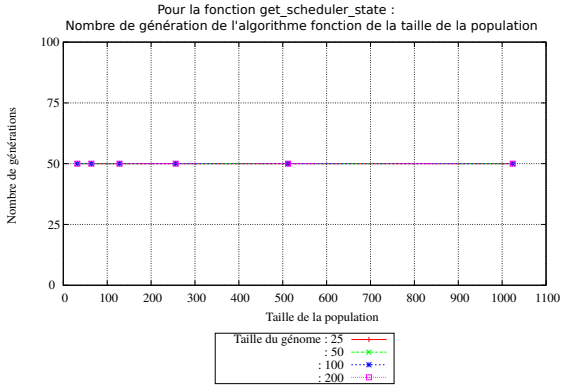

Fig. 7.6 – Nombre de générations (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction get scheduler state

Pour conclure sur cette fonction de base, notre algorithme n'a pas été en mesure de trouver un cas de déviance dans le code. Celui-ci est confirmé par le concepteur de freeRTOS qui garantit le temps d'exécution de cette fonction. Il n'y a donc pas de génération de faux positifs ou de faux chemins.

7.2.3 create binary semaphore

La fonction *create binary semaphore* est utilisée pour gérer les exclusions et la synchronisation des tâches.

Les sémaphores *mutex* et *binary* sont très proches en matière de code et de fonctionnement. Il existe néanmoins quelques différences subtiles : *Mutex* comprend un mécanisme d'héritage de priorité qui n'existe pas pour les sémaphores binaires. Les sémaphores binaires sont donc préférés pour les applications de synchronisation (entre les tâches ou entre plusieurs tâches et une interruption). *Mutex*, quant à lui, est préféré pour la mise en œuvre d'une exclusion mutuelle simple. La description du fonctionnement d'un *mutex* est la même que pour les sémaphores binaires. Cette sous-section ne décrit que les sémaphores binaires utilisés la synchronisation.

La fonction *create binary semaphore* rend possible la création d'un délai d'attente maximal modifiable. Le temps d'attente ainsi généré indique le nombre maximum de *ticks*(impulsions) d'horloge qu'une tâche doit attendre dans l'état bloqué en tentant de «prendre» un sémaphore car celui-ci ne doit pas être immédiatement disponible. Si plusieurs tâches tentent d'accéder à un même sémaphore, alors la tâche avec la priorité la plus élevée est celle qui s'exécutera à la prochaine disponibilité du sémaphore.

Le fonctionnement du sémaphore binaire équivaut à la gestion d'une file d'attente n'ayant qu'une seule ressource disponible à se partager. La file d'attente ne peut être que vide ou pleine (donc binaire). Les tâches et les interruptions qui utilisent la file d'attente ne se soucient pas de ce qu'elle contient - ils veulent uniquement savoir si la file d'attente est vide ou pleine. Ce mécanisme peut être exploité pour synchroniser (par exemple) une tâche avec une interruption.

Considérons le cas où une tâche est utilisée pour desservir un périphérique. L'interrogation systématique et régulière du périphérique serait du gaspillage de ressource, et empêcherait d'autres tâches de s'exécuter. Il est donc préférable que la tâche passe le plus clair de son temps à l'état bloqué (permettant à d'autres tâches de s'exécuter). Pour réaliser ce cas d'utilisation, nous réalisons un sémaphore binaire en mettant la tâche cherchant à s'exécuter dans la liste d'attente de la sémaphore. Une routine d'interruption est alors écrite pour que le périphérique 'donne' le sémaphore à la tâche lorsque celui-ci a quelque chose à lui fournir. La tâche «prend» toujours le sémaphore, mais elle ne redonne jamais celui-ci à quelqu'un d'autre en particulier. C'est aussi le cas des interruptions.

La priorisation des tâches peut être utilisée pour assurer à certains périphériques des services dans un temps donné et générer efficacement un régime “d'interruption différé”. Une autre approche consiste à utiliser une file d'attente à la place du sémaphore. Lorsque cela est fait, la routine d'interruption peut capturer les données associées à l'événement du périphérique et l'envoyer sur une file d'attente de la tâche. La tâche se débloque lorsque les données sont disponibles sur la file d'attente. Elle récupère ainsi les données à partir de la file d'attente, puis elle effectue tout le traitement de données nécessaire. Cette autre approche permet aux interruptions d'être aussi courtes que possible.

La création de sémaphore binaire génère donc une possibilité de dépendance entre les tâches. Nous étudions ici la fonction de création de ces dépendances et non de l'utilisation de celle-ci. La création d'une telle fonction génère un sémaphore vide. Il n'y a pas de paramètre à cette fonction. Le paramétrage se fait à posteriori. Le choix de cette fonction s'explique aussi bien par l'utilisation

d'une fonction de dépendance entre les tâches que parce-que cette fonction est seulement influencée par son contexte d'exécution.

L'absence de paramètre ne permet pas l'utilisation de l'algorithme génétique de Gross. En revanche, c'est un "défi" pour notre algorithme. La fonction est plus complexe que *Get scheduler state*. En effet, cette fonction crée une liste de queue, inscrit un registre et initialise les interruptions. Le temps de cette fonction reste néanmoins constant et prévisible.

Nous observons sur la figure 7.7 le WCET mesuré en fonction de la taille de la population. Chaque courbe est représentante d'une taille de génome. Ces courbes, à l'allure semblable à celles du *malloc* réalisent un effet de seuil. Autrement dit, la taille de la population atteint son maximum rapidement et ne peut plus faire évoluer le WCET. En effet, les chromosomes composant la population étudiée ont atteint leur maximum local. C'est la taille du génome, et donc du contexte d'exécution du système qui pourra à nouveau faire évoluer la valeur du WCET mesuré.

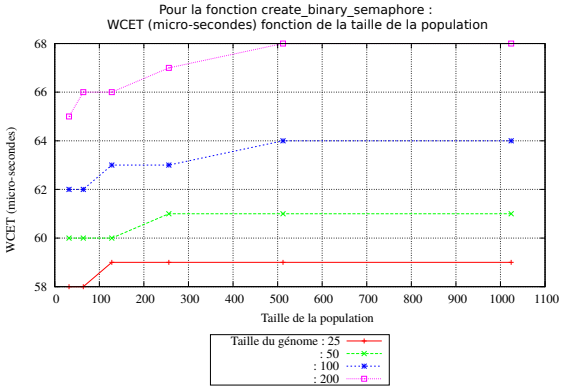

Fig. 7.7 – WCET (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create binary semaphore

La figure 7.8 représentant le WCET mesuré en fonction de la taille du génome confirme notre vision. Le contexte du système influe directement sur la valeur du WCET : plus celui-ci est grand, plus la valeur obtenue est importante, jusqu'à un certain seuil, toutefois obtenu avec un contexte de 200. Comme dit précédemment, la fonction réalise des accès mémoire pour créer une liste et inscrire un registre. La fonction testée devient donc sensible au phénomène de fragmentation mémoire. Comme nous le constatons précédemment, la fonction *malloc* qui mettait en avant la fragmentation mémoire était inférieure à 60 micro-secondes. Or ici nous obtenons 68 micro-secondes. Ceci n'est pas une erreur mais bien le temps supplémentaire pris pour l'initialisation des interruptions. Cette activation des interruptions, testée séparément de la fonction *create binary semaphore*, est indépendante du contexte d'exécution du système. En effet, son temps est constant et l'algorithme génétique n'a pu faire évoluer cette fonction constante.

En figure 7.9 nous présentons le nombre de génération de l'algorithme génétique en fonction de la taille de la population et avec des courbes représentant les différentes valeurs de la taille du génome. Ce graphique montre qu'il est nécessaire pour l'algorithme génétique de réaliser plus de générations (sélection, évaluation, croisement et mutation) lorsque le contexte d'exécution à explorer est vaste que lorsque celui-ci est restreint. En effet, l'espace de recherche étant plus grand, il lui faut plus de temps pour converger vers un résultat définitif. Ceci induit un temps d'exécution global plus important pour un génome de valeur élevée, et ce, sans prendre en compte que l'exécution sur plateforme réel prendra également plus de temps, car nous exécutons plus de primitives dessus. La taille

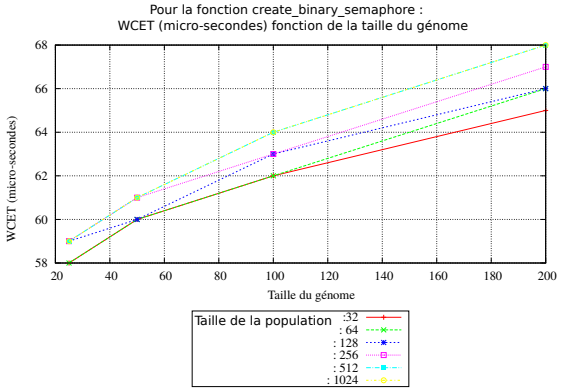

Fig. 7.8 – WCET (taille du génome) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create binary semaphore

de la population influe peu le nombre de génération. La fluctuation des points pour chacune des courbes s'explique par le fait que pour chaque point d'une courbe obtenu, l'algorithme génétique a été entièrement ré-exécuté avec une population différente. L'algorithme converge vers un résultat plus ou moins rapidement en fonction de ces différentes populations initiales.

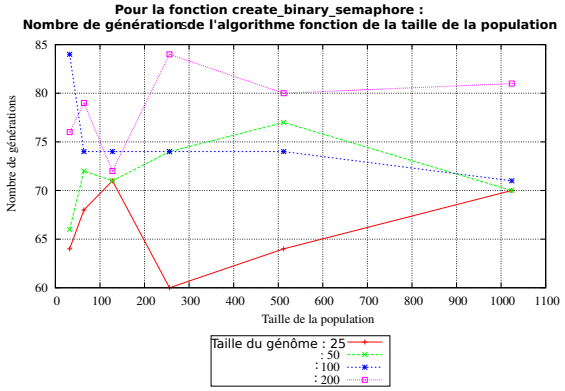

Fig. 7.9 – Nombre de générations (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create binary semaphore

Pour conclure sur la fonction *create binary semaphore*, nous savons qu'elle est dépendante du contexte d'exécution. Nous pouvons même déterminer que le contexte ayant le plus gros impact sur son temps d'exécution est celui de la fragmentation mémoire. Notre algorithme génétique a réussi à trouver automatiquement cette influence et a pu l'exploiter au mieux pour trouver un maximum local, jusqu'à la combinaison idéale d'une forte population et d'une taille de génome élevé, lui permettant d'obtenir un WCET mesuré.

7.2.4 create counting semaphore

De même que les sémaphores binaires peuvent être considérés comme des files d'attentes de longueur un, les *counting semaphore* peuvent être considérés comme des files d'attente d'une longueur supérieure à un. Encore une fois, les utilisateurs du sémaphore ne s'occupent pas des données qui sont stockées dans la file d'attente mais uniquement de savoir si la file d'attente est vide ou pleine.

Ces sémaphores sont généralement utilisés pour deux choses :

— le comptage des événements ;
— la gestion des ressources.

Dans le cas d'utilisation de comptage des évènements, un gestionnaire d'événement sera «donné» à un sémaphore à chaque fois qu'un événement se produit (incrémentation de la valeur du compteur du sémaphore), et une tâche du gestionnaire va «prendre» un sémaphore à chaque fois qu'il traite un événement (décrémente la valeur de comptage sémaphore). La valeur obtenue est par conséquent la différence entre le nombre d'événements qui s'est produit et le nombre qui a été traité. Dans ce cas, il est souhaitable que la valeur soit nulle lorsque le sémaphore est créé.

Dans le cas de la gestion de ressource, la valeur obtenue indique le nombre de ressources disponibles. Pour obtenir le contrôle d'une ressource, une tâche doit d'abord obtenir un sémaphore (elle décrémente la valeur de comptage du sémaphore). Lorsque la valeur atteint zéro, il n'y a pas de ressources libres. Lorsqu'une tâche se termine avec la ressource il rend le sémaphore en retour et incrémente la valeur de comptage du sémaphore. Dans ce cas, il est souhaitable que la valeur soit égale à la valeur maximale lorsque le sémaphore est créé.

Le graphique 7.10, représentant le WCET en fonction de la taille de la population, est semblable à celui de la fonction précédemment étudiée *create binary semaphore*. Cela s'explique par le code utilisé par ces deux fonctions. Il est quasiment identique. L'algorithme génétique démontre que le chemin maximisant la valeur du WCET est le même pour ces deux fonctions. Ainsi, nous obtenons toujours une augmentation du WCET lorsque la taille de la population augmente mais aussi lorsque la taille du contexte augmente. Nous observons toujours une influence non-négligeable du contexte d'exécution du système ainsi que des valeurs maximales obtenues par la taille de la population. Une valeur moyenne de la taille de la population (512) est suffisant pour trouver le WCET. En revanche, il est nécessaire d'augmenter au maximum le contexte d'exécution.

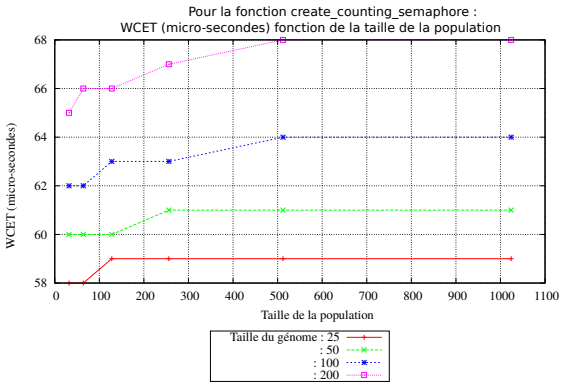

Fig. 7.10 – WCET (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create counting semaphore

La figure 7.11 complète les résultats obtenus. Cette courbe, représentant le WCET en fonction de la taille du génome, offre un nouvel angle de vue sur l'importance du contexte dans cette fonction. Nous observons un résultat à nouveau très proche de la fonction précédente. Le WCET dépendant d'une taille de population suffisante nécessite un contexte important. A nouveau la nécessité d'obtenir un contexte de forte valeur s'explique par la fragmentation de la mémoire déjà

étudiée dans la fonction *malloc* et observée dans la fonction “soeur” (en matière de comportement temporel) qu'est la fonction *create binary emaphore*.

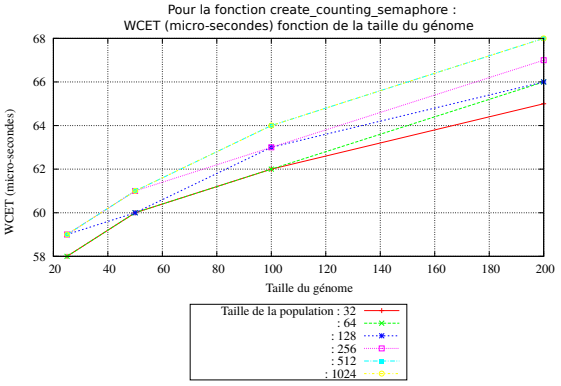

Fig. 7.11 – WCET (taille du génome) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create counting semaphore

La figure 7.12, représentant le nombre de génération en fonction de la taille de la population, présente de fortes similitudes avec celle représentant la fonction précédente. La question d'une erreur d'expérimentation de notre part s'est posée, cependant nos vérifications infirment cette hypothèse. Il s'avère que certaines fonctions, dont le code est suffisamment proche, sont maximisées de la même manière par notre algorithme. Ainsi il existe de nombreuses différences lors de l'exécution de l'algorithme. Cependant, ces courbes représentent uniquement les valeurs maximales obtenues. En effet, les différences d'exécutions de l'algorithme ne s'observent pas à l'oeil nu. Nous pouvons en conclure qu'il existe un chemin commun aux deux fonctions. Cela aurait un avantage certain si nous souhaitions comprendre le fonctionnement général d'une fonction inconnue mais proche d'une autre. Nous pouvons également conclure que notre algorithme est capable de suivre la même optimisation pour trouver un majorant au temps d'exécution.

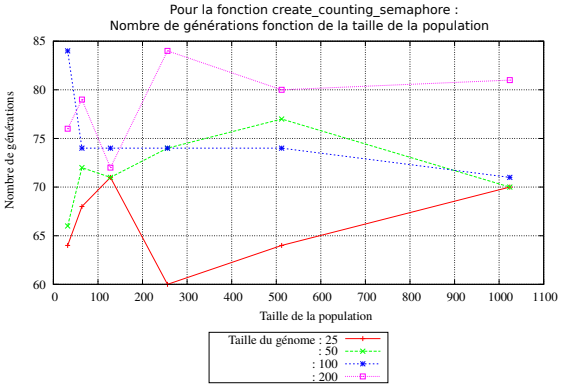

Fig. 7.12 – Nombre de générations (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create counting semaphore

7.2.5 create mutex

Les *mutex* sont des sémaphores binaires qui comprennent un mécanisme d'héritage de priorité. Étant donné que les sémaphores binaires sont le meilleur choix pour la mise en œuvre de synchro-

nisation (entre les tâches ou entre les tâches et une interruption), les *mutex* sont le meilleur choix pour la mise en œuvre d'exclusion mutuelle simple.

Lorsqu'il est utilisé pour l'exclusion mutuelle, le *mutex* agit comme un jeton qui est utilisé pour protéger une ressource. Quand une tâche souhaite accéder à la ressource, il doit d'abord obtenir (ou «prendre») le jeton. Quand il a fini avec la ressource, il doit «donner» le jeton en retour, ce qui permet à d'autres tâches d'accéder à la même ressource.

Les *mutex* utilisent les mêmes fonctions d'accès au sémaphore que les sémaphores binaires. L'héritage de priorité ne gère pas l'inversion de priorité. Il minimise uniquement son effet dans certaines situations. Par exemple, dans les applications temps-réel dure qui doivent être conçues de telle sorte que l'inversion de priorité ne se fait pas en premier lieu.

Les *mutex* ne doivent pas être utilisés avec une interruption car :

— ils comprennent un mécanisme d'héritage de priorité qui n'a de sens que si le *mutex* est pris et donné à partir d'une tâche, pas une interruption ;
— une interruption ne peut pas bloquer et attendre une ressource qui est gardée par un *mutex*.

Nous nous attendons à observer un WCET influencé par le contexte d'exécution encore une fois. En effet, vu l'utilisation des mêmes caractéristiques que les sémaphores binaires, nous prévoyons de voir l'influence de la mémoire et plus précisément de la fragmentation de celle-ci.

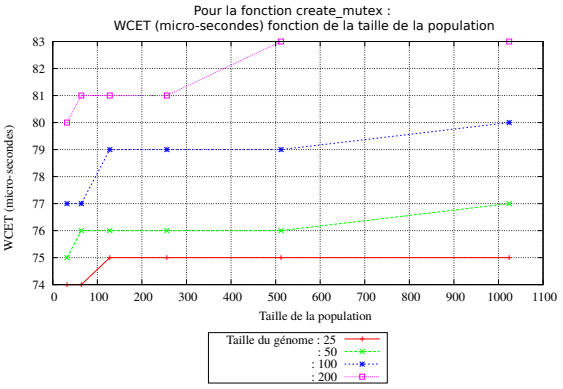

Fig. 7.13 – WCET (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create mutex

Dans la figure 7.13, représentant le WCET en fonction de la taille de la population, nous pouvons à nouveau observer que les différentes courbes représentant des valeurs de génomes trouvent un meilleur WCET lorsqu'elles sont élevées. Par ailleurs, la taille de la population est intéressante dans ce cas précis. En effet, il existe un saut de valeur entre une population de taille intermédiaire (512) et une valeur élevé (1024), ce qui entraîne un saut de la valeur du WCET mesuré. Ce saut montre que, pour ce cas particulier, la taille de la population a un impact majeur sur le résultat obtenu. En effet, lorsqu'il existait auparavant un palier (généré par une taille de la population trop faible pour explorer l'espace de recherche), nous avions uns stagnation rapide du WCET mesuré. Cette fois, il existe un deuxième palier beaucoup plus élevé de la taille de la population. L'impossibilité pour l'algorithme de parcourir l'espace de recherche et de trouver une meilleure valeur du WCET plus tôt ne s'explique pas par l'implémentation de la fonction *mutex*. C'est en réalité un effet de bord lié uniquement à la population initiale et plus précisément à la graine pseudo-aléatoire utilisée. Il

s'avère que dans l'ensemble des 10 générations d'algorithmes réalisés par point, les valeurs pseudo-aléatoires tirées étaient défavorables. Une ré-exécution a été réalisée par la suite afin de confirmer cette hypothèse et les courbes obtenues sont à nouveau lisses une fois la taille de la population de 128 atteinte.

