Architecture E/E et Logicielle pour les Systèmes Temps-Réels à Criticité Multiple : Etude Bibliographique

Daniel Loche

Architecture E/E et Logicielle pour les Systèmes Temps-Réels à Criticité Multiple : Etude Bibliographique

Daniel Loche
LAAS-CNRS, Toulouse, France.
email: dloche@laas.fr

9 février 2018

Résumé

Cette étude bibliographique vise à présenter une première vue d'ensemble sur les architectures Matérielles et Logicielles ainsi que les différentes problématiques inhérentes aux architectures modernes, notamment dans le domaine automobile. Cela s'inscrit dans le contexte du démarrage d'une thèse CIFRE entre le Groupe Renault et le LAAS-CNRS, intitulée "Conception E/E et Logicielle pour la voiture connectée et autonome". L'étude bibliographique est organisée comme suit : contextualisation et motivation qui amène à notre problématique, Architectures Actuelles (matérielle, logicielle ainsi que les processus de développement associés), analyses temps-réel mono et multi-cœur, et perspectives d'adaptation dynamique qui ouvrent sur les possibles pistes à suivre pour la thèse.

Table des matières

1 Introduction et Problématique

1.1 Le Contexte Automobile

Depuis près de 30 ans, l'industrie automobile n'a cessé de faire évoluer la façon de concevoir les véhicules et notamment leurs systèmes sous-jacents. Les voitures se modernisent de plus en plus avec l'ajout de calculateurs dédiés à des fonctions internes ou des services. Ainsi le développement des technologies de l'industrie 4.0 mène à une augmentation exponentielle du logiciel embarqué dans l'automobile au cours des 15 dernières années [12], avec la présence de plus de 60 calculateurs embarqués dans certains modèles. Les contrôles mécaniques et autres systèmes électriques cèdent la place au monde du numérique. Les équipements électroniques et logiciels se multiplient au sein du véhicule pour l'aide à la conduite et l'ajout de services [49].

Les systèmes automobiles sont ainsi devenus des systèmes cyber-physiques qui entrent en interaction à la fois avec les utilisateurs et l'environnement. On distingue deux grands domaines de logiciels embarqués dans le véhicule. Tout d'abord l'*infotainment*, qui réunit les systèmes multimédias et autres affichages non nécessaires à l'usage primaire du véhicule. Et deuxièmement les calculateurs enfouis qui réalisent des fonctions essentielles qui ne sont pas nécessairement visibles de l'utilisateur, telles que le contrôle moteur. De plus les véhicules sont de plus en plus connectés. On parle de communications *car-to-car* ou *car-to-infrastructure*, qui ouvrent la voie aux systèmes automobiles distribués. Cette ouverture du système à son environnement est à double tranchant. D'une part cela offre de nouveaux horizons, avec des possibilités d'évolutivité simplifiée, mais d'autre part cela pose des problèmes de sécurité-immunité, et complexifie la prédictibilité comme on le verra dans cet état de l'art. Cela fait évoluer les systèmes embarqués dans un environnement profondément à risques. Au vu des fonctions réalisées, cela donne un poids prépondérant aux systèmes embarqués, qui se doivent de respecter des standards de sûreté élevés pour garantir leur fonctionnement. Cela va notamment se traduire par l'introduction des mécanismes de sûreté de fonctionnement, et par la prise en compte de contraintes temps-réel dans le développement de fonctions critiques.

Cette complexité croissante des systèmes embarqués dans l'automobile se retrouve dans d'autres domaines, pour lesquels des architectures logicielles spécifiques ont été mises en place. On pourra citer ARINC 653 pour l'aéronautique ou AUTOSAR pour l'automobile en ce qui concerne les architectures logicielles, qui nous intéresse plus particulièrement. Ces architectures ont pour objectif premier d'aider à gérer la complexité et faciliter le développement du logiciel d'un point de vue compatibilité. Ce point sera détaillé dans une seconde partie. Historiquement, les calculateurs embarqués étaient conçus de manière *ad hoc* : le hardware et le software étaient intimement liés. Cela a conduit à un nombre de calculateurs très important, chaque calcula-

teur apportant sa fonctionnalité. On a donc actuellement une architecture avec un grand nombre d'ECU (Electronic Control Unit). On peut noter trois principales propriétés des systèmes embarqués automobiles :