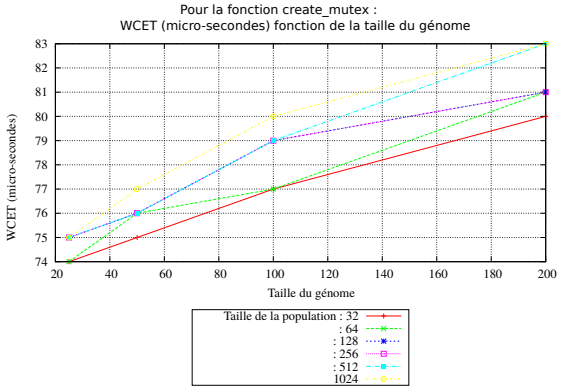

Fig. 7.14 – WCET (taille du génome) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create mutex

La figure 7.14 représente le WCET mesuré en fonction de la taille du génome. Chaque courbe représente une taille de population différente. Nous pouvons conclure sur celle-ci qu'il existe une influence du contexte du système : plus celui-ci est élevé, et plus le WCET obtenu est important. C'est à nouveau la fragmentation mémoire qui impacte le plus le système. Il est intéressant de noter que la fonction *mutex* prend plus de temps à s'exécuter que les fonctions précédentes. Le phénomène observé est identique aux précédents.

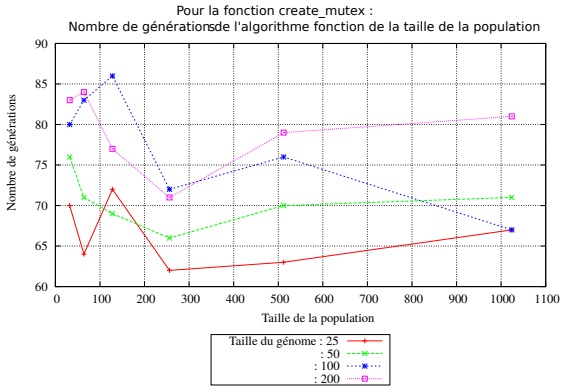

Fig. 7.15 – Nombre de générations (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create mutex

La figure 7.15 représente le nombre de générations de l'algorithme en fonction de la taille de la population. Chaque courbe représente une taille de génome différente. Nous constatons une forte augmentation du nombre de génération de l'algoirthme lors de l'analyse de petites populations. La particularité de cette fonction est la chute immédiate et ce, quelle que soit la taille du génome ou du nombre de générations de l'algorithme lorsque la population 256 est mesurée. Nous retrouvons ici le phénomène de seuil observé sur la figure 7.13. Nous voyons que l'algorithme génétique trouve rapidement un maximum local lorsque la population est relativement grande. Il s'avère que le

seuil précédemment observé déclenche un phénomène de convergence rapide de la population vers un WCET mesuré local. De manière globale, le temps d'exécution de l'ensemble des algorithmes génétiques reste plus élevé pour une forte population (>64) et une taille de contexte élevée (>100) que pour une population plus faible.

Pour conclure, cette fonction dépend du contexte d'exécution et plus précisément de la fragmentation mémoire. Néanmoins, bien que trouvant automatiquement un WCET mesuré proche du réel (taille de population 1024 et génome 200), nous avons pu observer que notre algorithme est fortement influencé et influençable par la population initiale. L'absence d'un choix purement aléatoire met en avant le risque de trouver des résultats incohérents ou en deçà de la valeur recherchée. Ceci montre également l'importance de la compréhension des courbes analysées et de la ré-exécution complète de l'algorithme plusieurs fois pour obtenir une valeur statistique.

7.2.6 create recursive mutex

Un *mutex* peut être utilisé de manière récursive à plusieurs reprises. Le mutex ne redevient disponible que lorsque le détenteur du sémaphore a appelé *xSemaphoreGiveRecursive ()* à chaque fois qu'une demande est réussie. Par exemple, si une tâche 'prend' avec succès le même mutex 5 fois, il ne sera pas disponible pour toutes les autre tâches, tant que la tâche n'aura pas «donné» le mutex retour exactement cinq fois.

Ce type de sémaphore utilise un mécanisme d'héritage de priorité de sorte qu'une tâche qui "prend" un sémaphore DOIT TOUJOURS «redonner» le sémaphore de retour une fois qu'il n'est plus nécessaire.

La figure 7.16 représente le WCET en fonction de la taille de la population. Chaque courbe représente une taille de contexte différente. Comparé aux courbes de la fonction *create mutex*, nous relevons une valeur de WCET différente de deux micro-secondes. Le code utilisé est pourtant fortement commun aux deux fonctions. Ce que nous observons particulièrement sur ce graphique est la différence de fonctionnement de notre algorithme génétique. Nous remarquons une difficulté de celui-ci à obtenir la pire valeur rapidement. En effet, nous notons un phénomène de “palier” entre les différentes tailles de populations. Ainsi l'algorithme génétique stagne à un maximum local lorsque les populations sont (avec) de faibles valeurs. La valeur maximale observée augmente subitement lorsqu'un seuil est atteint.

Lors des expérimentations, nous avons observé que la fonction *create recursive mutex* a de légères différences de code qui ont un impact direct sur les valeurs obtenues. Ainsi la valeur maximale observée est de 85 micro-secondes mais les paliers de stagnations ont été réguliés. Ainsi une population de 200 obtient la valeur maximale que prend la fonction. (Des tests ont été effectués avec de grandes populations pour garantir ces résultats). Ce n'est pas le cas de la fonction *create mutex*, qui, au cours d'expérimentations avec une population plus grande, a obtenu un score de 85 micro-secondes soit 1 micro-seconde de plus que ce qui avait été observé auparavant.

Le graphique 7.16 montre donc une limite de notre algorithme génétique. Il faut prendre en compte avec grande attention la taille de la population. Même si pour toutes les autres fonctions, la taille est suffisante pour les mesures, il existe un cas particulier qui nous a fait sous-estimer le WCET d'une fonction. Par chance, une autre fonction quasi-similaire a été influencée plus fortement par notre algorithme génétique. Il est intéressant de noter que les nombres pseudo-aléatoires utilisés ne sont pas identiques entre les deux fonctions testées. Ceci a également joué sur les valeurs obtenues.

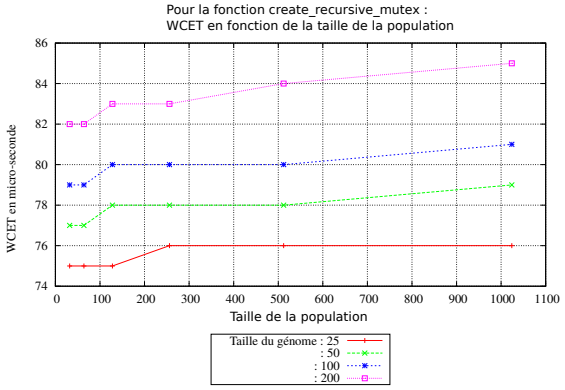

Fig. 7.16 – WCET (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create recursive mutex

Nous voyons ensuite, sur le graphique 7.17 représentant le pire temps d'exécution mesuré en fonction de la taille du contexte, que l'état du système réagit de la même manière que sur la fonction *create mutex* étudiée précédemment. Nous observons à nouveau une évolution de notre algorithme et une augmentation significative de la valeur du WCET lorsque la taille du contexte augmente. La fragmentation mémoire déjà décelée et observée plusieurs fois auparavant est la raison principale de cette augmentation progressive de la valeur du temps d'exécution de la fonction mesurée.

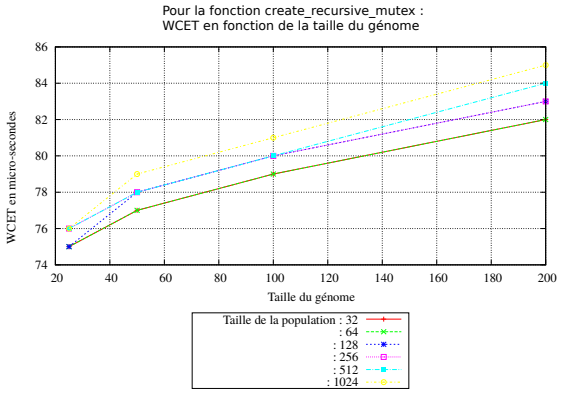

Fig. 7.17 – WCET (taille du génome) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create recursive mutex

Le graphique 7.18, représentant le nombre de génération de l'algorithme en fonction de la taille de la population et contenant plusieurs courbes définies par la taille du contexte, diffère de celui de la fonction *create mutex*. En effet, les phases de stagnation que nous avons observé sur le graphique 7.16 se retrouvent sur ce graphique. Nous observons que l'algorithme génétique a pris beaucoup de temps (et donc de générations) pour réussir à extraire un nouveau temps d'exécution supérieur à l'ancien. Lors des analyses des autres courbes représentant le nombre de génération en fonction de la population, nous observions peu de générations pour les tous petits contextes et pour les grands contextes (dans les cas ou la fragmentation mémoire a une influence directe sur la valeur du temps d'exécution). A l'inverse un nombre important de génération est obtenu pour les valeurs de contexte intermédiaire. Dans le cas de la fonction *create recursive mutex*, nous observons une différence notable. La plus grande population (124) augmente significativement son nombre

de générations). Cela s'explique par l'obtention, par l'algorithme, d'un nombre d'augmentation de la valeur du WCET obtenu très élevé. Chaque valeur étant supérieure à la précédente, mais fortement proche de celle-ci, l'algorithme réitère les opérations pour maximiser la valeur. Nous observons qu'il est donc possible pour notre algorithme de stagner sur des "plateaux" en évoluant lentement jusqu'à une valeur maximale. Pour autant, le temps de convergence reste très difficile à prévoir pour l'ensemble des fonctions.

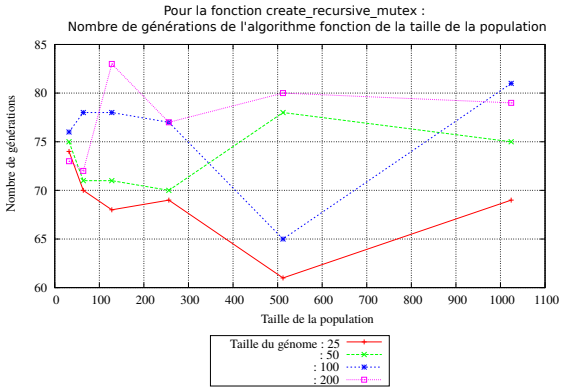

Fig. 7.18 – Nombre de générations (taille de la population) pour tournoi de 4, croisement de 0.2, mutation de 0.01, pour la fonction create recursive mutex

7.3 Analyse et méthodologie

Nous suivons pour ces expérimentations la méthodologie déjà utilisée par la communauté scientifique. Nous respectons particulièrement la reproductibilité énoncée par [Joh01]. L'algorithme génétique respecte la règle des statistiques et a également réalisé un ensemble de tests supérieur à 10 exécutions complètes. En l'occurrence, nous avons réalisé l'algorithme génétique de Gross [Gro03] sur les connaissances transmises dans ses publications de recherche. Cependant, les valeurs obtenues sont fortement inférieures aux valeurs émisent par notre algorithme génétique à cause de l'absence de contexte. C'est la raison pour laquelle nous ne comparons pas directement notre algorithme à celui de Gross. L'autre algorithme expérimenté est l'algorithme aléatoire. Les résultats obtenus sont en deçà de l'algorithme de Gross, cependant,l'algorithme aléatoire reste peu comparable au nôtre, toujours en raison de l'absence de contexte.

L'algorithme génétique que nous avons réalisé est efficace pour détecter et exploiter le contexte d'exécution d'un système. Il est simple d'utilisation et limite au maximum la présence de personnel. En effet, pour analyser l'ensemble des 13 primitives choisies dans freeRTOS (dont certaines viennent d'être présentées ici) il a fallu deux mois. L'ensemble de l'analyse comprend 3120 exécutions de l'algorithme génétique complet (c'est-à-dire lorsque le critère d'arrêt a été uniquement celui de la stagnation de la valeur du WCET depuis 50 générations de l'algorithme). Cela équivaut à une moyenne de 2,3 exécutions complètes de l'algorithme génétique par heure. La création d'un script d'exécution en python et en C++ a permis de gérer de manière automatique et sans intervention humaine l'ensemble des primitives.

La mise en oeuvre de l'algorithme génétique, les codes nécessaires à la mise en place des systèmes embarqués en parallèle, tous synchronisés ainsi que l'écriture des différentes fonctions étudiées,

a pris six mois pour un ingénieur. L'adaptation d'un nouvel ingénieur à la mise en oeuvre et l'exploitation de cet algorithme est d'un maximum de deux jours. De plus, le code est maintenant déjà écrit.

Ainsi, lors de l'exécution de notre algorithme, celui-ci ne demande qu'un contrôle visuel quotidien rapide (quelques secondes) afin de vérifier que l'ensemble des plates-formes testées sont en état de marche et que le système principal réalisant l'algorithme génétique n'est pas en défaut. La seule difficulté est la mise en place du système. En effet, il faut légèrement implémenter les codes nécessaires aux systèmes embarqués pour communiquer via UDP, et les paralléliser. Il faut également implémenter la partie adaptant l'algorithme génétique aux fonctions existantes sur les systèmes embarqués. Ce qui prend environ une semaine pour un système complexe.

L'interprétation des courbes peut se faire à tout moment puisque l'ensemble des valeurs obtenues pour chaque algorithme est enregistré sur un serveur distant permettant la génération automatique des graphes et figures présentés ici.

En tout, nous avons réalisé 18 mois d'évaluations sur les différentes primitives de FreeRTOS et nous en présentons les plus notables. Ces 18 mois n'ont pas tous été fructueux. En effet, certains bogues d'implémentations nous ont fait perdre une partie des données distantes et ont dû être ré-évalués. Cependant, il n'y a pas eu de bogues majeurs sur l'algorithme génétique. De plus, nous avons continué le développement de l'algorithme en parallèle des évaluations. Sur les 18 mois d'évaluations, quatre concernent la dernière version de notre algorithme génétique. Les autres concernent des algorithmes qui nous ont amené à pousser nos réflexions et à vous présenter notre dernière proposition.

Les systèmes embarqués évalués non pas eu de défaut majeurs, et ne sont pas restés dans des états non-récupérables de manière automatique. Les plates-formes ne sont pourtant ré-initialisables qu'à la main. Aucune intervention humaine n'a donc été nécessaire lors de cette phase. Cependant, les plate-formes sont tombées en panne une seule fois, à cause d'une mauvaise manipulation qui entraîna un court circuit entre plusieurs plate-formes non-isolées les unes des autres, car il n'existait pas de banc d'essai séparant les différentes plates-formes testées (placées en tas les unes sur les autres). Il reste nécessaire d'être vigilant à ce type de problème, bien qu'il ne soit intervenu qu'à la mise en place du système global.

Ces nombreuses évaluations nous ont permis de voir que certaines fonctions sont insensibles au contexte d'exécution du système. En effet, lorsque celles-ci ne font intervenir aucune politique de gestion dynamique d'un bus ou de périphérique particuliers, leur temps d'exécution est stable. Notre algorithme génétique n'a donc pu faire évoluer cette valeur. Néanmoins, il est impossible de savoir si l'algorithme a trouvé directement le WCET mesuré ou s'il est resté bloqué dans un WCET local sans réussir à réaliser la séquence nécessaire de primitive lui permettant d'atteindre un autre état du système. La seule façon de garantir que les résultats obtenus sont corrects est la connaissance, même partielle du rôle de la fonction exécutée.

7.4 Conclusion

Dans ce chapitre, nous avons observé l'efficacité de notre algorithme génétique et nous avons mis en avant l'influence du système sur le temps d'exécution des différentes fonctions. Nous avons pu particulièrement éprouver notre algorithme génétique, qui a toujours, de façon relativement

rapide, trouvé une valeur de temps d'exécution avec un résultat proche de celui attendu. Comme tout algorithme, il peut être amélioré et les pistes de réflexions sont données au chapitre suivant. Néanmoins notre algorithme donne satisfaction, comparé aux algorithmes aléatoires et autres algorithmes génétiques utilisés actuellement. Nous avons offert une résolution scientifique au problème de l'analyse du temps pris par un contexte d'exécution dans un système complexe, par méthode dynamique.

Une réflexion se pose sur le comportement de l'algorithme génétique en fonction des primitives considérées et des paramètres utilisés pour la génération aléatoire des contextes. Il existe en effet, du à l'exploitation d'un système de génération pseudo-aléatoire, le risque que celui-ci réalise un biais influençant les résultats de WCET mesuré. Actuellement, nos expérimentations n'ont pas été en mesure de mettre en avant ce phénomène, mais la problématique reste entière. Pour des raisons de reproductibilité, l'algorithme ne peut s'envisager avec un système purement aléatoire. Il faudrait "éprouver" des graines pseudo-aléatoires et démontrer que celles-ci sont suffisamment diversifiées les unes des autres pour générer des populations différentes.

Une autre solution serait d'utiliser un algorithme pseudo-aléatoire différent pour les différentes parties de l'algorithme génétique, également avec des graines de générations différentes. Cela oblige à faire de nombreux jeux statistiques mais reste réalisable techniquement. L'ensemble de ces voies reste clairement à explorer sur des systèmes où l'influence serait avérée.

Chapitre 8

Perspectives et travaux à venir

Sommaire

Les expérimentations précédentes montrent un intérêt particulier à l'évolution de notre algorithme génétique. Celui-ci est éprouvé sur petit système et nécessite encore quelques améliorations afin de pouvoir remplir pleinement son rôle. Ce chapitre a pour objectif de détailler les travaux à venir et de conclure sur cette partie II. Celle-ci nous a permis d'observer un problème industriel complexe et de mettre en place une solution adaptée en résolvant les difficultés de recherche du pire temps d'exécution.

8.1 Perspectives

Les perspectives de nos travaux sont nombreuses. La première serait d'utiliser un système connu des algo-généticiens qui permettrait de faire évoluer notre système. La critique majeure qui pourrait être faite est l'évolution des différentes techniques d'optimisation qui existe actuellement. Chaque optimisation répond plus efficacement à un problème plûtot qu'à un autre. Les limites de notre algorithme génétique reste encore à être affinées. Pour se faire, le mieux serait d'utiliser irace [16]. Ce logiciel permet de trouver la meilleure méthode d'optimisation en fonction du problème rencontré. Il met en concurrence tous les algorithmes heuristiques et sélectionne au fur et à mesure des itérations les méthodes les plus appropriées pour finir par la meilleure solution.

Nous proposons d'intégrer notre algorithme génétique à ce module afin de comparer celui-ci sur des cas concrets. Il nous permettra de faire évoluer notre système et d'améliorer nos critères de recherches.

Nous souhaitons également améliorer notre algorithme génétique sur les points suivants :

— taille de la population ;
— taille du contexte ;

16. http ://iridia.ulb.ac.be/irace/

— taux de croisement ;
— taux de mutation ;
— politique de mutation.

Les résultats obtenus montrent l'importance de ces paramètres et une meilleure connaissance de ceux-ci est primordiale.

Taille de la population : La taille de la population est directement dépendante de l'espace de recherche qu'il est possible d'observer, celle-ci doit donc être la plus grande possible. Néanmoins, une trop grande population ne permettra pas une évolution significative de notre problème et aura un coût en matière de temps d'exécution prohibitif. Il faut donc trouver un juste milieu en fonction du problème rencontré. Cela est difficile sans connaissance précise du système étudié. Il faudrait pour cela que la taille de la population soit variable durant l'exécution. Ainsi deux idées se développent.

La première est de prendre une population très grande et de réduire sa taille lors d'une convergence de l'algorithme génétique. Cela a l'inconvénient d'avoir un coût prohibitif dès le début de l'algorithme génétique, et ce, sans garantie que la population en question soit déjà suffisamment grande.

La deuxième solution est la mise en place d'une population évolutive inversement proportionnelle. Lorsque l'algorithme génétique converge, nous augmentons la taille de la population pour entreprendre d'autres voix possibles. Cette solution semble meilleure concernant le temps d'exécution, et la taille de la population ne grandie que tant qu'il y a une évolution sur le résultat de notre algorithme.

Taille du contexte :

L'ensemble de nos avancées majeures dans l'algorithmie génétique se trouve dans la génération de contexte. C'est naturellement vers ce point que nous souhaitons également porter nos efforts. Les résultats précédents ont montré tout l'intérêt et l'importance du critère de la taille du contexte. En effet, une fragmentation mémoire peut avoir de grandes conséquences, tout comme l'initialisation d'une fonction. Nous observons donc des problèmes où la taille du contexte d'exécution doit être faible et des cas où un très grand contexte a des conséquences fortement élevées.

Cette thèse développe ce phénomène, mais pour le rendre pleinement utilisable dans l'industrie, il faut que ce paramètre (la taille du contexte) soit géré, tout comme la taille de la population, de manière automatique. Il est possible d'envisager une évolution dynamique de la taille du contexte en fonction de l'évolution de l'algorithme génétique. Tout d'abord, l'algorithme génétique gérera une taille de contexte variable. Par la suite, à l'état initial de l'algorithme, chaque chromosome pourrait avoir une taille de contexte tirée au hasard.

Cela implique des difficultés au niveau de la mutation et du croisement des individus, comme croiser deux contextes de taille variable. Nous avons un certain nombre de pistes et de premiers résultats à ce sujet. Si le pire temps d'exécution mesuré est plus élevé sur un contexte important, alors la taille du contexte des fils sera élevé. Dans le cas contraire nous obtenons le résultat inverse. Cette approche est intéressante pour converger plus rapidement vers une solution. Mais la population, lors de nos mesures, a montré une perte de diversité et donc une convergence vers une valeur locale. Une autre solution, plus pratique, est la mise en place d'un fils au contexte court et d'un fils au contexte long, comme les parents. La vitesse de convergence en est fortement impactée mais les premiers résultats sont prometteurs.

Il faut néanmoins développer et expérimenter fortement ce taux de croisement afin d'obtenir les meilleurs résultats possibles. L'évolution de ce paramètre semble prioritaire si nous souhaitons le comparer via irace.

Taux de croisement :

Le taux de croisement dépend également du problème à résoudre. Selon la fonction à tester, nous observons dans des courbes obtenues lors de l'évaluation des primitives, que le paramètre influe sur les résultats.

Les valeurs que nous avons pu recueillir sont légèrement différentes de ce qui avait été proposé par Gross avant nous. Visible figure 6.8, nous observons que le taux de croisement a une influence sur la vitesse de convergence et permet donc à l'algorithme de converger plus vite et d'obtenir un maximum local, ou de trouver le pire temps d'exécution mesuré (en fonction de la taille de la population).

Pour une population importante (environ 500), un taux trop élevé (> 0.3) doit impacter sur la diversité génétique de la même manière qu'un taux de mutation trop grand (> 0.05). Il est donc préférable de prendre une valeur du taux de croisement intermédiaire. Nous avons donc pris un taux de 0.2.

Il est possible que le taux de croisement change en fonction de la plate-forme et des fonctions à tester. La solution que nous proposons consiste a évaluer ce taux au démarrage de l'algorithme, par un test rapide, sur une petite population. Celui-ci serait fixé en même temps que le taux de mutation.

Taux de mutation :

Le taux de mutation a été déterminé de la même manière que notre taux de croisement pour obtenir au final une valeur de 0.01. Il est préférable de déterminer ce taux avant chaque exécution concrète de l'algorithme génétique sur une fonction donnée. Nous proposons donc de l'évaluer en amont.

Politique de mutation :

Nous avons relevé lors de nos expérimentations qu'une technique différente de mutation pourrait avoir un effet intéressant sur nos résultats. En effet, la mutation des paramètres est très particulière. elle intervient au bit près et a donc un impact significatif sur les bits de poids fort et négligeable sur les bits de poids faible. Il serait intéressant de mettre en place un dispositif qui influence moins les résultats obtenus.

Il existe déjà plusieurs solutions qu'il faudrait évaluer et comparer, comme prendre une valeur diminuée ou augmentée d'un certain pourcentage pour des valeurs numériques, par exemple. Il faut néanmoins que nous adaptions ce problème aux contraintes des variables liées et aux paramètres de type pointeur. Cela reste complexe : les pointeurs sont déjà gérés dans notre algorithme génétique, comme nous avons pu le détailler précédemment, et restent un cas particulier de notre système.

8.2 Travaux à venir

Les travaux à venir sur notre algorithme génétique sont donc encore nombreux mais fortement envisageables et réalisables. Confronter clairement notre algorithme génétique aux autres méthodes

heuristiques devrait asseoir l'idée de l'importance de la prise du contexte d'exécution dans nos mesures. Il faut donc commencer par évoluer la taille du contexte et celle de la population pour obtenir un résultat plus facilement évolutif par rapport aux différentes plates-formes étudiées. Les taux de croisement et de mutation vont parfaire les évolutions et obtenir des résultats au plus proche du WCET réel.

La seule difficulté majeure est le temps à prendre pour l'implémentation de ces modifications et l'interprétation des résultats obtenus.

8.3 Conclusion

Cette partie II de la thèse nous a permis de déterminer une méthode pour mettre en place une solution dynamique afin de trouver le pire temps d'exécution mesurable sur plate-forme réelle. La prise en compte du contexte d'exécution nous a permis de démontrer l'efficacité des algorithmes génétique à trouver une solution acceptable dans un temps donné. Il est envisageable de porter notre solution facilement sur tout type de plate-forme temps-réel nécessitant le contrôle des valeurs de temps d'exécution relativement rapidement. Le contrôle de ce paramètre permet une meilleure sûreté de fonctionnement et sécurité du système étudié. Mais le temps n'est pas le seul paramètre à prendre en compte sur un système de type satellite. L'évolution des systèmes complexes multi-coeur amènent d'autres difficultés telles que la contention.

Troisième partie

Gestion de la mémoire et code malveillant

Chapitre 9

Les attaques mémoires pendant l'exécution

Sommaire

Nous présentons dans cette partie III, indépendante scientifiquement de la partie II, notre second apport majeur. Nous y développons plus particulièrement la problématique de contention mémoire sur systèmes multi-cœurs. En effet, la contention mémoire peut engendrer des problèmes de communication et des impacts temporels importants. Dans ce premier chapitre concernant la contention mémoire, nous développons son fonctionnement général, puis l'état de l'art relatif à ce domaine, pour finir par les attaques réalisables sur le système. Au chapitre suivant, nous traitons par la suite notre apport et notre solution au problème. Nous décrivons d'abord l'intérêt de notre apport puis nous modifions le système afin de l'intégrer. Nous présentons à ce moment un système électronique permettant la détection de contention mémoire. Le dernier chapitre permet de valider de notre concept sur plate-forme réelle puis de terminer par un ensemble d'idées de poursuite de recherche pour finalement conclure cette thèse.

9.1 Introduction

Nous avons exploré dans le chapitre 3 de la partie I différents types d'attaques sur les système aérospatiaux. Nous avons par la suite présenté une solution pour améliorer l'observation du temps d'exécution d'une fonction. Nous avons pu observer qu'il est possible de fortement influer sur le temps d'exécution avec le contexte d'un système. Nous cherchons maintenant à explorer un contexte spécifique pouvant rendre imprédictible l'ordonnancement et le temps d'exécution d'un système, même s'il existe une vérification du temps par algorithme génétique. Notre cas d'étude se portera au niveau de la mémoire sur un phénomène connu appelé contention mémoire (à ne pas confondre avec la fragmentation mémoire vu précédemment) et devenu particulièrement problématique avec l'arrivée du multi-cœur. La question que nous nous posons dans ce chapitre est *Comment détecter voire empêcher les accès importants en mémoire de la part d'une fonction au détriment des autres dans un environnement multi-coeur ?*

9.2 Contention mémoire

9.2.1 Introduction

Historiquement, les architectures logicielles du monde aérospatial ont été développées pour interagir avec les satellites en respectant les contraintes d'embarquabilité, de minimisation des coûts et de temps-réel. Plus récemment, les nouveaux systèmes virtualisés tel qu'XtratuM [MRPC10] ont été intégrés aux sein des satellites.

L'utilisation de systèmes multi-coeurs s'est imposé en vue de diminuer les coûts et d'augmenter les performances du système. En utilisant la virtualisation sur de telles architectures, il est possible d'exécuter des partitions (systèmes d'exploitations) temps-réels en parallèle sur un même processeur. L'hyperviseur doit alors assurer un isolement spatial et temporel entre les différents systèmes pour garantir le respect des contraintes de sûreté et de sécurité.

Les hyperviseurs développés pour des systèmes embarqués, attribuent de manière exclusive un ou plusieurs cœurs à chaque partition virtualisée. La mémoire, elle, est découpée en plages attribuées statiquement à chaque partition virtualisée. Les mécanismes de protection mémoire sont utilisés pour assurer l'isolation spatiale et temporelle. Si ces solutions assurent le partitionnement de la mémoire et du processeur, des ressources comme les bus et certains caches continuent d'être partagées. Un processeur qui effectue un grand nombre d'accès mémoire peut provoquer de la contention sur les bus et polluer les caches partagés. Cela induit une gêne sur le fonctionnement des autres cœurs.

La figure 9.1 montre le fonctionnement général d'une contention mémoire sur un système aérospatial. Nous observons sur ce schéma deux processeurs. Ceux-ci peuvent communiquer entre eux via un bus de données. Pour cet exemple, le bus est de type AMBA AHB. Ces deux processeurs sont reliés à la mémoire via un cache de niveau 2, chaque processeur ayant un cache de niveau 1 non partagé. Il se trouve sur ce bus un contrôleur mémoire qui distribue l'utilisation via le principe du *roundRobin*, c'est-à-dire que les processeurs peuvent parler sur le bus chacun leur tour.

La contention mémoire consiste, pour un processeur générant de la contention, à demander systématiquement des informations à la mémoire. Pour que ce cas soit réalisable, il doit invalider

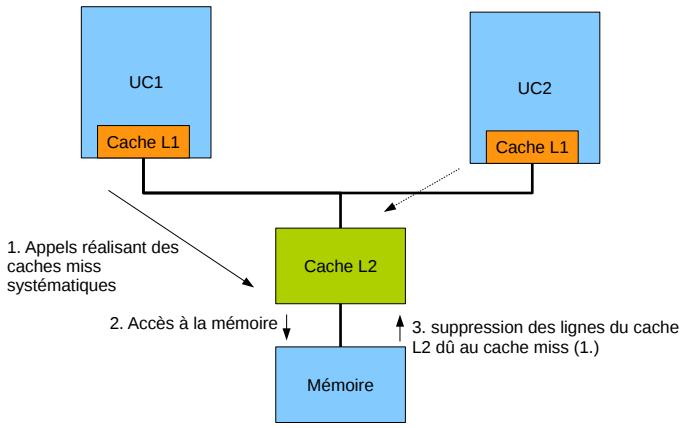

Fig. 9.1 – Contention mémoire avec système virtualisé

son cache de niveau 1, c'est-à-dire demander des accès à des informations non disponibles dans celui-ci. Ainsi l'ensemble de ces demandes sera traité par le cache commun de niveau 2. Au mieux le cache de niveau 2 est en mesure de répondre aux requêtes de ce processeur, et la contention est limitée. Dans le cas contraire, il se produit un accès en mémoire et le cache de niveau 2 s'en trouve également modifié.

Le processeur voisin souhaitant accéder à la mémoire se trouve ainsi impacté par les demandes répétées du premier processeur. Lorsque le processeur voisin tente d'accéder à une information qui doit se trouver en cache de niveau 2 et ainsi gagner du temps, il doit attendre que la mémoire soit disponible et puisse fournir son information au processeur et au cache de niveau 2.

L'intérêt des caches est d'optimiser le temps d'exécution et donc de rendre les systèmes plus performants, au point que l'ordonnancabilité d'un système dépend même de cette optimisation. Une faille trop prononcée pourrait donc mettre en défaut un système ou empêcher une tâche d'obtenir l'ensemble des bonnes informations dont elle a besoin pour un fonctionnement normal. Il existe pourtant un certain nombre de solutions à ce problème.

9.2.2 Travaux actuels

La contention mémoire est présentée dans l'article [ENBSH12]. Elle est souvent étudiée dans le domaine de la sûreté de fonctionnement afin de garantir l'isolation spatiale et temporelle d'un système, par exemple. La contention mémoire est également développée dans [DHW97]. Ce phénomène peut apparaître sous différentes formes, mais la plus étudiée est celle présentée sur les multi-processeurs.

Une attaque

Dans notre domaine, certains travaux ont déjà été effectués sur notre plate-forme. En effet, les développeurs de la plate-forme Leon3 (www.gaisler.com), processeur que nous utilisons dans le domaine spatial, ont déjà effectué un certain nombre de travaux afin de comprendre les conséquences d'une intrusion malveillante. Le problème de l'intrusion reste proche de la problématique de la

contention mémoire. Partis d'une hypothèse équivalente à la nôtre, ils ont cherchés dans un premier temps à observer les problèmes que cela entraînerait sur la sûreté de fonctionnement, puis ont émis l'hypothèse d'un acte malveillant [Sev]. Le système proposé est un mono-processeur, et l'article se tourne uniquement vers un critère de faisabilité d'une attaque par intrusion sans détail précis sur celle-ci. Réalisé par une partie des développeurs du processeur, il met en avant les qualités du processeur et la conclusion reste avantageuse commercialement.

L'acte malveillant proposé, qui fait partie de la catégorie des attaques de type invasive, n'est pas observé de façon habituelle. En effet, classiquement, les systèmes de détection de code malveillant traditionnels utilisent un logiciel pour combattre ce type d'attaque. Dans le papier fourni par les ingénieurs de Gaisler, c'est un composant électronique qui est proposé. Ce composant est mis en œuvre afin de gérer en interne l'exécution des programmes. Il fournit donc une valeur correspondant à l'analyse du programme en cours d'exécution. Il scrute le PC (*Programme Counter* ou compteur de programme) qui observe l'enchainement des instructions qui doivent être exécutées. Si un écart trop important est observé avec ce qui avait été prévu initialement, alors le composant réagit. L'ensemble des travaux est simulé ou implémenté en parti sur FPGA. Les logiciels testés sont, quant à eux, développés sur simulateur.

L'attaque se limite a un scénario de menace extérieure mettant en œuvre un code souhaitant s'insérer lors de l'exécution d'un autre programme. Celui-ci est observé et lorsque le nombre d'instructions anormales est considéré comme trop important, l'attaque est considérée comme réussie. Seul un signalement de la réussite est obtenu par la suite. La valeur servant à déterminer le dépassement ou non du nombre d'instructions malveillantes est déterminée avant exécution du programme lors d'une phase dite de "compilation". La détermination réelle de ces valeurs n'est pas explicitée. Néanmoins, l'utilisation d'un composant électronique a permis de faire une observation temps-réel et non interruptive du système lors de son exécution. C'est un des objectifs que nous nous fixons.

Le composant démontre la faisabilité d'une détection d'intrusion sur le système que nous utilisons, bien qu'en mono-processeur et n'observant pas exactement le même phénomène. De plus, il met en avant dans sa conclusion que le problème n'est pas résolu mais qu'il serait bon d'explorer la voie d'un système électronique, afin de mieux gérer certains phénomènes, notamment la contention. Fort de ces travaux introductifs, qui plus est sur notre plate-forme, nous explorons cette voie par la suite pour gérer la contention mémoire.

Un autre cas d'utilisation de notre plate-forme spatiale

Un autre article développe une vision de l'utilisation de la mémoire sur notre plate-forme spatiale [CCP+14]. En effet, un groupe dont font partie les développeurs de l'hyperviseur XtratuM [MRPC10] a mis en avant une idée nouvelle sur la communication avec la mémoire. Ils partent d'un système qui nécessite un niveau de criticité mixte des différentes partitions s'exécutant dessus. Pour réaliser cela sur une seule et même plate-forme, et conserver l'isolation spatiale et temporelle des interférences des applications, la plate-forme spatiale doit être modifiée.

Cette thèse discute de la possibilité d'intégrer un contrôleur mémoire dynamique en plus du contrôleur mémoire statique déjà existant. Celui-ci doit faciliter l'intégration d'un niveau de criticité mixte entre les applications et offrir un meilleur taux de performance. Ce document évalue le résultat sur une plate-forme XtratuM correspondant parfaitement à nos besoins spatiaux. Il met également en œuvre le multi-processeurs.

La capacité de traitement élevée qu'offre les systèmes embarqués multi-cœurs nous permet au-

jourd'hui d'exécuter plusieurs applications sur une seule et unique plate-forme matérielle. Toutefois, une partie des applications intégrées peut avoir des contraintes temps-réelles strictes nécessitant une preuve formelle afin de garantir que les délais soient correctement respectés. D'autres parties du système peuvent êtres, néanmoins, moins exigeantes. Seules les parties critiques nécessitent donc une certification. Pourtant l'obtention de celle-ci peut être très difficile en raison du partage avec des applications non certifiées de la plate-forme matérielle et logicielle. Le papier [CCP+14] tente de réduire les interférences spatiales et temporelles du système. Les chercheurs prennent en compte les difficultés d'une plate-forme multi-processeur en utilisant la MMU (*Management Memory Unit* ou gestionnaire de mémoire) [**?**]. La MMU permet d'assigner un espace mémoire fixe à chaque application. De cette façon, la mémoire est protégée d'accès malveillants d'une partition sur un espace mémoire qui ne lui est pas autorisé. Mais cette protection est inefficace contre notre problématique. Bien que l'espace mémoire de chaque partition soit limité, cela n'empêche pas de perturber le cache de niveau 1 de son processeur pour appeler continuellement la mémoire.

La papier [CCP+14] met en avant l'existence potentielle de la faille de la contention mémoire sur notre plate-forme spatiale et exprime l'ensemble des contres-mesures existantes sur un système multi-coeur léon3 avec un hyperviseur XtratuM. La contention mémoire fut mise de côté car leurs recherches se sont focalisées sur l'isolation spatiale et temporelle. Comme dans le papier [Sev], la solution trouvée fut un composant électronique, afin d'améliorer les performances et de pouvoir travailler sans interrompre le ou les processeurs.

9.2.3 Conclusion

Un ensemble de travaux a été fourni sur notre plate-forme spatiale et a permis de conclure sur l'absence de solutions vis-à-vis de la contention mémoire. Il s'avère que les spécificités de notre plate-forme spatiale facilite l'utilisation de composants électroniques pour répondre à des problématiques proches de la nôtre. Ces recherches justifient un apport sur la gestion de la contention mémoire sur système multi-coeur virtualisé avec présence de cache. Il faut définir les scénarios d'attaques afin de déterminer précisément la meilleure solution envisageable.