— Les ECU sont *inter-connectés*, ils communiquent les uns avec les autres,

— Les fonctions et services sont intégrés dans des sous-systèmes complexes. Ainsi un sous-système inclut divers fonctionnalités,

— Les fonctions sont *distribuées* sur plusieurs calculateurs. Certaines fonctions peuvent être hébergées par plusieurs micro-contrôleurs.

Ce type d'architecture présentée ci-dessus a des inconvénients évidents en terme d'évolutivité du système et de temps de développement. A chaque changement de support physique (micro-contrôleur) le logiciel doit passer par un nouveau stade de développement plus ou moins conséquent. Inversement, une mise à jour du logiciel va demander une prise en compte du hardware. Cela augmente donc les temps et les coûts de développement. L'évolution future est de réduire le nombre de calculateurs embarqué, en passant d'un grand nombre d'ECU à un nombre bien plus réduit de "super ECU", qui vont multiplexer différentes tâches. On passe de cette façon d'un système distribué à un système multi-cœur. Là où ce type d'architecture va permettre de faciliter l'évolutivité, réduire les coûts et l'encombrement, vont apparaître d'autres problématiques auxquelles nous allons nous intéresser. Sur le plan de la sûreté, le processus de développement occupe une place importante dans l'industrie et est guidée par des standards, DO178C pour l'aéronautique et ISO 26262 pour l'automobile.

1.2 Problématique

Avec les architectures automobiles telles qu'elles évoluent actuellement se dessine un certain nombre de problématiques qui prennent de l'ampleur. Comme cela a été mentionné, les systèmes embarqués dans l'automobile sont soumis à des contraintes de sûreté fortes. Typiquement, des fonctionnalités qui étaient auparavant totalement décorrélées peuvent désormais se retrouver sur le même calculateur et interférer les unes avec les autres. Le multiplexage d'applications sur un calculateur puissant permet de réduire le nombre de micro-contrôleurs et de fait l'architecture matérielle. Dans ce contexte, similaire à celui de l'avionique, il faut mettre en place des mécanismes pour la conception, la vérification, la validation et la maintenance du logiciel. Que ce soit dans l'automobile ou l'avionique, on retrouve des similitudes sur les moyens employés.

Tout d'abord d'un point de vue des mécanismes de sûreté de fonctionnement, on distingue plusieurs catégories de fonctions logicielles, par niveau de criticité, pour un total de 5 niveaux de criticité allant d'un risque "catastrophique" pour les fonctions les plus vitales à "aucune conséquence"

pour les fonctions les moins importantes. Pour l'automobile, il s'agit des niveaux d'ASIL (Automotive Safety Integrity Level) allant respectivement de l'ASIL D à l'ASIL A, et QM pour les fonctions sans exigence particulière. Selon ce niveau de criticité, les mécanismes mis en œuvre seront plus ou moins importants tout au long du processus de développement. Afin de garantir l'intégrité du système avec la coexistence de tâches à différents niveaux de criticité, un des grands principes est la séparation spatiale et temporelle des tâches au sein du calculateur (*Time and Space Partitioning - TSP*). Spatiale car une tâche donnée aura son propre espace mémoire réservé, afin d'éviter toute interaction indésirable avec d'autres tâches. Temporelle afin de garantir que chaque tâche dispose de son temps d'exécution propre afin de garantir son bon fonctionnement. C'est suivant ce principe que dans le domaine de l'Avionique, le choix a été fait de cloisonner les tâches par l'utilisation du partitionnement. Ce partitionnement fixe va allouer à chaque tâche un espace mémoire prédéfini ainsi que des cœurs sur lesquels s'exécuter. Et pour les aspects temporels, il s'agit d'un ordonnancement *statique*. Dans le cadre des systèmes automobiles, cette solution n'est pas satisfaisante en raison de contraintes budgétaires fortes. Les avancées sur la connexion de la voiture au monde ouvert (notamment pour permettre des mises-à-jour *Over-The-Air*) nous impose presque naturellement de rechercher des stratégies de partitionnement *dynamique*. De cette façon, il sera possible de faire évoluer le logiciel embarqué en réduisant au maximum les coûts.