9.3 Nos scénarios d'attaques relatifs à la contention

9.3.1 Introduction

Le scénario d'attaque est un fil conducteur permettant de déterminer exactement les cas possibles ou improbables. Ainsi, en fonction des capacités de l'attaquant, nous cherchons à déterminer la meilleure contre-mesure possible. Celle-ci doit être la plus simple et la moins coûteuse possible. Elle doit être en mesure de ralentir l'attaque suffisamment longtemps pour en supprimer tout l'intérêt ou de bloquer celle-ci en évitant d'en créer de nouvelles.

9.3.2 Scénario 1 : Un hyperviseur unique

Notre premier scénario est représenté figure 9.2. Il met en œuvre un système muni de plusieurs processeurs et d'un hyperviseur unique. Celui-ci gère les différentes applications intervenant sur son cœur mais aussi sur les autres. En effet, il distribue au mieux les partitions en fonction de leur

ordonnancement sur les différents processeurs. Cette distribution des partitions peut se faire sur d'autres coeurs (si le nombre de coeur est supérieur à 2), ce qui laisse envisager un coeur entièrement dédié à l'hyperviseur. Celui-ci peut donc fonctionner la majeure partie du temps et évaluer rapidement l'ensemble des partitions en cours d'exécution ou répondre à leurs sollicitations. Grâce à un mécanisme supplémentaire au niveau de la contention mémoire, l'hyperviseur pourrait détecter un dépassement du nombre d'accès en mémoire et donc sanctionner une partition défaillante en conséquence.

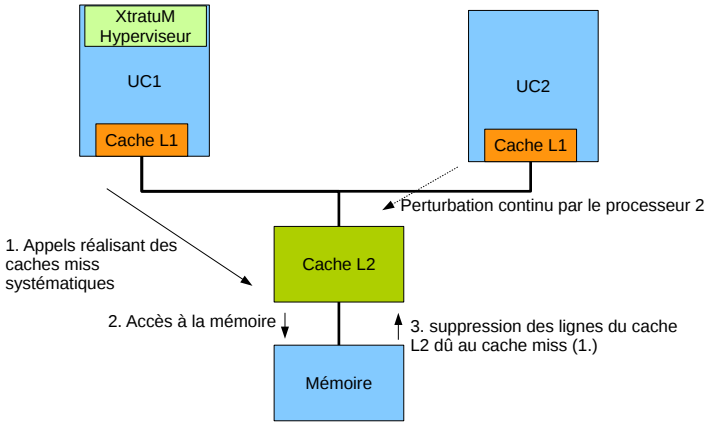

Fig. 9.2 – Hypothèse 1, contention avec un hyperviseur

Notre cas d'étude est un cas idéal, nous ne traitons pas ici les coûts de communications inter-processeurs. Nous voyons qu'il est possible de prendre en compte la contention mémoire dans l'hyperviseur mais qu'il lui manque potentiellement un composant pour l'aider à observer les accès mémoires. En effet, il ne pourrait, au mieux, qu'observer le bus commun aux processeurs et mémoires avec l'obtention de privilèges qui lui sont propres, mais il ne pourrait savoir si le cache de niveau 2 a répondu à la requête ou si c'est la mémoire qui l'a fait. Il ne peut donc discriminer le composant ayant réellement répondu et déterminé s'il y a contention mémoire. Ainsi l'écoute du bus, idée simple et efficace pour des systèmes limités à un cache de niveau 1 propre a chaque processeur, ne peut être considéré dans notre cas. Nous constatons la nécessité d'une aide à l'hyperviseur à situer au niveau du cache de niveau 2.

L'attaque envisageable de ce type de système, que nous étudierons, peut prendre la forme d'un coeur qui est infecté par une partition souhaitant faire un maximum d'erreur de cache de niveau 2. Pour cela, la partition tente des communications avec la mémoire à chaque fois que le bus le lui permet. La perturbation engendrée empêche les autres partitions qui travaillent sur un processeur différent d'accéder dans leur temps d'exécution normal aux informations souhaitées dans la mémoire. Ainsi nous prendrons dans ce cas un processeur avec une partition malveillante générant des accès mémoires systématiques et un processeur "innocent" subissant l'attaque. Nous évaluerons sur système réel, par la suite, son impact réel.

9.3.3 Scénario 2 : Un hyperviseur par cœur

Le second scénario d'attaque prend en considération un système avec deux processeurs et cette fois-ci deux hyperviseurs, un par processeur. Deux cas sont envisageables : celui où les deux hyperviseurs sont complets et indépendants, ou bien celui où l'hperviseur est découpé en deux parties. Cette hypothèse est visible avec la figure 9.3.

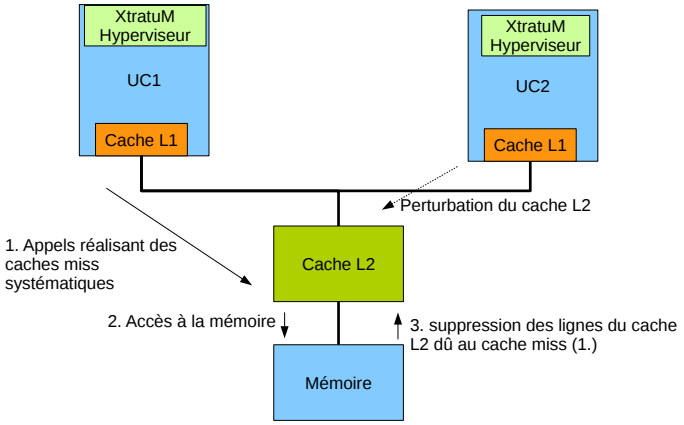

Fig. 9.3 – Hypothèse 2, contention avec deux hyperviseurs

Nous y observons un cache de niveau 2. D'un point de vue contention mémoire, la problématique reste identique à l'hypothèse précédente. Dans le cas d'hyperviseurs indépendants entre chaque processeurs, il n'est plus envisageable d'observer le bus pour connaître les demandes mémoires de chacun d'eux. En revanche, il s'avère nécessaire de mettre en place un système au niveau du cache de niveau 2 pour authentifier ce qui s'arrête au niveau du cache ou ce qui accède réellement à la mémoire. Faire cette différenciation nous permet de savoir dans un premier temps quel processeur tente de communiquer avec la mémoire et combien de fois. Dans un deuxième temps, il permet aux hyperviseurs d'identifier la partition demandeuse au niveau de la mémoire.

La forme de l'attaque envisagée dans ce cas de figure est légèrement différente. En effet, une partition placée sur un cœur est malveillante et tente de rendre aveugle le système de détection. Elle tente de faire des perturbations sur le cache de niveau 2 afin qu'une partition innocente soit détectée comme coupable de contention à sa place. L'objectif est de remplir le cache de niveau 2 de données inutiles pour la partition innocente. Nous développons ce second cas également sur système réel, et profitons des propriétés du système que nous venons de présenter. Le fait qu'il existe 2 hyperviseurs indépendants rend l'attaque plus aisée.

9.3.4 Conclusion

Il existe plusieurs scénarios d'attaques possibles mais il en ressort une conclusion identique. Le système n'est actuellement pas en mesure de détecter ou de traiter le problème de contention mémoire. Une solution envisagée est l'apport d'un composant proche du cache de niveau 2.

9.4 Détection

Nous mettons en oeuvre une plate-forme spatiale aux caractéristiques équivalentes aux figures présentées au chapitre précédent. Celle-ci est basée sur un système multi-processeur Léon 3 (avec deux processeurs) et l'hyperviseur XtratuM. La figure 9.4 extraite de la documentation technique présente le système. Nous y retrouvons un bus commun entre les processeurs appelé AMBA AHB également relié à un contrôleur mémoire pouvant faire office de cache de niveau 2 commun entre les processeurs. Nous retrouvons également un certain nombre de mémoires différentes disponibles derrière ce contrôleur.

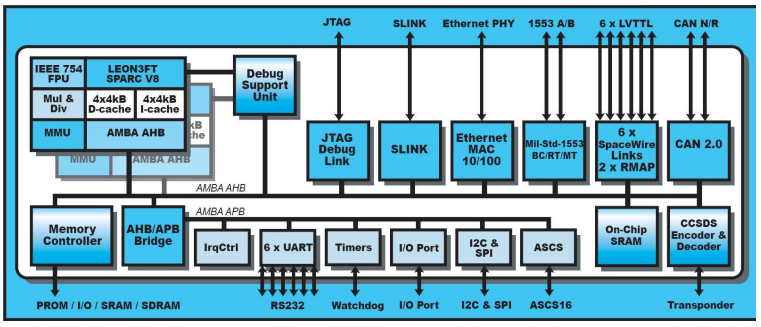

Fig. 9.4 – Léon 3 multi-coeurs

Sur ce système, nous souhaitons adjoindre un composant de détection de la contention mémoire. L'étude précédente nous a permis d'identifier l'intérêt d'un composant électronique. Comme indiqué précédemment celui-ci doit se trouver aux abords du cache de niveau 2. Le schéma suivant 9.5 montre comment notre composant peut être introduit aux abords du cache commun. Notre composant vient se brancher dans le contrôleur mémoire et ce branche en amont et en aval du cache de niveau 2. Ainsi le composant est en mesure de détecter les messages provenant du bus AHB, c'est-à-dire l'ensemble des demandes d'accès en mémoire. Il peut également détecter le nombre de refus de cache et donc d'accès en mémoire réelle. Grâce à la connectique du bus AMBA, il peut de plus, différencier les différents processeurs qui le sollicitent.

Ce composant est, dans un premier temps, un simple système de vérification et de contrôle. Il se limite à observer la vie normale du système et ne fait que compter le nombre réel d'accès au cache et aux différentes mémoires. Ce composant très simple prend peu de place d'un point de vue électronique et organise une gestion de la contention mémoire par le logiciel.

9.5 Modification du processeur

Le Léon3 n'étant pas un processeur intégré(SoC), il est simulé sur FPGA, et plus particulièrement sur notre plate-forme d'évaluation spatiale composée d'un Virtex 5 LX110. Cette plate-forme,

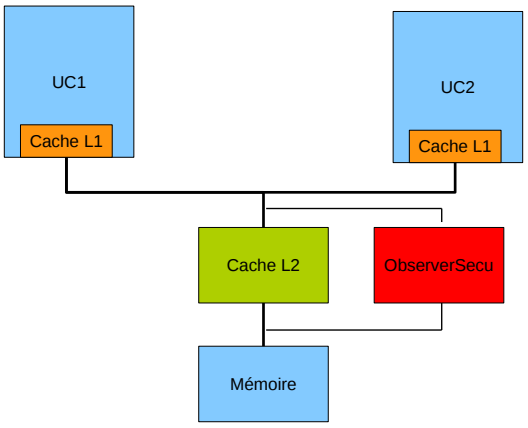

Fig. 9.5 – Branchement de notre composant : observerSecu

peu répandue et commune, est celle fournie par la partie industrielle. Elle répond parfaitement aux contraintes spatiales que nous nous posons. Ainsi, c'est de l'électronique que nous réalisons en intégrant directement dans le contrôleur mémoire notre composant. Celui-ci est composé de 250 lignes de VHDL (dont la plupart servent à s'intégrer dans le processeur) et comprend 37 portes logiques, ainsi que les compteurs permettant de dissocier les appels au cache et à la mémoire ainsi que les appels aux différents types de mémoires disponibles. Nous utilisons 76 emplacements mémoire (soit un espace inférieur à 1%) pour sauvegarder les données. Notre composant est aussi en mesure de différencier les micro-processeurs. Pour comparaison, le processeur Léon3 est composé de 69 120 portes dont 5 416 utilisées pour les périphériques (soit 7%) et 15 346 LUT (soit 22% du processeur), le reste étant exclusivement exploité pour le fonctionnement du processeur lui même.

Notre composant utilise donc moins de 1% de la mémoire et 0.0001% du nombre de portes utilisé par le processeur. Il est donc fortement discret et accessible.

Le composant réalisé se nomme observerSecu, et se décompose en plusieurs blocs électroniques optimisés afin d'en réduire au maximum le nombre de composants. Nous y trouvons un bloc gérant la détection des messages qui concernent les mémoires, avec la possibilité de choisir le type de mémoire que nous souhaitons surveiller. Un autre bloc identifie de connaître le processeur demandeur de la ressource. Un troisième bloc est conçu pour permettre l'accès des processeurs à notre nouveau composant, afin de connaître le nombre d'appel réalisé aux différentes mémoires par ceux-ci.

Pour valider la faisabilité, dans un premier temps nous avons réalisé un composant qui renvoie sur demande le nombre d'accès mémoires. Le processeur demandeur réalise une demande sur le bus AMBA. Cette demande se fait a une adresse différente des mémoires (et correspondant à notre composant) mais permet de connaître le nombre d'accès à la mémoire ainsi que de réinitialiser le compteur de notre composant à zéro. Dans un système hypervisé tel que le nôtre, quelle que soit l'hypothèse d'attaque, il est donc possible que l'hyperviseur vienne contrôler régulièrement le nombre d'accès effectué. Cette première approche est néanmoins encombrante au niveau du bus AMBA (l'utilisation des ressources équivalaient a 25% des données utiles) et ne donne pas les moyens à l'hyperviseur de bloquer un système malveillant facilement.

La remise à zéro du composant ne s'avère pas suffisant. En effet, il faut que le composant fonctionne de façon pleinement autonome pour plus de performance, c'est-à-dire ne pas avoir besoin d'arrêter régulièrement le processeur ou augmenter significativement le temps utilisé par l'hyperviseur. Pour cela, il suffit de fournir à notre composant une valeur de réinitialisation correspondant au nombre d'accès réel à la mémoire que nous autorisons pour chaque processeur. Dans notre cas, cette valeur est fournie uniquement par l'hyperviseur, seul autorisé à communiquer avec notre composant. L'autorisation lui est donnée de manière statique et ne peut être modifiée en cours d'exécution. Le nombre d'accès mémoire autorisé est donc donné par l'hyperviseur en fonction des partitions qu'il fait fonctionner sur chaque processeur.

Il a donc été adjoint un bloc de gestion des interruptions qui permet d'envoyer aux processeurs une interruption dès que la valeur initialisée pour chaque processeur est dépassée. L'information est directement collectée par l'hyperviseur qui peut réagir à ce dépassement.

Nous étudierons dans le chapitre suivant, avec des valeurs sur système réel, les résultats suivants :

— Combien de temps prend un accès normal en mémoire ?
— Combien de temps prend-il lors d'une contention mémoire ?
— Est-il possible de détecter cette contention autrement que par le temps ?

Nous étudierons 4 points en plus de l'observation de la contention mémoire :

— Comment déterminer le nombre d'accès en mémoire que fait une partition durant son fonctionnement normal ?
— Comment décider de la limite que nous lui fixons ?
— Quelle conséquence lors d'un dépassement d'accès ?
— Si le "coupable" n'est pas celui qui dépasse le nombre d'accès mémoire, pouvons nous le détecter ?

L'hyperviseur XtratuM dans sa version adaptée au Léon 3 n'était pas disponible en version multi-coeur. En effet, ce système est dépourvu de gestion séparée de la mémoire et autorise des conflits entre les différents processeurs. Une fois le composant électronique réalisé, nous avons dû modifier légèrement le processeur ainsi que l'hyperviseur XtratuM afin que sa version 3.2 puisse travailler en multi-coeur. Les deux scénarios sont évaluables sur la plate-forme, c'est-à-dire celui n'utilisant qu'un seul hyperviseur gestionnaire commun à l'ensemble des processeurs et celui réalisant un hyperviseur par processeur.

Une fois ce travail effectué, nous avons pu mettre en oeuvre une contention mémoire entre les processeurs. Néanmoins, le bloc électronique de cache de niveau 2 étant propriétaire et n'ayant pas accès à celui-ci, il fut nécessaire de ré-écrire un composant électronique pour réaliser cette tâche. Pour des raisons de simplicité, ce cache ne gère que quatre lignes de 32 octets dans le principe du FIFO (*First In First Out*). Cela veut dire que quatre instructions sont prisent en compte dans le cache. Lorsqu'il est vide, il y a quatre accès mémoires qui remplissent le cache. Si ces mêmes pages sont demandées ensuite, seul le cache répond à la sollicitation. Dans le cas contraire, le cache retire sa ligne la plus ancienne pour la remplacer par la nouvelle page demandée. La taille restreinte de ce cache s'explique par la volonté de démontrer la contention mémoire par la réalisation d'une preuve de concept.

L'ensemble du système est donc prêt à être évalué.

9.6 Conclusion

Ce chapitre nous a permis de présenter la problématique de la contention mémoire. En effet, notre système n'est pas protégé mais des travaux sur des problématiques proches montrent un intérêt à l'utilisation de système électronique plutôt que de code informatique. Nous avons enfin développé nos scénarios d'attaques afin de déterminer les limites de ceux-ci.

Ce chapitre développe également notre proposition ainsi que la plate-forme étudiée, recentre nos questions et met en oeuvre la partie électronique. Celle-ci, fortement optimisée doit répondre à l'ensemble de nos contraintes. Le composant proposé et qui permet de ne pas arrêter le processeur en cours de fonctionnement est mis à l'épreuve dans le chapitre suivant.

Chapitre 10

Implémentation et résultats

Sommaire

Dans ce chapitre, nous observons les résultats obtenus lors de l'évaluation de la contention mémoire. Nous y décrivons l'effet des différentes hypothèses observées au chapitre précédent. Nous finissons en présentant des solutions adaptées à la gestion et la détection des tâches malveillantes. Ce chapitre ouvre sur les perspectives et travaux à venir de cet apport majeur.

10.1 Introduction

La contention mémoire est étudiée sur le Léon3 multi-coeur. Cette plate-forme spatiale est conçue avec le portage d'XtratuM 3.2 en multi-coeur. Ce portage que nous avons réalisé, simplifié, permet la gestion de la mémoire commune en différée. La contention mémoire d'un point de vue plus général est un problème difficile à résoudre. Nous ne cherchons pas à trouver une solution universelle mais une solution propre à notre système. En effet, la complexité de celui-ci est spécifique au domaine des systèmes embarqués temps-réel multi-processeurs avec cache commun. Ce qui s'approche le plus de nore système dans le cas général est la contention générée par deux systèmes souhaitant accéder à une ressource commune avec un impératif de nombre d'accès limité dans le temps. Nous tenterons, par la suite, d'adapter les solutions apportées à un niveau plus général. Nous développons, dans un premier temps, la contention mémoire réalisable sur le Léon3. Puis nous traitons l'influence et l'intérêt de notre composant sur la plate-forme. Enfin nous exposons les scénarios envisageables avec notre composant.

10.2 Quantification de la contention

La mémoire du Léon3, commune à tout les processeurs est accessible via le bus AMBA, tel que développé au chapitre précédent. Accessibles via un contrôleur mémoire, ses sous-parties sont réunies dans une table d'adresses correspondant à chaque type de mémoire disponible. Un récapitulatif des mémoires disponibles et de leurs adresses respectives (gérées par le contrôleur mémoire avec des adresses héxadécimales comprises entre 0x00000000 et 0xFFFFFFFF) est décomposé tel que :

— 0x00000000 - 0x1FFFFFFF correspond à la taille de la PROM ;
— 0x20000000 - 0x3FFFFFFF correspond à la taille des I/O ;
— 0x40000000 - 0x7FFFFFFF correspond à la taille des SRAM/SDRAM ;
— 0x80000000 - 0x80FFFFFF correspond à la taille du bridge 1 de l'APB ;
— 0x90000000 - 0x9FFFFFFF correspond à la taille du DSU3 ;
— 0xFFFB0000 - 0xFFFB0FFF correspond à la taille de l'ETH OC ;
— 0xFFFC0000 - 0xFFFC0FFF correspond à la taille du CAN MC ;
— 0xFFFFF000 - 0xFFFFFFFF correspond à la taille du AHB plug and play.

La mémoire PROM est une mémoire ré-inscriptible, qui conserve les données en cas de coupure électrique et contient les informations relatives au fonctionnement des processeurs Léon3.

Les I/O (pour *Input/output*, respectivement entrée/sortie) correspondent à des mémoires externes.

La SRAM/SDRAM est la mémoire vive du système. Elle exécute des parties de codes dynamiquement et est utilisée pour sa rapidité de lecture et d'écriture. Nous la détaillerons plus précisément par la suite.

Le *bridge 1* de l'APB correspond à une petite mémoire permettant de faire des communications entre le bus de données lent (APB du bus AMBA) avec un bus plus rapide AHB (celui qui rend possible la communication inter-processeur et d'accès à la mémoire). Ce "pont" permet une communication, plus lente, avec des périphériques extérieurs tels que des mémoires supplémentaires que nous ajouterions pour un besoin spécifique.

Le DSU (*Data Service Unit* ou unité de service des données) est un périphérique d'interface pour relier un réseau local (LAN) à un réseau étendu (WAN).

L'ETH OC est une zone mémoire réservée aux fonctions de la partie éthernet pour, potentiellement, brancher le système sur un réseau de type internet.

Le CAN MC, est un espace mémoire utilisé pour le bus de communication CAN, particulièrement utilisé dans l'automobile. Celui-ci réduit fortement le nombre de câbles nécessaires dans un véhicule. Cela montre la volonté de modularité de la plate-forme. Ce bus est typiquement spécifique aux systèmes embarqués sol.

L'AHB plug and play (pour les branchements en cours de fonctionnement), alloue une mémoire permettant de reconnaître et brancher certains systèmes sur le bus rapide AHB AMBA. Le code contenu est réalisé par l'utilisateur. L'espace est alloué mais non implémenté dans notre cas.

La mémoire sélectionnée que nous allons évaluer n'aura pas d'influence particulière sur le résultat concernant le nombre d'accès mémoire. En revanche, celle-ci aura une conséquence temporelle.

En effet, chaque mémoire a un temps de réponse différent. L'effet de la contention mémoire est observable par l'augmentation de ce temps pour un même nombre d'accès mémoire, mais celui-ci est propre à chaque système. Nous cherchons à trouver une solution unique pour l'ensemble des mémoires disponibles sur le marché. Ceci explique notre choix de fixer un temps donné et d'observer le nombre d'accès mémoire sur ce laps de temps. Celui-ci est donc complètement indépendant du type de mémoire observée.

Notre choix de mémoire se porte sur la RAM(*Random Access Memory*) aussi appelée mémoire vive. Cette mémoire, définie comme SRAM/SDRAM, a la particularité d'être bien plus rapide que les autres mais ne permet pas la conservation des données en cas de coupure électrique. La mémoire RAM est configurée selon la façon dont le programme remplit celle-ci, en mettant en premier le code utilisé par les processeurs pour fonctionner. Les programmes supplémentaires suivants tels que l'hyperviseur ou un simple code bare-métal se trouvent directement à la suite dans la mémoire. Ainsi la répartition des codes à exécuter se fait toujours en début de mémoire (mémoire dont les adresses sont de valeur basse). Les adresses hautes sont donc vides de toute donnée utile.

10.2.1 Mémoire RAM

Nous étudions le fonctionnement normal de la mémoire RAM sans contention. Dans un premier temps, notre évaluation se porte sur un principe simple. Nous envoyons un maximum d'accès à la mémoire RAM pour pouvoir constater la contention mémoire sur cette dernière dans la section suivante. Dans un premier temps, seul un processeur est activé et envoie des requêtes d'informations à la RAM. Le cache de niveau 2 est désactivé afin de ne pas parasiter la mesure. Le cache de niveau 1, en revanche, reste actif.

L'effet du cache de niveau 1 sur notre mesure est non-négligeable. Nous avons un cache d'instructions L1 qui fait 256 lignes de 32 octets et un cache de données L1 qui fait 128 lignes de 32 octets. Cette décomposition du cache L1 en deux parties distinctes, instructions et données, est imposée par les concepteurs du Léon3. Elle permet de meilleures performances avec une plus grande rapidité liée aux accès. Dès qu'une instruction ou une donnée nouvelle est demandée à la mémoire, le cache enregistre la donnée envoyée par la mémoire au principe de la FIFO (déjà détaillée dans le chapitre 9). Ainsi l'appel régulier à la même mémoire passe automatiquement par le cache de niveau 1, qui, contenant la réponse désirée répond toujours en lieu et place de la mémoire. Dans ce cas, la mémoire ainsi que le bus AMBA ne sont pas sollicités. Afin de passer outre ce cache, utile en fonctionnement optimisé, mais gênant pour notre mesure, nous obligeons le système à aller lire la mémoire RAM en des points (adresses) fortement éloignés les uns des autres. Ce principe simple à mettre en place génère un défaut de cache. Celui-ci ne peut gérer les cas imprédictifs, et c'est ce que nous faisons en l'envoyant lire des parties de mémoires fortement éloignées et dé-corrélées les unes des autres.

Nous exprimerons, par la suite, les plages d'adresses que nous observons. Lors de la mesure de contention ou de fonctionnement normal, celles-ci prennent en compte un bout de code pseudo-aléatoire permettant de mettre en défaut le cache de niveau 1, et ainsi de vérifier le nombre d'accès en mémoire réel maximal réalisable dans un temps donné.

Première évaluation

Notre première évaluation mono-processeur se fait par une succession d'appels (en boucle) représentant une lecture dans la RAM. Il y a donc un seul processeur allumé, un cache de niveau 2

Expérience	Nombre moyen accès RAM	Intervalle de mesure (micro-secondes)	Temps d'accès moyen en RAM (nano-secondes)
CPU1 seul	4837	521	107

Table 10.1 – Tableau référençant les valeurs maximales d'accès en RAM d'un mono-processeur Léon3

Expérience	Nombre moyen accès RAM	Intervalle de mesure (micro-secondes)	Temps d'accès moyen en RAM (nano-secondes)
CPU1 + CPU2	4837	521	107
CPU1	1919	521	271

Table 10.2 – Tableau référençant les valeurs maximales d'accès en RAM d'un multi-processeur Léon3

désactivé et 1 cache de niveau 1 réalisant uniquement des *cache miss*. Cette plate-forme minimaliste permet un étalonnage précis du nombre et du temps d'accès mémoire. Nous déterminons le temps avec la fonction de gestion disponible sur le processeur Léon3 précédemment utilisée dans la phase d'observation des attaques sur hyperviseur lors de la présentation de cette thèse (chapitre 3). Nous déterminons le nombre d'accès mémoire effectué par comparaison avec le nombre de tours de boucle réalisé par la fonction de lecture en RAM que nous avons écrit. Les valeurs ainsi obtenues par un processeur non perturbé par un acte malveillant, seront prises comme références par la suite.

Le tableau 10.1 représente le nombre d'accès mémoire dans une période de temps équivalente à 2 micro-secondes. Les mesures sont réalisées plus de 20 fois afin d'obtenir des valeurs moyennes représentatives. Nous observons dans ce tableau les valeurs correspondantes à un seul processeur allumé et lisant continuellement dans la mémoire RAM à des adresses différentes (pour éviter les effets des caches L1 et L2). La première colonne du tableau représente le nom de l'expérience. La deuxième est le nombre d'accès en RAM réalisé dans un laps de temps indiqué en micro-secondes dans la colonne suivante. La dernière colonne est le temps mis en nano-secondes par une demande d'accès mémoire par un processeur. Les valeurs indiquées sont des valeurs statiques en réalisant la moyenne des résultats obtenus lors des différents tests.

Un accès mémoire correspond à 107 ns, ce qui équivaut à 9 accès mémoire par micro-secondes. Cette valeur correspond au nombre d'accès maximal réalisable par un mono-processeur. Ce premier test a permis de vérifier la valeur d'accès à la mémoire RAM directement. Afin de calibrer le système, nous étudions maintenant deux autres références.

Nous mettons en oeuvre le deuxième processeur qui imite le premier et fait également des appels à la mémoire RAM. Par le même principe, celui-ci met en défaut son cache de niveau 1 et le cache de niveau 2 reste désactivé. Pour ne pas créer de collision lors de la lecture, il regarde des parties de la mémoire RAM différente du premier processeur. Nous étudions également le nombre d'accès réalisé sur le processeur 1 seul. Les valeurs sont visibles dans le tableau 10.2.

Le tableau 10.2 conserve une présentation identique au tableau précédent. Néanmoins, nous

analysons deux données : Les valeurs obtenues par le processeur 1 et le processeur 2 ensemble, sur la première ligne du tableau ; et les valeurs obtenues par le processeur 1 dissocié du processeur 2 (également en fonctionnement). Cette dissociation du processeur 1 permet de déterminer le nombre d'accès de chaque processeur indépendamment de la valeur globale obtenue dans la première ligne du tableau.

Le tableau montre qu'en multi-processeur, la valeur du nombre d'accès mémoire est équivalente à la valeur obtenue dans le tableau 10.1 soit 107 nano-secondes par accès en RAM. Cependant, ce n'est pas un mais deux processeurs qui communiquent avec la mémoire RAM (C'est la valeur globale obtenue par les 2 processeurs). Le premier des processeurs voit son taux d'accès diminué de plus de la moitié. Cela s'explique par deux faits : Le bus AMBA divise équitablement son taux d'accès ce qui a pour conséquence de réserver 50% du temps au premier processeur. Ainsi le même taux est à la disposition du deuxième, ce qui divise équitablement le temps par 2 ; De plus, notre code fait régulièrement des appels à la fonction de temps sur le processeur 1. Le nombre d'accès à la mémoire RAM par le processeur 1 se trouve alors encore diminué. Nous trouvons donc respectivement 3 accès mémoires par micro-seconde pour le processeur 1 et 6 pour le processeur 2 (n'ayant aucun appel à la fonction de temps durant son exécution). D'autres tests ont été effectués et confirment que chaque processeur peut en moyenne faire 4.5 accès mémoire par micro-seconde, et au maximum 6 accès par micro-seconde. Le temps correspondant au temps d'accès moyen en RAM (271ns) prend en compte le temps d'attente de l'accès au bus AMBA, ce qui explique sa valeur plus élevée.

Nous avons quantifié le nombre d'accès maximal réalisable dans un temps donné et exprimé celui-ci en accès par micro-seconde. Il a été possible de répondre à la question, exprimée en fin de chapitre 9, *combien de temps prend un accès normal en mémoire ?*. Nous avons également observé que la valeur correspondant au nombre d'accès mémoires dépend, sur notre système, du nombre de processeurs parallélisés.

Mise en place du cache

Nous mettons maintenant en oeuvre le cache de niveau 2 afin de déterminer le temps d'exécution pris par le système pour y accéder. Pour cela nous réalisons deux programmes distribués sur les deux processeurs de la plate-forme multi-processeur. Ils accèdent une première fois à deux adresses précises afin qu'elles soient intégrées dans le cache de niveau 2. Puis chacun des processeurs appelle une de ces deux adresses afin d'accéder, sans passage par la mémoire, au cache de niveau 2. Les caches de niveau 1 sont désactivés pour l'évaluation car la taille du cache de niveau 2 (4 lignes) est trop faible pour mettre en défaut le cache de niveau 1. Normalement, les caches de niveau 2 sont plus imposants que ceux de niveau 1 et auraient permis une mise en défaut du cache de niveau 1 sans altérer les capacités du niveau 2. Ce ne peut être le cas dans notre configuration. Les valeurs obtenues sont visibles dans le tableau 10.3

Les résultats montrent qu'il est possible de réaliser en tout 12 accès en cache de niveau 2 par micro-seconde, ainsi qu'en moyenne 6 accès par processeur. La valeur visible sur le processeur 1 est obtenue grâce à l'analyse du temps maintenant réalisé par le processeur 2. Nous observons une répartition égalitaire du temps d'accès au système.

Pour conclure sur ces premières observations, nous constatons qu'il est possible de réaliser 6 accès à la mémoire par micro-seconde au maximum pour chaque processeur et que chacun d'eux

Expérience	Nombre moyen accès RAM	Intervalle de mesure (micro-secondes)	Temps d'accès moyen en RAM (nano-secondes)
CPU1 + CPU2	21942	1694	77
CPU1	10971	1694	154

Table 10.3 – Tableau référençant les valeurs maximales d'accès au cache L2 d'un multi-processeur Léon3

Expérience	Nombre moyen accès RAM	Intervalle de mesure (micro-secondes)	Temps d'accès moyen en RAM (nano-secondes)
Exp1 : CPU1 + CPU2	2194	1694	772
Exp1 : CPU1 accède au cache	2194	1694	772
Exp2 : CPU1 + CPU2	9435	1694	179
Exp2 : CPU2 perturbe	7863	1694	215
Exp2 : CPU1 accède en RAM	1572	1694	1077

Table 10.4 – Tableau référençant les valeurs de contention mémoire du cache L2 d'un multi-processeur Léon3 via deux expériences

a une durée équivalente à 271ns. De plus, il est possible d'accéder 12 fois au cache L2 par micro-seconde avec un accès correspondant à 154ns par processeur. L'intérêt du cache de niveau 2 n'est plus à démontrer. Il est possible maintenant de mettre en place des scénarios afin de quantifier la contention mémoire.

Effets de la contention

Nous développons des codes sur chacun des deux processeurs à notre disposition. Le premier doit fonctionner normalement en ne réalisant qu'un seul type d'accès, une lecture à une adresse fixe en mémoire. La première lecture se fait en inscrivant cette ligne dans le cache L2. Le deuxième processeur, lui, fait des accès en lecture dans l'ensemble de la plage d'adresse de la mémoire RAM afin de réaliser un maximum d'accès non référencés dans le cache L2. Les caches L1 sont désactivés.

Nous cherchons à montrer qu'il existe un impact direct sur le fonctionnement du processeur 1. Celui-ci ne pouvant optimiser ses accès en mémoire se voit sanctionné par une perte de temps et une longue attente. Les valeurs obtenues sont visibles au tableau 10.4.

Ce tableau 10.4 se décompose en deux parties. La première expérience appelée Exp1, ne fait fonctionner que le processeur 1 sur le système multi-processeur. Le processeur 2 ne fait aucun accès à la mémoire, ni au cache de niveau 2. Nous voyons que le programme observé sur le processeur 1 ne fait que des accès en cache correspondant à un accès mémoire par micro-seconde. En effet, l'initialisation par première lecture dans la mémoire se fait avant la prise de mesure. Le CPU1

communique volontairement à la mémoire avec une fréquence diminuée du sixième par rapport à son maximum autorisé. Le ralentissement de la fréquence est voulu, pour permettre au cache de niveau 2 d'être corrompu facilement par le CPU2. Ainsi, les valeurs correspondant aux deux CPU pris simultanément sont identiques à celle du CPU1, seul utilisateur du bus AMBA en direction de la mémoire.

L'EXP2 représente, quant à elle, la contention mémoire à proprement parler. Le programme intégré dans le CPU2 se met à communiquer à la fréquence maximale afin de demander systématiquement de nouvelles zones mémoires ne se trouvant pas dans le cache de niveau 2. Ainsi, nous retrouvons un nombre d'accès mémoire correspondant au maximum de demandes possibles. Il y a donc en moyenne quatre accès mémoires par micro-seconde pour le CPU2. Le CPU1 a parfaitement le temps de s'exécuter sur le système et dispose de suffisamment d'accès au bus AMBA pour répondre dans le même temps que l'expérience 1. Mais le CPU1 subit les ré-actualisations continues du cache par le processeur 2, le CPU1 devant faire maintenant systématiquement des accès en mémoire pour récupérer à chaque fois la valeur de son adresse désirée. Ainsi nous passons d'une valeur supérieure à un accès par micro-seconde environ, à une valeur inférieure à un par micro-seconde. La contention est avérée et impacte directement et fortement le temps d'exécution de la tâche réalisée sur le CPU1, ce qui répond à la question présentée en fin de chapitre 9, *Combien de temps prend le système pour accéder à la mémoire lors d'une contention ?*. Dans ces deux expériences, le CPU1 prend en compte le temps perdu par les fonctions d'observation de notre programme.

Ainsi nos deux expériences démontrent qu'un programme utilisant une faible partie du cache de niveau 2 peut, sans gestion correcte, subir une contention mémoire et se voir obliger de réaliser systématiquement des accès à la mémoire. Cela a un impact certain sur le système. En effet, pour que certains systèmes complexes fonctionnent comme des partitions virtualisées, il faut parfois prédire qu'au moins un certain nombre d'accès se fera en cache et non en mémoire. Pour palier à ce problème, nous proposons la mise en oeuvre de notre composant électronique.

10.3 Mise en place du composant de sécurité

Le composant électronique est simple et robuste. Il permet de comptabiliser, au fur et à mesure des accès, le nombre réel d'appel à la mémoire et au cache de niveau 2. Il se trouve dans le contrôleur mémoire et observe le bus AMBA ainsi que le bus mémoire dédié. Il notifie chaque accès à une mémoire via un compteur précis. Pour notre expérience, le compteur n'envoie de réponse qu'en cas de demande spécifique d'un des processeurs et ne retourne l'accès qu'aux adresses de valeur haute de la RAM, afin d'éviter le bruit du fonctionnement normal du système. Nous appliquons la contention mémoire via les deux expériences citées au-dessus mais nous prenons en compte les valeurs également prises par notre composant. Nous y observons les mêmes valeurs que le tableau 10.4. Le nombre d'accès moyen au cache de niveau 2 est identique au nombre fourni par notre composant. Notre composant renvoie le nombre d'accès à la RAM qu'a effectué le processeur demandeur à une adresse précise du contrôleur mémoire. Ainsi les valeurs retournées par le composant aux processeurs sont visibles dans le tableau 10.5. Les paramètres des deux expériences sont inchangés.

La valeur 0 obtenue pour l'expérience 1 n'est pas une erreur. En effet, le composant, bien que prenant en compte le nombre d'accès au cache de niveau 2, ne retourne que les valeurs d'accès en

Expérience	Valeur du composant	Intervalle de mesure (micro-secondes)
Exp1 : CPU1 + CPU2	0	1694
Exp1 : CPU1 accède au cache	0	1694
Exp2 : CPU1 + CPU2	9435	1694
Exp2 : CPU2 perturbe	7863	1694
Exp2 : CPU1 accède en RAM	1572	1694

Table 10.5 – Tableau référençant le nombre moyen d'accès RAM fourni par le composant pour les Exp1 et 2

RAM. Cela confirme parfaitement que le CPU1, dans l'EXP1, ne fait que des accès au cache, et que l'EXP2 ne fait que des accès en mémoire.

Notre composant permet donc le contrôle dynamique du nombre d'accès mémoire, et ceci au cycle près, et sans perturbation du fonctionnement général des processeurs. La dernière question que nous posions en matière de faisabilité générale, dans le chapitre 9, était *Est-il possible de détecter cette contention autrement que par le temps ?*. C'est le cas de notre composant qui ne maîtrise pas le temps et ne l'observe pas. Celui-ci ne fait que regarder les accès en cache et en mémoire. Nous appliquons maintenant les deux scénarios que nous avons fixés afin de mettre à l'épreuve notre solution.

10.4 Proposition à l'hypothèse 1

L'hypothèse numéro 1 est la prise en compte d'un système permettant à un hyperviseur situé sur un unique processeur de gérer les travaux de l'ensemble des autres processeurs. Cet hyperviseur centralisé envoie des tâches à exécuter (partitions) sur les autres processeurs dans un temps donné. Bien que le temps soit contrôlable, il existe le problème de contention mémoire. Dans cette hypothèse, nous incluons le composant que nous avons réalisé. Celui-ci permet uniquement de compter le nombre d'accès mémoire propre à chaque processeur et de le remettre à zéro si besoin. L'hyperviseur doit donc demander régulièrement le nombre d'accès au composant.