Cependant, cette solution présente de nombreux défis pour être viable. De fait il y a un équilibre à trouver entre dynamicité et prédictibilité des systèmes embarqués, pour permettre leur exécution dans des conditions sûres de fonctionnement et notamment dans le respect des contraintes temporelles. C'est dans ce contexte là que nous allons nous efforcer à identifier les différents mécanismes pour permettre une utilisation dynamique appropriée de calculateurs multi-cœurs dans l'automobile. On verra dans un premier temps les architectures et standards automobiles actuels dans lesquels vont s'inscrire la thèse. On étudiera dans un deuxième temps les différents travaux qui ont été menés sur les aspects temporels et l'ordonnancement des systèmes critiques. Dans un dernier temps nous présenterons les travaux relatifs à l'exploitation des multi-cœurs et les stratégies d'adaptation dynamique de l'architecture.

2 Architectures E/E et Logicielles Actuelles

2.1 Architectures Matérielles

2.1.1 Plateforme Physique Automobile

Historiquement, les deux grands domaines de systèmes embarqués dans l'automobile (systèmes enfouis d'une part et infotainment d'autre part)

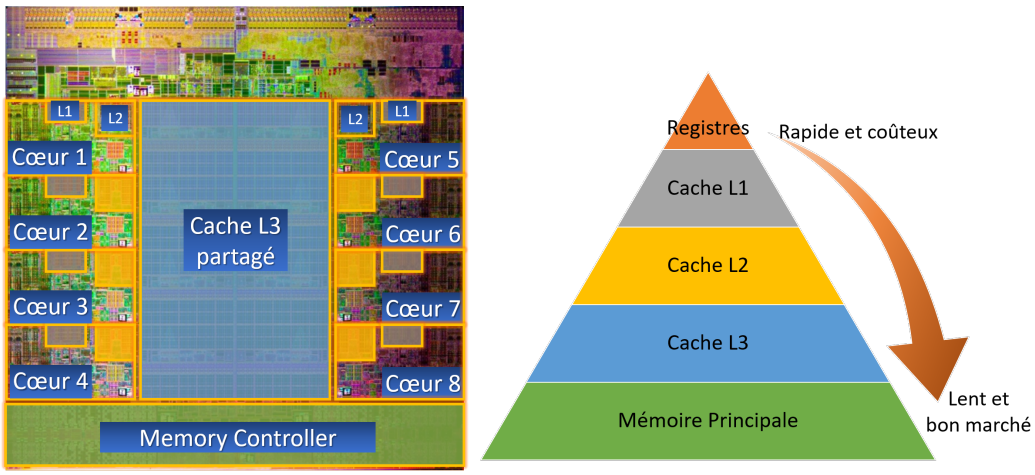
étaient totalement séparés. D'une part on avait donc des systèmes hautement critiques soumis tout le long de leur développement à des contraintes strictes pour la sûreté de fonctionnement, le respect des contraintes temps-réel et l'intégration. D'autre part, du côté infotainment les contraintes sont moins strictes, nous sommes face à des systèmes à temps-réel dit mou (*Soft real-time systems*, à l'opposé des *hard real-time systems*), les défaillances ayant des conséquences moindres, ou tout du moins non critiques. Cette séparation logicielle a mené logiquement en une séparation de l'architecture matérielle. Cependant, cette séparation pose des problèmes d'encombrement, de coût, de consommation thermique et électrique qui deviennent non négligeables. C'est ce qui mène à une évolution de l'architecture matérielle.