Cette hypothèse de composant nous pose une difficulté. Il encombre le bus AMBA de demande de l'hyperviseur et ne permet pas une gestion efficace du temps, car le processeur dédié à l'hyperviseur devrait arrêter celui-ci à chaque demande. Le composant est donc modifié afin que le processeur dédié à l'hyperviseur puisse lui envoyer une valeur pour chacun des processeurs qu'il surveille. Cette valeur correspond au nombre maximal d'accès à la mémoire que l'hyperviseur lui autorise dans un temps donné. En cas de dépassement de la valeur, une interruption est directement envoyée a l'hyperviseur. Cela permet à l'hyperviseur de libérer fortement le bus AMBA en réduisant les appels faits à notre composant. Néanmoins, il faut déterminer le nombre d'accès "normal" de

chaque partition dans un temps donné. Pour cela, il y a plusieurs possibilités que nous discuterons plus en détails dans les travaux à venir. Ces valeurs sont déterminées par expérimentation lors de cette preuve de concept.

L'objectif de notre hypothèse est de vérifier qu'il est possible pour l'hyperviseur, et donc le système, de gérer efficacement la contention mémoire.

La mémoire est partagée statiquement entre les différents processeurs et entre les différentes partitions s'exécutant sur le système. Ainsi, chaque bloc mémoire appartient à une partition unique d'un processeur unique. Une autre partition ne peut accéder à cette partie de la mémoire. Cela n'empêche pas la contention au niveau du cache de niveau 2. Néanmoins, grâce à notre composant, il est possible d'identifier le demandeur d'information. Par conséquent, nous pouvons contrôler la validité de la demande et garantir le nombre maximal d'exécution dans un temps donné, en fonction de la zone mémoire visée.

La difficulté pour l'hyperviseur est la phase de transmission des partitions sur les différents processeurs. Cette répartition est un ordonnancement statique fixé par l'utilisateur. Néanmoins, au niveau du cache de niveau 2, cela change les résultats. En effet, il y a deux possibilités. Soit l'hyperviseur prend en compte le coût de transfert, c'est-à-dire le nombre d'accès à la mémoire lorsqu'il initialise notre composant à une valeur, soit il attend que la tâche commence à s'exécuter sur le processeur cible pour initialiser notre composant.

Le coût du transfert d'une tâche sur un processeur est très complexe. En effet, il est difficile de savoir quel est l'état du cache commun au moment du transfert et quel impact il aura sur le système. Si le coût de transfert n'est pas pris en compte, il est possible de générer de la contention mémoire telle que montrée dans le tableau 10.4. En effet, une nouvelle partition qui doit s'installer sur son processeur doit pouvoir s'exécuter et faire pour cela de très nombreux accès en mémoire. Les autres processeurs se verront supprimer leurs lignes de caches commun (niveau 2) et subiront une contention.

Une hypothèse est de décomposer le cache statiquement, de la même manière que la mémoire. Le cache de niveau 2, est alors dépendant de chaque partition mais perd son utilité commune. En revanche, si nous considérons que chaque processeur a accès à une partie seulement en écriture, mais peut lire dans l'ensemble, alors nous conservons l'intérêt du cache en réduisant l'impact de la contention. C'est-à-dire qu'un processeur, lorsqu'il accède en mémoire, modifie le cache. La politique de gestion du cache serait de remplacer une ligne dédiée à ce processeur par une autre. Les blocs dédiés aux autres processeurs ne subissant aucune modification.

Cette solution n'est pas sans impact sur le fonctionnement du système. Les performances peuvent en être impactées. En effet, nous partions implicitement du principe que chaque partition avait le même nombre d'accès au cache nécessaire pour son fonctionnement ; Pourtant certaines partitions nécessitent plus d'accès que d'autre et ce, indépendamment du processeur. La contention en serait pourtant fortement diminuée.

Afin d'appuyer nos propos, nous avons réalisé un test permettant d'allouer 2 lignes de cache de niveau 2 à un processeur et 2 au deuxième. Nous y étudions maintenant le temps de transfert d'une tâche vers un processeur et son impact en fonction de la taille du cache autorisé. La fonction à transférer fait 4 appels à la mémoire. Elle s'exécute ensuite en appelant systématiquement les deux premières lignes de caches nécessaires au transfert. Le résultat est visible dans le tableau 10.6. Le deuxième processeur, également hyperviseur, fait accès systématiquement à la même adresse

Expérience	Valeur du composant	Intervalle de mesure (micro-secondes)
Début : CPU1 + CPU2	0	1694
CPU1 accède au cache	0	1694
CPU2 ne fait rien	0	1694
Transfert : CPU1 + CPU2	7	1694
CPU1 recharge sa valeur	1	1694
CPU2 est transféré et s'exécute	6	1694

Table 10.6 – Tableau référençant le nombre moyen d'accès RAM fourni par le composant pour un transfert de partition

mémoire (déjà présente en cache).

Le tableau 10.6 montre qu'en fonctionnement normal, le CPU1 accéde au cache de niveau 2, celui-ci n'est pas perturbé et notre composant retourne une valeur nulle. En revanche, lors du transfert du code vers le processeur 2, nous observons 3 accès au cache sans conséquence, car la première ligne de cache est détenue par le processeur 1 qui continue ses transferts. Puis le processeur 2 annule la dernière ligne de cache (principe du FIFO). L'annulation de cette dernière ligne de cache oblige le CPU1 à recharger sa donnée sur la première ligne de cache utilisée pour le transfert du CPU2. Cette ligne étant utilisée par le CPU2 lors du fonctionnement, il invalide la suivante, pourtant également indispensable. Celle-ci est donc ré-écrite à la ligne de cache suivante au tour d'après. Il y a donc 6 opérations d'accès mémoires pour le transfert sur le CPU2 dont 4 pour le passage du CPU1 au CPU2. Les 2 dernières correspondent à la mise en route du programme sur le CPU2.

Cette illustration montre la limite de notre composant. En effet, il ne peut savoir quelles opérations sont imputables à un transfert. Ceci est contrôlé par l'hyperviseur. Il est néanmoins impératif de contrôler le transfert car il est générateur de contention mémoire.

Une autre mesure prend cette fois en compte un cache divisé en deux blocs. Chacun de ces caches est affecté à un processeur différent avec une taille de 2 lignes. Nous exécutons le même programme que précédemment dans le tableau 10.7.

Nous constatons que la séparation du cache modifie le fonctionnement général du système. Celui-ci, identique à l'étape précédente, n'accède qu'au cache de niveau 2 au début du programme. Lorsque la nouvelle tâche est allouée au CPU2, le système écrit dans le cache, ligne par ligne l'ensemble de ces 4 instructions. Les deux dernières supprimant les 2 premières. Puis, la mise en route

Expérience	Valeur du composant	Intervalle de mesure (micro-secondes)
Début : CPU1 + CPU2	0	1694
CPU1 accède au cache	0	1694
CPU2 ne fait rien	0	1694
Transfert : CPU1 + CPU2	6	1694
CPU1 recharge sa valeur	0	1694
CPU2 est transféré et s'exécute	6	1694

Table 10.7 – Tableau référençant le nombre moyen d'accès RAM fourni par le composant pour un transfert de partition avec cache divisé par 2

de la tâche fait se recharger les deux premières lignes en cache. Ainsi, jamais le processeur 1 n'a eu à recharger les données utiles. Il ne subit pas de contention mémoire et s'exécute correctement. En revanche le coût de transfert et d'exécution de la tâche sur le CPU2 reste identique en nombre d'accès. Il est évident que ce cas particulier est fortement bénéfique pour nous. Il met en avant l'avantage certain d'un cache partitionné. Néanmoins, si le programme à exécuter était plus complexe, il y aurait de nombreux échecs de caches qui auraient pu être palliés en ayant un cache plus grand. Cependant cette hypothèse de partitionnement reste efficace même lors de l'exécution du système et limite l'impact d'un processeur sur un autre.

Il faut pourtant conserver la propriété suivante : *Les processeurs peuvent lire dans l'ensemble du cache*. En effet, notre système, muni d'une MMU, contrôle les autorisations d'accès à certains emplacements mémoires. Le fait de lire des pages de cache concernant un autre programme peut ne pas être gênant dans l'éventualité où le logiciel y est autorisé. Cela évite de recharger sa propre partie de cache avec une donnée identique à celle déjà pré-chargée.

Conclusion

Notre hypothèse de fonctionnement démontre qu'un hyperviseur central peut générer une contention mémoire en distribuant les tâches sur différents processeurs. Néanmoins, limiter le nombre de lignes de caches accessibles pour chaque processeur permet d'en limiter l'effet. Notre composant, associé à une gestion efficace de l'hyperviseur, s'est avéré cohérent et répondant aux attentes.

10.5 Proposition à l'hypothèse 2

L'hypothèse numéro 2 prend en compte un partage de l'hyperviseur sur plusieurs processeurs simultanément. Contrairement à la vision précédente, l'hyperviseur est partagé. Il interrompt donc régulièrement l'ensemble des processeurs. Ainsi, l'hyperviseur n'est pas toujours activé sur au moins un des processeurs et peut donc laisser plus de temps aux différents programmes pour s'exécuter. Lorsqu'un processeur a fini une tâche, il peut lancer l'hyperviseur qui lui ordonnancera la tâche suivante à réaliser et qui répondra aux demandes des autres processeurs.

Dans cette hypothèse, nous cherchons à savoir si il est possible de détecter une contention mémoire et si nous pouvons la gérer. La difficulté supplémentaire réside dans une volonté manifeste d'un autre programme de vouloir influencer le résultat pour mettre en défaut la tâche normale de l'hyperviseur.

Nous partons du principe qu'il existe deux programmes. L'un est dit normal et doit faire un maximum de 4 exécutions en mémoire durant son fonctionnement. Mais celui-ci demande 8 accès en direction de la mémoire, ce qui implique qu'il doit au moins avoir 50% des pages recherchées en cache. Les 8 accès mémoires se font sur deux adresses. Il y en a respectivement 4 par adresse.

Le programme malveillant, est quant à lui, autorisé à faire 20 accès en mémoires au maximum. Celui-ci a pour seul objectif de faire condamner une autre tâche par l'hyperviseur à sa place. Lorsqu'il n'attaque pas, le programme ne fait que 2 accès en mémoires et va lire les adresses en questions.

Il existe toujours un cache de 4 lignes commun à 2 processeurs. La tâche dite normale se trouve sur le CPU2 et la tâche malveillante sur le CPU1. Les résultats sont visibles dans le tableau 10.8.

Nous observons que lorsqu'il n'y a pas d'attaque, le programme fonctionne correctement. En particulier, l'ensemble des tâches, ordonnancées efficacement, est très en deçà des limites attendues. Chacun d'eux ayant atteint un nombre d'accès en RAM inférieur d'au moins 50% par rapport aux limites.

Dans le cas de l'attaque, le programme malveillant tournant sur le CPU1, réalise autant d'accès qu'il lui est permis. Il n'est donc pas considéré comme fautif, ni par l'hyperviseur, ni par notre composant. En revanche le programme normal se retrouve en défaut. En effet, celui-ci subit le remplacement des lignes de caches successifs par le CPU1 et se retrouve dans l'obligation de recharger à chaque fois son code. Il dépasse la valeur maximale autorisée et se fait condamner par l'hyperviseur car il est détecté comme responsable de la contention mémoire.

Notre apport

Nous proposons, pour déceler ce type de programme malveillant, de vérifier les accès mémoire des CPU avec la gestion d'un hyperviseur commun. Celui-ci peut vérifier le nombre d'accès au cache de niveau 2 et en déduire si un tel type d'attaque est réalisé.

La stratégie consiste à déterminer le nombre d'accès "normal" d'une fonction lors d'une exécution du plan d'ordonnancement. Cette valeur est sauvée par l'hyperviseur, qui, en cas de dépassement de l'une des partitions, vérifie dans son tableau si la partition précédente à fait plus d'accès qu'à l'accoutumé. Si c'est le cas, l'hyperviseur peut intervenir sur la tâche réellement malveillante. La politique de gestion est développée plus en détail dans l'ouverture de la thèse.

Expérience	Valeur du composant	Intervalle de mesure (micro-secondes)
Sans attaque : CPU1 + CPU2	4	1694
CPU1 accède au cache	2	1694
CPU2 ne fait rien	2	1694
Avec attaque : CPU1 + CPU2	6	1694
CPU1 recharge sa valeur	20	1694
CPU2 est transféré et s'exécute	8	1694

Table 10.8 – Tableau référençant le nombre moyen d'accès RAM fourni par le composant pour une attaque dissimulée

Il est donc possible de détecter, grâce à l'hyperviseur et à notre composant, une tâche qui tenterait de mettre le système en défaut tout en se dissimulant.

10.6 Conclusion

Notre composant a permis de relever deux difficultés et de déterminer les cas de contention mémoire. Cela a aussi permis à l'hyperviseur de différencier les partitions, ainsi que les consommations "normales" et incohérentes de ces dernières. Il reste à déterminer deux points. Quelle politique de gestion est utilisée par l'hyperviseur lors de l'observation d'une contention mémoire ? Comment déterminer de manière automatique le nombre d'accès mémoire d'un programme ?

Chapitre 11

Perspectives et travaux à venir

Sommaire

11.1 Introduction

Notre apport a permis de démontrer la faisabilité et l'efficacité d'un composant dénombrant les accès au cache de niveau 2 et à la mémoire. Cette différenciation permet, pour les systèmes complexes tels que dans l'aérospatial, d'observer l'ordonnançabilité par exemple. Dans ce chapitre nous concluons le deuxième apport majeur de notre thèse et amenons à la conclusion finale de celle-ci. Nous développons, dans ce chapitre, deux perspectives et travaux à venir. La première est dans la continuité directe des résultats précédemment obtenus et a pour objectif de proposer des politiques de gestions de la contention mémoire. La deuxième perspective est de faire un lien avec le premier apport et exprimer la faisabilité de la mesure automatique du nombre d'accès d'un programme.

11.2 Politiques de gestions de la contention

Dans cette section, nous déterminons les différentes politiques de gestions qu'il est possible de mettre en place grâce à notre apport. Notre composant électronique remonte l'information du nombre d'accès à la mémoire mais laisse totalement libre l'exploitation de cette information. Pour des raisons de sécurité, seul l'hyperviseur a accès aux informations de notre composant et peut modifier ces valeurs. L'hyperviseur est donc en charge de déterminer les valeurs normales ou non de chaque partition qu'il contrôle.

Hypothèse 1

Prenons la première hypothèse. Lorsque l'hyperviseur fonctionne sur un processeur unique, la politique de gestion est plus simple à comprendre. Pour que l'hyperviseur puisse contrôler le dépassement d'une partition, il faut déterminer au préalable le nombre d'accès mémoire qu'elle réalisera par la suite. Considérons dans un premier temps que cette valeur est connue. Avant le démarrage de chaque partition sur un processeur, l'hyperviseur doit accéder à notre composant afin de le réinitialiser, avec la valeur correspondant au nombre d'accès maximal.

Cette étape effectuée, il est possible de lancer la partition sur le processeur visé. Lorsque le composant détecte un dépassement de la partition sur son processeur, une interruption intervient dans l'hyperviseur qui doit gérer le problème détecté. Dans notre hypothèse, l'hyperviseur est toujours en cours de fonctionnement et peut immédiatement gérer le problème. La politique la plus simple est d'observer celui-ci et de le noter dans un journal d'événement.

La contention mémoire est ainsi observée mais non traitée. Cela permet de faciliter l'intégration mais empêche toute résolution dynamique du problème.

Une autre solution, radicale, est la suppression de la partition réalisant un dépassement pour garantir l'ordonnancement du système. Cette solution permet une action dynamique sur le système et régule le nombre d'accès en mémoire. Il permet également d'associer cette gestion dynamique avec la technique de fichier d'évènement.

La solution n'est pas optimale. En effet, outre le fait qu'elle ne gère pas et ne détecte pas un programme malveillant dissimulé comme nous le verrons dans l'hypothèse 2, cette solution peut mettre en défaut les fonctions premières du système. Dans un satellite, certaines partitions restent inactives un long laps de temps et se réveillent lors d'un évènement exceptionnel. Ce réveil fait se réaliser un plus grand nombre d'accès mémoire. Si celui-ci a été mal déterminé ou calibré en fonction des autres programmes s'exécutant sur le système alors l'hyperviseur bloque la partition impactée et l'empêche d'observer le phénomène exceptionnel. Cela n'est pas envisageable.

Une réponse alternative, plus graduée, peut être proposée. Un dépassement exceptionnel d'une partition pourrait déclencher une inscription au fichier d'évènements et une succession de défauts déclencheraient alors un blocage de la partition. Ce blocage peut être temporaire ou définitif en fonction de la taille du dépassement et de la répétition. Cela permet la prise en compte de partitions complexes, à l'instar de celles citées précédemment.

Un niveau de priorité peut même être mis en place afin que chaque partition ait sa politique de gestion associée. Ainsi une partition client subirait la politique la plus stricte et une partition de vol se verrait octroyée certains droits supplémentaires pour ne pas mettre en défaut le système.

L'ensemble des politiques de gestions se trouvent au sein de l'hyperviseur, et plus précisément dans la gestion des interruptions.

Hypothèse 2

La deuxième hypothèse met en avant une attaque dissimulée en faisant condamner une autre partition à sa place mais montre aussi le problème d'un hyperviseur partagé. Le problème de l'hyperviseur partagé est son temps de réaction limité en fonction du problème observé. Le délai de réaction a un coût. En effet, le côté dynamique de la réaction doit, dans un partage tel que celui énoncé être le plus faible possible afin de permettre à l'hyperviseur de gérer l'ordonnancement du mieux possible avec le plus de temps possible. Le choix d'un fichier d'évènements n'est pas suffisant dans notre cas. Il ne permet pas une gestion dynamique du problème. Associer un "limiteur"

suspendant la tâche incriminée permettrait à une partition hostile dissimulée de mettre en défaut les autres partitions et ainsi voler du temps processeur.

Il est nécessaire de mettre en place un système de détection de "culpabilité" dans l'hyperviseur. Celui-ci doit en effet prendre en compte les débordements liés à une autre partition. Pour connaître la partition réellement responsable du dépassement mémoire, il faut que l'hyperviseur s'attache plus particulièrement à plusieurs paramètres.

La partition détectée comme ayant dépassé son quota a t'elle fait plus d'accès au cache de niveau 2 que d'habitude ? Si c'est le cas, il est fort probable que cette partition soit associée à un processus malintentionné. Dans le cas inverse, existe t'il une autre partition, précédemment exécutée, ayant dépassé son quota habituel, et se trouvant dans l'hyper-période d'ordonnancement des tâches ? Si tel est le cas, il est possible que la partition fautive soit la partition précédente. Si une autre partition que celle en cours d'exécution a réalisé un nombre d'accès supérieur à la valeur habituelle alors la sanction doit lui être portée.

Cette politique est intéressante mais a ses limites. En effet, une partition événementielle sera détectée comme malveillante à chaque démarrage. Une politique de gestion efficace doit donc être prise en compte pour chaque partition. La politique du compromis reste, en l'état, la plus viable avec notre composant. Il montre ses limites en terme d'information, et il existe une probabilité de détection de faux positifs et négatifs. Cependant, il permet déjà la régulation et la gestion des problèmes de contention mémoire. Il est difficilement envisageable d'améliorer le composant physique, mais il est possible de discriminer de manière plus efficace les partitions par l'hyperviseur. Toute notre politique de défense se base sur la connaissance par l'hyperviseur du nombre d'accès maximal d'une partition. Mais comment la calculer sans obtenir une valeur trop faible ou trop élevée ?

11.3 Calcul du nombre d'accès d'un programme

Pour connaître le nombre d'accès mémoire, il est nécessaire de pouvoir déterminer le chemin d'exécution amenant au plus grand nombre d'appel mémoire. La base serait de prendre le nombre d'accès mémoire sans considérer l'existence de caches. Cette première idée est intéressante mais rend, pour les systèmes temps-réel modernes, des partitions non ordonnançables car trop grandes. L'existence et la création du cache a été déterminant pour résoudre ce problème et doit également être pris en compte.