Avec l'apparition de calculateurs de plus en plus puissants, cette architecturé opérationnelle fédérée se modifie pour devenir une architecture opérationnelle intégrée. A présent, les composants logiciels commencent à être regroupés au sein du même ECU. Cela permet de diminuer les coûts et la maintenance, de par la réduction du nombre d'ECU. Ce regroupement d'applications impose cependant des standardisations communes, ce qui nous mène au standard AUTOSAR décrit précédemment. En diminuant le nombre de calculateurs physiques, on constate cependant une perte en terme de sûreté de fonctionnement, de par la perte du cloisonnement naturel des tâches qui était effectué auparavant. À présent cette séparation physique est abolie et il est nécessaire de complexifier la partie logicielle de façon à compenser ce manque. Ce qui nous mène à la norme ISO 26262 qui s'intègre dans le processus de développement.

2.1.2 Calculateurs modernes

Avec l'évolution des besoins de calculs au sein des systèmes cyber-physiques, les unités de calcul classiques utilisées jusqu'alors, à base de micro-processeurs dits "simples", n'est plus suffisant. Cela mène à l'exploitation de processeurs dits "modernes", qui offrent des fonctionnalités plus avancées. Nous présenterons ici succinctement 4 types de micro-processeurs, ainsi que leurs caractéristiques principales et l'intérêt que l'on peut leur trouver.

Calculateurs Multi-cœurs La première évolution des microprocesseurs a été de chercher à augmenter les capacités de parallélisme de ces derniers, pour améliorer leur vitesse de calcul et donc leur efficacité. La notion de calculateur multi-cœur est apparue dès les années 1950 [52] pour nous amener aux architectures physiques actuelles. Le principe est de disposer d'un plus grand nombre d'unités de calculs (dit cœurs) qui pourront donc exécuter des instructions en parallèle. Dans le cas d'un parallélisation au niveau circuit (*chip-level multiprocessing - CMP*), plusieurs coeurs sont intégrés au sein d'un même boîtier. La mémoire locale est alors partagée à différents degrés entre les coeurs. De façon à décongestionner les accès mémoires et



(a) Répartition des Caches sur un processeur Intel

(b) Hiérarchie des espaces mémoire

FIGURE 1 – Hiérarchie Mémoire appliquée à un processeur Intel I7-36xx

accélérer ces dernières, une hiérarchie mémoire est mise en place, associant des espaces mémoire progressivement plus petits et rapides en fonction de leur proximité au processeur. Il s'agit ici de trouver un équilibre entre coût de la mémoire et vitesse d'accès aux données. En effet, la mémoire dispose de trois caractéristiques antagonistes : la **latence** (temps d'accès aux données), la **bande passante** (volume de données accessible simultanément) et la **taille** mémoire (quantité de donnée stockée pour un coût donné). Un espace mémoire pourra donc soit être de petite taille mais rapide au niveau de son temps d'accès, soit de grande taille et plus lent. On a par conséquent au plus proche des coeurs les Registres, de taille très limitée mais au temps d'accès très rapide. À l'opposé, la mémoire principale, de plus grande taille pour laquelle tous les coeurs doivent passer par un bus commun pour y accéder. C'est donc la mémoire la plus lente d'accès mais aussi la moins chère. Plusieurs intermédiaire ont été mis en place entre ces deux types de mémoire. Il s'agit de trois niveaux de cache L1, L2 et L3. Les caches L1 et L2 sont dits non partagés, c'est-à-dire propres à chaque cœur. Le cache L3 est partagé entre les coeurs, il fait partie des caches LLC (*Last Level Cache*). Le contenu de ces caches est régulé empiriquement par principe de localité temporelle [17] et spatiale [57]. On considère que plus une donnée a été récemment accédée, plus elle a de chance d'être à nouveau utilisée. De même si une donnée est utilisée, alors les données proches spatialement ont aussi plus de chance d'être utilisées. Un exemple de processeur multi-cœoeur avec hiérarchie mémoire est indiqué en figure 1.