Nous décomposons donc notre idée en deux sous parties :

— recherche du chemin d'exécution amenant au plus grand nombre d'accès mémoire ;
— adaptation aux caches.

Recherche du plus grand nombre d'accès mémoire

Cette problématique est quasi identique à celle du pire temps d'exécution. En effet, nous cherchons dans un système complexe, à maximiser une variable : le nombre d'accès mémoire. Si nous conservons les hypothèses précédemment citées dans la Partie II de notre thèse, comme la volonté d'utilisation de techniques dynamiques, alors nous retrouvons ici clairement notre problématique du WCET.

Nous pouvons dès lors proposer de résoudre la recherche du plus grand nombre d'accès mémoire par un algorithme génétique, en l'occurence par l'utilisation de notre proposition adaptée.

C'est une hypothèse réaliste, car le seul changement fondamental est la mesure réalisée sur le système embarqué. En effet, notre algorithme prenait en compte de nombreuses données ; le contexte, qu'il faut conserver tel quel, ainsi que les paramètres, tout aussi importants. Seule la fonction de *fitness* prend en considération la mesure à observer. Nous proposons de remplacer le pire temps d'exécution, par le nombre d'accès mémoire. Cela est rendu possible par le composant que nous avons réalisé précédemment. Celui-ci combiné à l'absence de cache lors de l'évaluation nous permet de connaître exactement le nombre d'accès mémoire.

Le résultat obtenu par notre algorithme génétique, dans un temps raisonnable, ne nécéssite pas de ré-évaluation complète par un spécialiste de l'analyse statique et permet un gain de temps pour la vérification moyenne du nombre d'accès mémoire. En revanche, le problème de cette technique reste identique. En l'absence de l'analyse statique, il n'y a pas de garantie sur le résultat et il est possible d'avoir découvert une valeur locale.

La meilleure des solutions serait donc, à l'instar de la conclusion dans la Partie II, de faire réaliser une analyse statique du nombre d'accès en mémoire, puis de le vérifier par notre algorithme génétique adapté. L'intérêt de l'industriel ou du client serait d'intégrer un algorithme génétique simple pour vérifier l'ensemble des caractéristiques du système à évaluer. Rien ne diffère de l'algorithme génétique, excepté la variable d'optimisation, ce qui rend le travail à réaliser par l'homme quasi nul. Ainsi, un même ingénieur peut évalué l'ensemble du système sans formation ou temps supplémentaire. En effet, seul le travail automatique des machines sera augmenté en temps de calcul, mais la mise en place du dispositif sera comprise dans le temps d'installation du calcul du WCET.

La mise en place d'un test d'évaluation rapide offrirait une plus value certaine à notre algorithme génétique sur système complexe.

Adaptation aux caches

La valeur obtenue par notre algorithme génétique ne prend pas en compte les effets bénéfiques du cache et offre une valeur d'accès mémoire brute. Il faut donc réussir à prendre en compte l'effet du cache sur les partitions exécutées. Cela n'est pas un problème simple. En effet, la valeur dépend des autres partitions sur le système. Il est possible de prendre en compte une valeur pessimiste fixe mais les performances du système seront à la baisse. Il faut donc approcher une hypothèse de recherche automatique du nombre d'accès au cache.

Ce qui est certain, c'est le nombre d'accès au cache L1. Chaque partition étant seule sur son processeur lors de son exécution, le cache L1 qui lui est dédié reste donc fixe. De plus, le cache L1 est remis a zéro à chaque changement de partition. Il est donc parfaitement prévisible. Seul le nombre d'accès au cache L2 est difficile à définir.

La solution la plus simple serait de ré-exécuter le chemin obtenu par notre algorithme génétique, impactant le plus la mémoire, et observer en comparaison le nombre d'accès au cache L1 et au cache L2. Cette valeur nous offre un ordre d'idée du taux d'accès au cache. Plusieurs résultats sont envisageables, dont voici les cas :

— tous les accès mémoires sont en réalité des accès au cache L1 ;
— les accès sont très nombreux en cache L2 ;

— les accès sont principalement en mémoire.

Dans le premier cas, le pire chemin obtenu par notre algorithme n'est pas le plus intéressant. Ce chemin correspond à un chemin parfaitement prévisible et n'ayant aucune optimisation au niveau du cache L2. Le second chemin le plus important doit être étudié dans le cas ou celui-ci dépend du cache L2. On opère jusqu'à trouver un chemin sollicitant peu le cache de niveau 1 et beaucoup plus la mémoire ou le cache de niveau 2. Dans le deuxième cas, le nombre d'accès très important au cache L2 démontre que la partition sera fortement sensible aux variations des autres partitions sur le système. Il risque d'y avoir des problèmes de contention mémoire lors de l'exécution de cette partition dépendante du cache L2. Définir une valeur pessimiste du nombre d'accès à la mémoire plutôt qu'au cache de niveau 2 serait donc plus réaliste et approprié. De même pour le dernier cas qui nous fourni directement une valeur plutôt pessimiste. Il est important de prendre une marge de "sûreté" pour garantir qu'une autre partition ne déclenche pas de détection de contention mémoire inutilement.

Pour conclure, il s'avère nécessaire de sur-évaluer le nombre d'accès en mémoire en prenant une équation (nécessitant amélioration) tel qu'égale au nombre d'accès réels en mémoire plus un pourcentage du nombre d'accès au cache L2.

11.4 Conclusion

Les perspectives nous montrent que notre proposition s'intègre parfaitement dans un système complet de mesure industrielle pour systèmes embarqués temps-réel spatiaux. Il est possible de réguler entièrement la contention mémoire avec un ensemble de dispositifs réalistes et intégrant la complexité des nouveaux systèmes. Ce chapitre conclut sur notre deuxième et dernier apport majeur de la thèse.

Quatrième partie

Conclusions et perspectives

Chapitre 12

Conclusions et perspectives

Sommaire

Les satellites hypervisés multiprocesseurs constituent un apport majeur au monde de l'électronique embarqué dans le domaine spatial. Grâce à l'utilisation de ces nouvelles technologies, le coût de production est restreint et le nombre de satellites nécessaire dans l'espace est diminué par autant de clients que peut l'accepter le système. Les enjeux sont multiples, notamment dans le domaine commercial. Ce type de système est particulièrement modulable et adaptable aux différents besoins. Faisant gagner un temps de production précieux, il améliore encore les performances générales d'un ensemble toujours à la pointe de la technologie.

Malgré tous ces avantages, cette nouvelle technologie n'a pas encore l'intégralité des outils industriels pour une exploitation optimale. L'évolution des normes est constante et répond peu à peu au besoin toujours croissant. La conception même d'un satellite est aujourd'hui entièrement modifiée et le schéma de production industriel fortement complexifié. L'appel à des sociétés de sous-traitance, ou la mise a disposition de plate-forme de développement pour le client acheteur rend la production et le contrôle de celle-ci particulièrement difficile. A cela s'ajoutent de nouvelles menaces. Le caractère géo-stratégique des satellites et le pouvoir qu'ils représentent, devient une cible prioritaire pour tout état, groupe, ou entreprise souhaitant contrôler ou perturber un ou plusieurs clients. Le noeud de communication que représente le satellite est fondamental et doit être protégé au mieux face aux nouveaux types d'attaques.

12.1 Problématique

Dans cette thèse, nous développons dans un premier temps l'évolution de la production d'un satellite et son impact en terme de sûreté de fonctionnement et de sécurité. C'est cet impact qui nous permet d'aborder deux problématiques majeures.

La première est directement liée au schéma industriel mis en place pour la production des satellites. Le schéma retenu par l'industrie est de sous-traiter au maximum le travail à effectuer afin de diminuer les coûts sur le long terme. Cela implique que le producteur final n'a pas la maîtrise complète du système. C'est d'autant plus vrai que le client produit son propre code. Le client comme le producteur final souhaitent vérifier la qualité du satellite, sans s'impliquer dans la complexité du système étudié. Ils utilisent alors des techniques dynamiques qu'ils maîtrisent déjà. Les sujets de questionnement découlent donc de l'analyse temporelle de ces techniques. Le système étant de type temps-réel, il est nécessaire de connaître le temps maximal pris par chaque fonction, quel que soit l'état du système. La problématique est donc : *Comment observer dynamiquement le pire temps d'exécution d'une fonction sur un satellite aussi complexe ?*.

La deuxième problématique est liée aux caractéristiques des systèmes multi-coeur. Chaque évolution technologique introduit de nouvelles problématiques. C'est en particulier le cas du multi-coeur qui fait ré-apparaître le phénomène bien connu de la contention mémoire. Un système embarqué de manière générale utilise souvent de la mémoire qui, dans notre cas, est partagée entre différents processeurs. Une politique de gestion des accès a été mise en place. Néanmoins, l'intégration du cache commun à des fins de performances crée un effet de bord pouvant de façon contradictoire les détériorer, ou mettre le système en défaut. Le contenu du cache, exécuté de façon malveillante aurait des conséquences graves sur le système. Notre deuxième problématique est donc : *Comment gérer le phénomène de contention mémoire dans un satellite ?*.

12.2 Principales contributions

Dans ce rapport de thèse, nous avons découpé les contributions en trois parties. La première présentant les attaques réalisables sur les nouveaux systèmes satellitaires, et les deux autres répondant chacune à l'une des deux problématiques citées ci-dessus.

La première contribution, développée en Partie I, montre l'évolution des satellites et la difficulté de maîtriser entièrement les actes hostiles sur celui-ci. Elle met également en avant des techniques malveillantes, des essais, ainsi que des mesures sur système réel. Cette partie démontre les capacités à mettre en défaut un système pourtant correctement implémenté en modifiant le paramètre de temps, par exemple. La contribution démontre également que le développement d'un système avec une optique de sûreté de fonctionnement n'est pas suffisant pour se prémunir d'actes malintentionnés. Un certain nombre d'attaques ont été détaillées pour le démontrer.

La seconde contribution, faisant écho à la première, répond à la problématique : *Comment observer dynamiquement le pire temps d'exécution d'une fonction sur un satellite aussi complexe ?*. En effet, une meilleure connaissance du système et une plus grande maîtrise de celui-ci permet d'éviter un grand nombre d'effets de bord et donc d'attaques. Dans cette contribution nous avons repris les travaux existant sur la problématique dynamique. Nous y avons intégré la capacité de notre algorithme à s'adapter de façon autonome à la complexité du système qu'il étudie. Nous proposons la reprise d'un algorithme génétique permettant la vérification des valeurs déjà observées en amont par un autre industriel. Ceci tout en conservant l'idée d'améliorer un principe dynamique souvent délaissé au profit du statique et pourtant nécessaire aux méthodes hybrides.

La troisième et dernière contribution répond à la question : *Comment gérer le phénomène de contention mémoire dans un satellite ?*. Cette question, en apparence indépendante de la seconde

partie, répond aux problème d'évolution des systèmes. Nous proposons un composant discret et efficace, ne prenant pas de cycle processeur et permettant d'observer physiquement la contention mémoire. Il n'impose pas la gestion du problème observé, mais offre une information permettant à un système sécurisé, ici sous la forme d'un hyperviseur, de prendre une décision adéquate. Nous exprimons les différents cas possibles en fin de cette partie et déterminons également comment observer le nombre d'accès mémoire d'une fonction dans un temps donné. Cette dernière est en lien direct avec notre apport de la partie II.

12.3 Perspectives

De nombreux travaux restent à réaliser dans ce domaine. En illustration, la première partie de cette thèse pourrait encore être agrémentée de nouvelles attaques. Celles-ci ouvriraient de nouvelles perspectives liées aux technologies utilisées, aux principes de fabrications et autre. La gamme des attaques réalisables est telle qu'il est difficile d'en faire une liste exhaustive. Cet état de l'art est amené à changer rapidement, et est donc en perpétuelle évolution.

Les perspectives de la deuxième partie sont axées sur plusieurs points distincts. L'un d'eux, majeur, est l'évolution de l'algorithmie génétique avec des méthodes encore plus récentes et précises, permettant d'affiner la fiabilité et les résultats de notre solution. Nous pourrions ainsi faire évoluer la taille de la population, le taux de croisement, ou le taux de mutation, ainsi que la taille du contexte du système etc... Une autre perspective serait la mise en place d'une analyse statique basée sur l'analyse de binaires afin d'acquérir une valeur supérieure au pire temps d'exécution réel d'un système. Cela permettrait en cas de doute des résultats fournis par le sous-traitant, de refaire des mesures exploitables et indépendantes avec un certain niveau de garantie des résultats.

La troisième partie offre également certaines perspectives. Notre composant fournit le nombre d'accès à chaque mémoire, et ce pour chaque processeur. Il est difficile d'imaginer une évolution majeure de celui-ci. De nombreux points sont à étudier, on peut citer : comment déterminer le nombre d'accès mémoire normal d'une application ? et quelle politique utiliser si cette application a dépassé son état ? Ces deux points ont été développés mais peuvent être présentés dans toute leur complexité.

12.4 Synthèse générale

Cette thèse tourne autour du monde de l'aérospatial et plus particulièrement de la conception des satellites. Elle met l'accent sur les problématiques liées à la conception de ces systèmes et leurs implications dans la sécurité face à des actes malveillants. Des questions se posent sur l'intérêt de chaque évolution quant à la sécurité et aux changements à appliquer pour y parvenir.

Nous nous sommes intéressés à l'observation de deux paramètres, le temps et le nombre d'accès pris par une fonction s'exécutant sur un satellite complexe. Nous avons donc proposé une méthode de mesure et une méthode d'observation de ces deux paramètres. L'un se basant sur une série de tests réalisés automatiquement et l'autre sur la détection de dépassement de quota.

Le choix des approches appliquées, compte tenu de l'état de l'art, et des recherches existantes, nous a permis de comparer les difficultés de mise en place de ces deux solutions et leur importance dans la conception des satellites modernes.

Les expérimentations que nous avons réalisé sur nos outils ont montré leur efficacité. Cependant des améliorations peuvent toujours être apportées. Nous avons proposé un ensemble de travaux permettant d'améliorer l'observation du pire temps d'exécution d'une fonction ainsi que de son dépassement du nombre d'accès mémoire lors d'une utilisation malveillante.

Bibliographie

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11) :2–13, 2006.

[ABHPW10] S Ali, L C Briand, H Hemmati, and R K Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36 :742–762, 2010.

[ACBL96] Jean Arlat, Alain Costes, J-P Blanquart, and Jean-Claude Laprie. *Guide de la sûreté de fonctionnement*. Cépaduès-éditions, 1996.

[AG09] François Armand and Michel Gien. A practical look at micro-kernels and virtual machine monitors. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–7. IEEE, 2009.

[ANLBHM00] L Amsellem, JL Noyer, T Le Bourgeois, and M Hossaert-McKey. Comparison of genetic diversity of the invasive weed rubus alceifolius poir.(rosaceae) in its native range and in areas of introduction, using amplified fragment length polymorphism (aflp) markers. *Molecular Ecology*, 9(4) :443–455, 2000.

[Bar10] Richard Barry. Using the freertos real time kernel a practical guide. 2010.

[BBCL11] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *FM 2011 : Formal Methods*, pages 231–245. Springer, 2011.

[BBD+10] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The vmware mobile virtualization platform : is that a hypervisor in your pocket ? *ACM SIGOPS Operating Systems Review*, 44(4) :124–135, 2010.

[BCE+78] A Boggess, FA Carr, DC Evans, D Fischel, HR Freeman, CF Fuechsel, DA Klinglesmith, VL Krueger, GW Longanecker, and JV Moore. The iue spacecraft and instrumentation. *Nature*, 275 :372–377, 1978.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5) :164–177, 2003.

[BDG+00] J Benedicto, S Dinwiddy, G Gatti, R Lucas, and M Lugert. Galileo : Satellite system design. *European Space Agency*, 2000.

[BDL10] Christophe Belleval, Ioana Deniaud, and Christophe Lerch. Modèle de conception à base de réseau de contradictions. le cas de la conception des microsatellites au cnes. *Information Science for Decision Making*, 40, 2010.

[Bel07] Fabrice Bellard. Qemu open source processor emulator. *URL : http ://www. qemu. org*, 2007.

[BER97] Philippe BERNASCOLLE. Le telescope spatial hubble. 1997.

[BFB09] Jacques Brygier, Rudolf Fuchsen, and Holger Blasum. Pikeos : Safe and secure virtualization in a separation microkernel. Technical report, Technical report, SYSGO, 2009.

[BICL11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In *Smart Card Research and Advanced Applications*, pages 283–296. Springer, 2011.

[BJ00] P.M.S. Bueno and M. Jino. Identification of potentially infeasible program paths by monitoring the search for test data. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, IEEE Piscataway, NJ, Grenoble, France (2000), pp. 209–218*, 2000.

[BJF05] Shaofeng Bian, Jihang Jin, and Zhaobao Fang. The beidou satellite positioning system and its positioning accuracy. *Navigation*, 52(3) :123–129, 2005.

[BK10] Iain Bate and Usman Khan. WCET analysis of modern processors using multi-criteria optimisation. *Empirical Software Engineering*, 16(1) :5–28, June 2010.

[Bou06] J Boukachour. Rapport de Mémoire Application des Algorithmes Génétiques au Problème du Voyageur de Commerce Table des matières. 2006.

[Bou07] Arnaud Boudou. Essai de vmware fusion. 2007.

[Bra10] David Brash. Extensions to the armv7-a architecture. In *Hot Chips*, volume 22, 2010.

[Bur91] Alan Burns. Scheduling hard real-time systems : a review. *Software Engineering Journal*, 6(3) :116–128, 1991.

[BZK11] Sven Bunte, Michael Zolda, and Raimund Kirner. Let's get less optimistic in measurement-based timing analysis. *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, pages 204–212, June 2011.

[Can09] Gaëtan Canivet. *Analyse des effets d'attaques par fautes et conception sécurisée sur plate-forme reconfigurable*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2009.

[CCP+14] Bekim Cilku, Alfons Crespo, Peter Puschner, Javier Coronel, and Salvador Peiro. A memory arbitration scheme for mixed-criticality multicore platforms. In *Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 27–32, 2014.

[CFLH06] Ying-wu Chen, Yan-shen Fang, Ju-fang Li, and Ren-jie HE. Constraint programming model of satellite mission scheduling. *JOURNAL-NATIONAL UNIVERSITY OF DEFENSE TECHNOLOGY*, 28(5) :126, 2006.

[CP01] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 37–44. IEEE, 2001.

[CP03] Antoine Colin and Stefan M Petters. Experimental Evaluation of Code Properties for WCET Analysis. 2003.

[CYC+09] James R Clapper, John J Young, James E Cartwright, John G Grimes, Sue C Payton, SJ Stackley, and D Popps. Fy2009-2034 unmanned systems integrated roadmap. *Department of Defense : Office of the Secretary of Defense Unmanned Systems Roadmap*, 2009.

[CYC+13] James R Clapper, John J Young, James E Cartwright, John G Grimes, Sue C Payton, SJ Stackley, and D Popps. Fy2013-2034 unmanned systems integrated roadmap. *Department of Defense : Office of the Secretary of Defense Unmanned Systems Roadmap*, 2013.

[DC98] Alok Das and Richard Cobb. Techsat 21-space missions using collaborating constellations of satellites. 1998.

[DD08] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development : A systematic review. *Information and software technology*, 50(9) :833–859, 2008.

[DDAN12] Anthony Dessiatnikoff, Yves Deswarte, Eric Alata, and Vincent Nicomette. Potential attacks on onboard aerospace systems. *IEEE Security & Privacy*, 10(4) :0071–74, 2012.

[Deh11] Amine Dehbaoui. *Analyse Sécuritaire des Émanations Électromagnétiques des Circuits Intégrés*. PhD thesis, Montpellier 2, 2011.