Calculateurs Many-cœurs Les calculateurs many-cœurs ont cela de différent vis-à-vis des multi-cœurs que les unités de calcul ne sont pas reliées

entre elles de la même façon. Là où les microprocesseurs multi-cœurs ont un bus de données commun entre chaque coeur (hors communication avec les caches non partagés), une architecture many-cœur dispose d'un maillage sous forme matricielle entre les coeurs. De cette façon, un groupe de cœurs peut s'occuper d'une tâche et se partager des données sans interférer d'aucune façon avec un autre groupe de cœur, du fait que les données ne transitent pas sur un bus commun. Ce type d'architecture demande un travail particulier pour exploiter le réseau de coeurs, mais offre alors de très bonnes performances lorsqu'il s'agit d'effectuer un grand nombre de tâches indépendantes et plus généralement pour tout logiciel hautement parallélisable. Seiler présente un exemple d'architecture many-cœur en [51].

GPU Les Processeurs Graphiques (*Graphic Processing Unit* - GPU), d'abord dédiés aux calculs graphiques 3D et peu utilisés hors d'un usage spécifique, sont à présent très répandus pour des usages variés [43]. Cette forte expansion des GPU est due non seulement aux capacités de rendu graphique, mais surtout à leurs capacités de forte programmation parallèle. Un grand nombre de domaines, notamment dans la recherche, y voient donc un micro-processeur d'usage général à hautes capacités de calcul programmable. Les GPU sont efficaces du fait qu'ils permettent de réaliser le même calcul sur un très grand nombre de données différentes pour obtenir tout autant de résultats en sortie. Il s'agit d'un modèle dit *SPMD - Single-program, multiple-data*.

FPGA Contrairement aux processeurs tels que présentés ci-dessus, programmer un FPGA (*Field-Programmable Gate Array*) consiste à redéfinir le circuit intégré en lui-même, et non à y adosser un code qui sera exécuté. C'est cette définition du circuit intégré qui permet d'implémenter les fonctionnalités souhaitées. Les FPGA incluent de nombreux avantages : les temps d'accès aux entrée/sorties sont plus rapides, ils permettent la mise en oeuvre de fonctions personnalisées qui seront optimisées, et ce de façon fiable. Enfin, ils restent reconfigurables et permettent du prototypage sans repasser par le processus de fabrication tel que pour les ASIC (*Application-Specific Integrated Circuits*). Cependant, l'utilisation de FPGA impose un niveau de complexité d'implémentation supplémentaire vis-à-vis d'un microprocesseur classique.

2.2 Processus de Développement

L'intégration des systèmes électriques et électroniques (E/E) soulève comme dit précédemment la problème de la coexistence de fonctions ou services de différents niveaux de criticité. De plus plusieurs acteurs sont impliqués lors de la conception d'une voiture : le Constructeur Automobile

(*Original Equipment Manufacturer - OEM*), divers sous-traitants (dénommés *Tier 1*, *Tier 2*...) qui développent les produits pour le système défini par l'OEM. Chaque entité a ses propres équipes et méthodes de développement. Il faut donc définir des règles de conception robustes pour permettre la documentation et l'intégration de toutes les parties du système.

Notons qu'il n'existe aucune régulation stricte ni directive sur la sûreté de fonctionnement dans l'automobile. Il n'y a pas d'exigence légale pour la certification des systèmes E/E. Cependant les principaux acteurs du domaine ont décidé d'adhérer volontairement à un état de l'art ayant donné lieu au standard IEC 61508 [25], qui n'est pas réservé au seul domaine de l'automobile. Il propose une approche générique pour les systèmes embarqués. Le standard IEC 61508 se focalise sur les processus de développement du système afin d'assurer la sûreté de fonctionnement. C'est de ce standard qu'a été dérivé la version courante ISO 26262 utilisée dans l'automobile.

2.2.1 Attributs de la Sûreté de Fonctionnement

La Sûreté de Fonctionnement (*Dependability*) rassemble six attributs [4] qui permettent de caractériser la qualité du service fourni :

- Disponibilité (*Availability*) - la capacité d'un service à se rendre opérationnel pour s'exécuter ;
- Fiabilité (*Reliability*) - la capacité à effectuer un service sans erreur de façon continue ;
- Sûreté (*Safety*) - l'absence de risques catastrophiques pour l'utilisateur ou l'environnement ;
- Confidentialité (*Confidentiality*) - le non partage de données sans autorisation ;
- Intégrité (*Integrity*) - l'absence d'altération indésirable du système
- Maintenabilité (*Maintainability*) - la capacité de gérer des modifications et réparations du système.

Selon le contexte industriel, chaque attribut se verra considéré de manière différente, et aura une signification plus précise. Ce choix dépend directement des objectifs que doit réaliser le service. Par exemple dans le cas des transports, la Fiabilité et la Sûreté sont primordiales, tandis que dans les systèmes de communication, les attributs clés sont la Disponibilité, la Fiabilité et la Confidentialité.
Dans le cadre de cette étude, nous nous concentrerons principalement sur les aspects liés à la *Safety*.

2.2.2 Safety et Niveaux de Criticité

Pour un système donné, la Safety va se traduire par une discrimination des services selon une échelle de criticité. Cela permet d'identifier et dis-

tinguer les parties du système les plus critiques, pour lesquelles il faudra fournir le plus d'efforts, que ce soit en termes de développement, de vérification et d'intégration, de celles qui sont le moins critiques, voire qui n'ont aucune incidence sur la Safety et donc demanderont un moindre effort de conception sur ces aspects. Cette échelle est définie différemment selon les domaines, mais certaines similarités se dégagent. Un modèle simple est fondé bien souvent sur deux niveaux de criticité : Haut ou Bas. Par simplification, un grand nombre d'études sont réalisées suivant cette hypothèse de niveau de criticité binaire, comme on le verra sur les aspects temps-réel ci-après. Un autre modèle classique repose sur 5 niveaux de criticité. C'est notamment le cas dans l'automobile tel que défini dans le standard ISO 26262 ou encore dans l'avionique avec le DO178C.

Le standard ISO 26262 [27] est la référence pour la sécurité-innocuité dans le domaine de l'automobile. Elle est issue d'une collaboration entre les acteurs majeurs du secteur. L'objectif d'ISO 26262 est la sûreté fonctionnelle (*functionnal safety*). Dans le cadre du domaine automobile, cela se traduit par l'absence de risques déraisonnables en raison de dangers causés par des dysfonctionnements de systèmes E/E. Les aspects de sûreté non-fonctionnelle sont hors du cadre de la norme, par exemple un dysfonctionnement dû à un départ de feu ou tout autre évènement externe aux systèmes E/E.

ISO 26262 introduit le concept d'ASIL (*Automotive Safety Integrity Level*) pour la définition des niveaux de criticité. On retrouve quatre niveaux, allant de l'ASIL A pour le niveau le moins critique à l'ASIL D pour le plus critique. Il existe aussi un cinquième niveau noté QM (*Quality Management*), qui définit une absence d'exigences spécifiques de criticité et donc aucune prise en compte nécessaire du standard ISO 26262 dans le développement. Pour la détermination de ces niveaux, trois critères sont pris en compte : la sévérité, la probabilité d'accomplissement et la contrôlabilité.

La Sévérité est basée sur le degré de conséquence en cas de défaillance. Elle peut être Légère et Modérée (*S1*), Sévère et potentiellement mortelle -mais survie probable- (*S2*), Potentiellement mortelle -survie incertaine- voire mortelle (*S3*).

La Probabilité d'accomplissement indique le taux d'occurrence potentiel. Elle peut être Très Faible (*E1*), Faible (*E2*), de probabilité Moyenne (*E3*) ou de Haute probabilité (*E4*).

La Contrôlabilité est un concept plus subjectif basé sur les capacités de l'utilisateur à gérer la défaillance. Une défaillance peut être facilement contrôlable (*C1*), normalement contrôlable (*C2*), difficilement voire impossible à contrôler (*C3*).