[DG] Nicolas Durand and Jean-baptiste Gotteland. Algorithmes génétiques appliqués à la gestion du trafic aérien Plan de l ' exposé.

[DHW97] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6) :779–805, 1997.

[Dip] A Dipanda. Algorithmes génétiques Algorithmes génétiques.

[DN09] Christophe De Nardi. *Techniques d'analyse de défaillance de circuits intégrés appliquées au descrambling et à la lecture de données sur des composants mémoires non volatiles*. PhD thesis, Toulouse, INSA, 2009.

[ENBSH12] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth bandit : Understanding memory contention. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 116–117. IEEE, 2012.

[esi15] esieespace. www.esieespace.fr. Technical report, esiee, 2015.

[ESX] VMware ESXi. Bare metal hypervisor.

[Fer06] Daniel R Ferstay. *Fast secure virtualization for the arm platform*. PhD thesis, The University of British Columbia, 2006.

[FFBM06] Olivier Faurax, Laurent Freund, Frédéric Bancel, and Traian Muntean. Une méthode générique pour l'injection de fautes dans les circuits. *Actes des 9èmes Journées Nationales du Réseau Doctoral en Microélectronique*, 2006.

[FMG+01] Wendy L Freedman, Barry F Madore, Brad K Gibson, Laura Ferrarese, Daniel D Kelson, Shoko Sakai, Jeremy R Mould, Robert C Kennicutt Jr, Holland C Ford, John A Graham, et al. Final results from the hubble space telescope key project to measure the hubble constant. *The Astrophysical Journal*, 553(1) :47, 2001.

[For00] US Air Force. Analysis of the intel pentium's ability to support a secure virtual machine monitor. 2000.

[FSLM12] Laurent Fribourg, Romain Soulat, David Lesens, and Pierre Moro. Robustness analysis for scheduling problems using the inverse method. In *Temporal Representation and Reasoning (TIME), 2012 19th International Symposium on*, pages 73–80. IEEE, 2012.

[FZBW10] Birhanu Fufa, Chen Zhao-Bo, and Ma Wensheng. Modeling and simulation of satellite solar panel deployment and locking. *Information Technology Journal*, 9(3) :600–604, 2010.

[GÓ5] L E S Algorithmes Génétiques. Les algorithmes génétiques. pages 1–31, 2005.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks : Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).*, pages 136–146, 2010.

[GE02] Robin Gubby and John Evans. Space environment effects and satellite design. *Journal of atmospheric and solar-terrestrial physics*, 64(16) :1723–1733, 2002.

[GGPZ14] Venkita Giri, Shankar Giri, Gregory A Price, and Seyed Zekavat. Generic sdr architecture : Vendor independent implementation. In *Aerospace Conference, 2014 IEEE*, pages 1–6. IEEE, 2014.

[GJP14] Grimaud G., Iguchi-Cartigny J., and Buret P. Genetic algorithm for DWCET Evaluation on complex platform. In *IEEE International Symposium on Industrial Embedded Systems*, jun 2014.

[GJP15] Grimaud G., Iguchi-Cartigny J., and Buret P. Contexte d'exécution dans la recherche de pire temps d'exécution d'un système complexe. In *Compas'15*, juin 2015.

[GL01] Pierrick Grandjean and François Lecouat. Scheduling and plan execution : from ground segment automation to autonomous spacecraft operation concepts. In *Proceedings of the On-Autonomy Workshop, ESTEC, Noordwijk, Netherlands*, pages 17–19, 2001.

[Gol89] David E Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, volume Addison-We. 1989.

[Gro03] HG Gross. An evaluation of dynamic, optimisation-based worst-case execution time analysis. *Proceedings of the International Conference on ...*, 2003.

[Gui07] Sylvain Guilley. *Contre-mesures géométriques aux attaques exploitant les canaux cachés*. PhD thesis, Télécom ParisTech, 2007.

[Har07] Mark Harman. The Current State and Future of Search Based Software Engineering The Current State and Future of Search Based Software Engineering. 2007.

[Haw91] DP Haworth. Military satellite communications. *IEE telecommunications series*, 24 :296–328, 1991.

[HBR00] Jerry D Holmes, Kenneth S Barron, and Anthony Reid. Jamming suppression of spread spectrum antenna/receiver systems, October 31 2000. US Patent 6,141,371.

[Hei09] Gernot Heiser. Hypervisors for consumer electronics. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5. IEEE, 2009.

[HL10] Gernot Heiser and Ben Leslie. The okl4 microvisor : Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.

[Hol10] Simon Holmbacka. *Task migration in virtualized multi-core real-time systems*. PhD thesis, Master's thesis, Åbo Akademi University, 2010.

[Hon03] Jerry Honeycutt. Microsoft virtual pc 2004 technical overview. *Microsoft, Nov*, 2003.

[HP06] Casse H and Sainrat P. OTAWA, a framework for experimenting WCET computations. *3rd European Congress on Embedded Real-Time*, pages 1–8, 2006.

[IC] Mention Informatique and Antoine Colin. Phd antoine, irisa.

[IMTSB11] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, and Moris Behnam. Hard real-time support for hierarchical scheduling in freertos. In *23rd Euromicro Conference on Real-Time Systems*, pages 51–60, 2011.

[IRS+04] Mohamed Ibnkahla, Quazi Mehbubar Rahman, Ahmed Iyanda Sulyman, Hisham Abdulhussein Al-Asady, Jun Yuan, and Ahmed Safwat. High-speed satellite mobile communications : technologies and challenges. *Proceedings of the IEEE*, 92(2) :312–339, 2004.

[ISM09] Asif Iqbal, Nayeema Sadeque, and Rafika Ida Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2110, 2009.

[Isn01] Jaques Isnard. La belgique rejoint la france dans le project de satellite-espion militaire hélios 2. *Le Monde*, page 3, 2001.

[JFA13] David Jaramillo, Borko Furht, and Ankur Agarwal. Virtualization techniques for mobile devices. *International Review on Computers & Software*, 8(8), 2013.

[Joh01] David S Johnson. A Theoretician's Guide to the Experimental Analysis of Algorithms. *American Mathematical Society*, 220 :1–36, 2001.

[Jon11] M Tim Jones. „virtualization for embedded systems. *The how and why of small-device hypervisors", IBM developerWorks*, 2011.

[JW98] Mark S Johnstone and Paul R Wilson. The memory fragmentation problem : solved ? In *ACM SIGPLAN Notices*, volume 34, pages 26–36. ACM, 1998.

[Kan73] AHG Rinnooy Kan. *The machine scheduling problem*, volume 27. Interfaculty for Graduate Studies in Management, 1973.

[KBP+10] Young W Kwon, Luke N Brewer, Rudolf Panholzer, Daniel J Sakoda, and Chanman Park. Direct manufacturing of cubesat using 3-d digital printer and determination of its mechanical properties. Technical report, DTIC Document, 2010.

[KHQ66] DG King-Hele and Eileen Quinn. Table of the earth satellites launched in 1965. *Planetary and Space Science*, 14(9) :817–838, 1966.

[KIE02] Bernhard KIEWE. The electrical power system of fhe small satellite champ. *ESA SP*, pages 383–388, 2002.

[Kis09] Jan Kiszka. Towards linux as a real-time hypervisor. In *Proceedings of the 11th Real-Time Linux Workshop*, pages 215–224. Citeseer, 2009.

[KK99] Oliver Kömmerling and Markus G Kuhn. Design principles for tamper-resistant smartcard processors. In *USENIX workshop on Smartcard Technology*, volume 12, pages 9–20, 1999.

[KK12] David Kleidermacher and Mike Kleidermacher. *Embedded systems security : practical methods for safe and secure software and systems development*. Elsevier, 2012.

[KLE] DAVID KLEIDERMACHER. Connected-device.

[KLM+04] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, and Srivaths Moderator-Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760. ACM, 2004.

[KOW07] Leslie D Kohn, Kunle A Olukotun, and Michael K Wong. Multi-core multi-thread processor, April 24 2007. US Patent 7,209,996.

[KWHM] P. Kulkarni, D. Whalley, C. Healy, and F. Mueller. Tuning the WCET of embedded applications. *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, pages 472–481.

[KWRP] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using Measurements as a Complement to Static Worst-Case Execution Time Analysis .

[Lan87] Pierre Langereux. L'italie et l'espagne s' associent au satellite français hélios. *Air et Cosmos*, (1139) :44, 1987.

[Lau11] Yonne Lautre. Déchets spatiaux brèves. 2011.

[LHT+09] Simon Lee, Amy Hutputanasin, Armen Toorian, Wenschel Lan, and R Munakata. Cubesat design specification. *The CubeSat Program*, 2009.

[LLL+11] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android : a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 39–50. ACM, 2011.

[M+09] Riki Munakata et al. Cubesat design specification rev. 12. *The CubeSat Program, California Polytechnic State University*, 1, 2009.

[MAC] MAC. Parallels desktop.

[Mar78] James Martin. Communications satellite systems. *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1978. 419 p.*, 1, 1978.

[Mar12] Amine Marref. Evolutionary Algorithms for Parametric WCET Analysis. pages 1–24, 2012.

[MCS04] Phil Mcminn, Regent Court, and Portobello Street. Search-based Software Test Data Generation : A Survey. pages 1–58, 2004.

[Mil87] Jonathan K Millen. Covert channel capacity. In *null*, page 60. IEEE, 1987.

[Mon99] Thierry Monnier. *Durcissement de circuits convertisseurs a/n rapides fonctionnant en environnement spatial*. PhD thesis, 1999.

[MRPC10] Miguel Masmano, Ismael Ripoll, S Peiró, and A Crespo. Xtratum for leon3 : an open source hypervisor for high integrity systems. In *European Conference on Embedded Real Time Software and Systems. ERTS2*, volume 2010, 2010.

[MUL02] Kevin C McCarthy, Eugenie V Uhlmann, and Niall R Lynam. Complete mirror-based global-positioning system (gps) navigation solution, November 5 2002. US Patent 6,477,464.

[MW84] P Marx and H Laporte Weywada. Ariane 5-a new launcher for the 90's. 1984.

[MW98] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245)*, 1998.

[Ngu01] Anne-Thérèse Nguyen. Les échanges technologiques entre la france et les états-unis : les télécommunications spatiales (1960-1985). *Flux*, (1) :17–24, 2001.

[Ngu11] Minh Huu Nguyen. *Sécurisation de processeurs vis-à-vis des attaques par faute et par analyse de la consommation.* PhD thesis, Paris 6, 2011.

[NKS+03] Koji Nakaya, Kazuya Konoue, Hirotaka Sawada, Kyoichi Ui, Hideto Okada, Naoki Miyashita, Masafumi Iai, Tomoyuki Urabe, Nobumasa Yamaguchi, Munetaka Kashiwa, et al. Tokyo tech cubesat : Cute-i. 2003.

[NPST02] Isaac Nason, Jordi Puig-Suari, and Robert Twiggs. Development of a family of picosatellite deployers based on the cubesat standard. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 1, pages 1–457. IEEE, 2002.

[NSIC+09] Agnès Cristèle Noubissi, A Séré, Julien Iguchi-Cartigny, Jean-Louis Lanet, Guillaume Bouffard, and Julien Boutet. Cartes à puce : Attaques et contremesures. *MajecSTIC*, 16 :1112, 2009.

[Ora13] VM Oracle. Virtualbox. *User Manual–2013*, 2013.

[Par05] Lisa Parks. *Cultures in orbit : Satellites and the televisual.* Duke University Press Durham, NC, 2005.

[PDSEK] Anup Patel, Mai Daftedar, Mohmad Shalan, and M Watheq El-Kharashi. Embedded hypervisor xvisor : A comparative analysis.

[PG74] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7) :412–421, 1974.

[PG01] Joseph C Pemberton and Flavius Galiber. A constraint-based approach to satellite scheduling. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 57 :101–114, 2001.

[PHP99] R.P. Pargas, M.J. Harrold, and R.R. Peck. Test-data generation using genetic algorithms. In *J. Software Test. Verif. Reliab., 9 (4) (1999), pp. 263–282*, 1999.

[PKI+02] GM Polischuk, VI Kozlov, VV Ilitchov, AG Kozlov, VA Bartenev, VE Kossenko, NA Anphimov, SG Revnivykh, SB Pisarev, and AE Tyulyakov. The global navigation satellite system glonass : Development and usage in the 21st century. Technical report, DTIC Document, 2002.

[PS09] Vincent PERY and Thierry SERDIN. L'europe et astrium vers la station spatiale internationale : La mission jules verne. *Essais & simulations*, (99) :42–44, 2009.

[PSTA01] Jordi Puig-Suari, Clark Turner, and William Ahlgren. Development of the standard cubesat deployer and a cubesat class picosatellite. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 1, pages 1–347. IEEE, 2001.

[PU89] P Puschner and Technische Universita. Calculating the Maximum Execution Time of Real-Time Programs. 176 :159–176, 1989.

[Pua06] Isabelle Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 10–pp. IEEE, 2006.

[RH10] David Rohret and Jonathan Holston. Exploitation of blue team satcom and milsat assets for red team covert exploitation and back-channel communications. In *International Conference on Information Warfare and Security*, page 288. Academic Conferences International Limited, 2010.

[RHP99] P. Pargas Roy, Mary Jean Harrold, and Robert R. Peck. Test Data Generation Using Genetic Algorithms. *Journal of Software Testing, Verification and Reliability*, page 19, 1999.

[RS07] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 179–188. ACM, 2007.

[RSW+06] Mahendra Ramachandran, Ned Smith, Matthew Wood, Sharad Garg, Jim Stanley, Eswar Eduri, Rinat Rappoport, Arie Chobotaro, Carl Klotz, and Lori Janz. New client virtualization usage models using intel virtualization technology. *Intel Technology Journal*, 10(3), 2006.

[RWR96] Timothy G Reddin, David S Walsh, and Jeremy S Round. Parallel computer having mac-relay layer snooped transport header to determine if a message should be routed directly to transport layer depending on its destination, December 24 1996. US Patent 5,588,121.

[Sav13] Ralph Savelsberg. An analysis of north korea's satellite launches. *Journal of Military Studies*, 3(1), 2013.

[SBF97] J.E. Smith, M. Bartley, and T.C. Fogarty. Microprocessor design verification by two-phase evolution of variable length tests. In *Proceedings of the IEEE International Conference on Evolutionary Computation, IEEE Piscataway, NJ, Indianapolis, IN, pp. 453–458*, 1997.

[Sev] Michael Sevilla. Host–based intrusion detection system using the leon3.

[SGGS98] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[SKN+13] Jeremy Straub, Christoffer Korvald, Anders Nervold, Atif Mohammad, Noah Root, Nicholas Long, and Donovan Torgerson. Openorbiter : A low-cost, educational prototype cubesat mission architecture. *Machines*, 1(1) :1–32, 2013.

[Smi68] Delbert D Smith. Legal ordering of satellite telecommunication : Problems and alternatives, the. *Ind. LJ*, 44 :337, 1968.

[Spe06] ARINC Specification. 653-2 : Avionics application software standard interface : Part 1-required services. Technical report, Technical report, Avionics Electronic Engineering Committee (ARINC)(March 2006), 2006.

[SR12] Josef Strnadel and Peter Rajnoha. Reflecting rtos model during wcet timing analysis : Msp430/freertos case study. *Acta Electrotechnica et Informatica*, 12(4) :17–29, 2012.

[SS07] Rathijit Sen and YN Srikant. Wcet estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 203–212. ACM, 2007.

[Sta85] Paul B Stares. The militarization of space : Us policy, 1945-1984. 1985.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.

[Tan06] Lili Tan. The worst case execution time tool challenge 2006 : The external test. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 241 – 248, University of Duisburg-Essen, 2006.

[TH03] Keith A Temple and Oved W Hanson. Method of determining refrigerant charge level in a space temperature conditioning system, June 3 2003. US Patent 6,571,566.

[TSWW06] M. Tlili, H. Sthamer, S. Wappler, and J. Wegener. Improving Evolutionary Real-Time Testing by Seeding Structural Test Data. *2006 IEEE International Conference on Evolutionary Computation*, pages 885–891, 2006.

[Val01] Thomas Vallée. Présentation des algorithmes génétiques et de leurs applications en économie. (1995) :1–23, 2001.

[Var02] Philippe Varnoteaux. La part du cnrs dans les débuts de la conquête de l'espace (1945-1965). *La revue pour l'histoire du CNRS*, (6), 2002.

[VBS05] Tanya Vladimirova, Roohi Banu, and M Sweeting. On-board security services in small satellites. In *MAPLD Proceedings*, 2005.

[Ver82] J Vercheval. Les 25 ans du lancement du 1er satellite artificiel : De spoutnik 1 à columbia. *Ciel et Terre*, 98 :293, 1982.

[VF10] Eric Vetillard and Anthony Ferrari. Combined attacks and countermeasures. In *Smart Card Research and Advanced Application*, pages 133–147. Springer, 2010.

[VG02] Frank Vahid and Tony Givargis. *Embedded system design : a unified hardware/software introduction*, volume 4. John Wiley & Sons New York, NY, 2002.

[VH11] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 11. ACM, 2011.

[Vil88] Alain Villemeur. Sûreté de fonctionnement des systèmes industriels. 1988.

[VLW+12] Tanja E. J. Vos, Felix F. Lindlar, Benjamin Wilmes, Andreas Windisch, Arthur I. Baars, Peter M. Kruse, Hamilton Gross, and Joachim Wegener. Evolutionary functional black-box testing in an industrial setting. *Software Quality Journal*, 21(2) :259–288, February 2012.

[VMW05] ESX VMWare. Server, 2005.

[vSH07] Carl van Schaik and Gernot Heiser. High-performance microkernels and virtualisation on arm and segmented architectures. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems, Sydney, Australia*, 2007.

[VV09] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[VVPV08] Jacques Villain, Jacques Villain, France Physicist, and Jacques Villain. *À la conquête de l'espace : de Spoutnik à l'homme sur Mars*. Vuibert Ciel & Espace, 2008.

[VZS98] VUNDEZUR VON, GEWICHTUNGSKOEFFIZIENTEN ZUR, and DVONLMIT STERNSENSOR. Calculs d'attitude avec un capteur stellaire. *mars*, 1998 :119–127, 1998.

[WEE+] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstr. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. V :1–47.

[Whi98] Thomas G Whitten. Method and computer program product for generating a computer program product test that includes an optimized set of computer program product test cases, and method for selecting same, September 8 1998. US Patent 5,805,795.

[WHK+07] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R Nair, Mauricio Breternitz Jr, Zhiwei Ying, and Youfeng Wu. Stardbt : an efficient multi-platform dynamic binary translation system. In *Advances in Computer Systems Architecture*, pages 4–15. Springer, 2007.

[Wlo07] Joachim J Wlodarz. Virtualization : A double-edged sword. *arXiv preprint arXiv :0705.2786*, 2007.

[WRKP05] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 606–611. IEEE, 2005.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D Gribble. Denali : A scalable isolation kernel. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 10–15. ACM, 2002.

[WSJE97] Joachim Wegener, H Sthamer, BF Jones, and DE Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 21(2) :259–288, February 1997.

[WWBD08] Gary Waters, Gary Waters, Desmond Ball, and Ian Dudgeon. *Australia and cyberwarfare*. ANU E Press, 2008.

[XLvdS+08] Tao Xu, JJ Lukkien, PDV van der Stok, Ir PHFM Verhoeven, and Ir B Mesman. Performance benchmarking of freertos and its hardware abstraction, 2008.

[XWLG11] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen : towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48. IEEE, 2011.

[YZ08] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, pages 80–89. IEEE, 2008.

[ZdlP13] Juan Zamorano and Juan A de la Puente. On real-time partitioned multicore systems. *ACM SIGAda Ada Letters*, 33(2) :33–39, 2013.

[Zim06] Dennis Zimmer. *VMware Server & VMware Player*. Galileo Press, 2006.