Bien entendu, l'interprétation de ces trois critères doit se faire de façon relative au système étudié, et non de façon absolue et déterministe. Avec ces trois critères subdivisés sur une échelle de valeurs, il est possible pour toute défaillance potentielle des éléments du système de quantifier son niveau de criticité et donc le niveau de confiance que l'on va imposer à ce composant

Severity of the harm	Probability of exposure	Controllability		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	ASIL A
	E4	QM	ASIL A	ASIL B
S2	E1	QM	QM	QM
	E2	QM	QM	ASIL A
	E3	QM	ASIL A	ASIL B
	E4	ASIL A	ASIL B	ASIL C
S3	E1	QM	QM	ASIL A
	E2	QM	ASIL A	ASIL B
	E3	ASIL A	ASIL B	ASIL C
	E4	ASIL B	ASIL C	ASIL D

FIGURE 2 – Matrice de Définition des Niveaux d'ASIL - ISO 26262 (2011)

pour qu'il puisse être utilisé de façon sûre de fonctionnement. Plus un composant sera critique, plus son niveau d'ASIL sera élevé en conséquence, et donc plus il faudra apporter d'efforts pour qu'il respecte les standards. Cette discrimination de criticité est schématisée dans la figure 2.

Par la suite, le standard recommande en fonction du niveau d'ASIL des éléments les différents types de mécanismes qui peuvent être mis en place. Il y a trois sujets principaux de préoccupation : les flots de données, les flots de contrôle et l'architecture.

Plus généralement, ISO 26262 se base sur le cycle de conception en V pour définir à chaque niveau de conception pour les systèmes électriques et électroniques (E/E) les processus à suivre afin de garantir la qualité du développement. A chaque étape du processus, elle décrit le cadre d'application, les activités et les méthodes à utiliser afin d'obtenir un système sûr de fonctionnement. On retiendra en particulier, les parties 4 à 6 indiquées dans la figure 3, qui décrivent respectivement le développement d'un produit au niveau "système", "matériel" et "logiciel". Elle offre une approche reposant sur l'analyse de risque afin de déterminer pour chaque élément un niveau de criticité appelé ASIL comme dit précédemment. En fonction de l'ASIL de chacun des éléments, différentes approches peuvent être adoptées pour mettre en place des mécanismes de sûreté adaptés. Les questions de validation pour les systèmes E/E sont également mis en avant dans la norme. Enfin, les questions de relation entre les constructeurs et les fournisseurs sont détaillées.

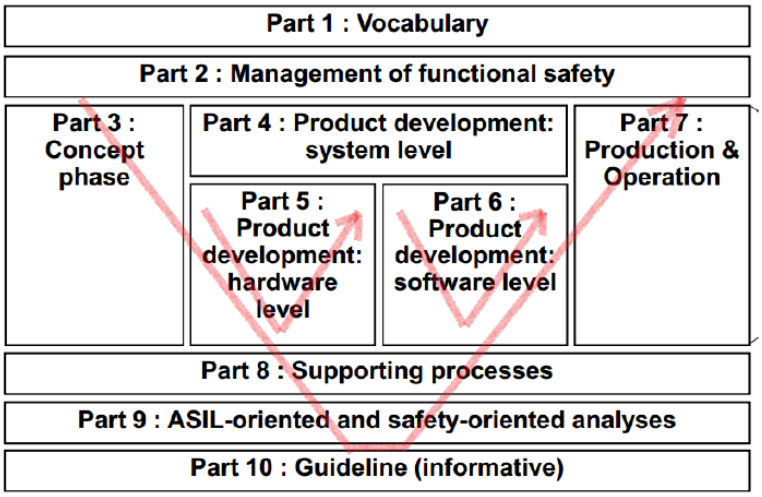


FIGURE 3 – Organisation d'ISO 26262, avec les Cycles en V

2.3 Architectures Logicielles

Afin de palier à la complexité grandissante du logiciel dans les systèmes embarqués, les approches par modèles, dit "Model-Based Design" sont préférées. Ce type d'approche permet un développement facilité et accéléré de systèmes complexes, par le biais d'outils spécialisés. Cela permet de répondre aux besoins de développement pour les systèmes embarqués, simplifie les tests et l'évolutivité. Ainsi dans l'avionique les Bancs d'Intégration Simulés Virtuellement (*Virtual Simulated Integration Bench* - VSIB) sont utilisés afin de faciliter le test, l'intégration et la démonstration des fonctionnalités avioniques, tout au long du processus de développement, de la phase de spécifications jusqu'à la phase d'intégration, comme décrit dans [50]. Aussi, une autre approche "Model-Based" dans l'aéronautique est donnée dans [26]. Ce travail explique comment cette approche permet de réduire les coûts et les risques au cours du développement, dans le cadre de la conception du système de gestion du carburant.

2.3.1 Architectures Automobiles - AUTOSAR

Classic AUTOSAR Le standard AUTOSAR [56] (pour *Automotive Open System Architecture*) a pour objectif de faciliter la réutilisation du logiciel ainsi que sa modification, sa maintenance ou son intégration dans le contexte automobile. Depuis la version 4 de AUTOSAR, un certain nombre de mécanismes devant être présents dans l'architecture embarquée ont été spécifiés. Il s'agit de mécanismes qui vont permettre d'implémenter plus facilement au sein des architectures AUTOSAR les concepts définis par la norme ISO 26262. AUTOSAR présente une architecture en couche qui permet d'abs-

traire la partie physique. Cette architecture modulaire offre une interface pour standardiser la conception du logiciel. Elle tente de cette façon de s'approcher d'un développement logiciel par Modèle, comme présenté précédemment, afin de notamment réduire les coûts de développement.

AUTOSAR permet une approche orientée application qui propose une architecture en 3 couches.

— La couche Applicative - elle inclut les composants logiciels *Software Components - SW-C*,

— Le Basic Software (BSW) - couche basse qui contient tous les services nécessaires à l'exploitation de la plateforme. Il est constitué de 3 sous-couches (Service, Abstraction ECU & Microcontroller) qui se décomposent en 5 stacks (Service, Mémoire, Communication, Abstraction des I/O hardware, Complex Devices Drivers), chaque stack étant intercouches. Il contient ainsi le système d'exploitation, les pilotes relatifs aux périphériques de l'ECU, les services logiciels et de communication.

— Le Run-Time Environment (RTE) - code généré pour permettre la communication entre les SW-C et les services du BSW. Cette couche cruciale permet l'abstraction de la partie matérielle et donc rend la conception des SW-C indépendantes de cette dernière.

Un des aspects important pour notre étude est l'ordonnanceur inclus dans le BSW. Il s'agit d'un ordonnancement basé sur les priorités (*priority-based scheduling*) appelé AUTOSAR-OS, chaque tâche étant composée d'exécutables (*runnables*) qui appartiennent aux différents SW-C applicatifs. Chaque exécutable peut être périodique ou apériodique. Ils sont connectés via le RTE pour la communication des données. Les exécutables sont regroupés en tâche selon leur caractéristiques (périodes, entrées...). Dans la pratique, AUTOSAR-OS est dérivé du kernel OSEK/VDX pour permettre l'ordonnancement des tâches et les Routines de services d'interruptions (*Interrupt Service Routines - ISRs*). L'architecture globale est représentée en figure 4.

On note avec ce standard une absence de séparation physique entre différents composants, que l'on pouvait retrouver auparavant. C'est de par ces problématiques de complexité croissante de l'intégration du logiciel et la sûreté de fonctionnement que la norme ISO 26262 devient complémentaire au standard AUTOSAR pour garantir les contraintes non-fonctionnelles imposées aux applications temps-réel dur implémentées.

Adaptive AUTOSAR Avec l'ouverture au monde des objets connectés et l'explosion des fonctionnalités pour tendre vers une voiture autonome de nouvelles contraintes s'appliquent à l'architecture logicielle. Cette dernière doit inclure des puissances de calculs plus importantes pour traiter les données, la possibilité de mises à jour à distance *Over-The-Air - (OTA updates)*, ou encore le déploiement dynamique de nouvelles fonctionnalités. Il se trouve

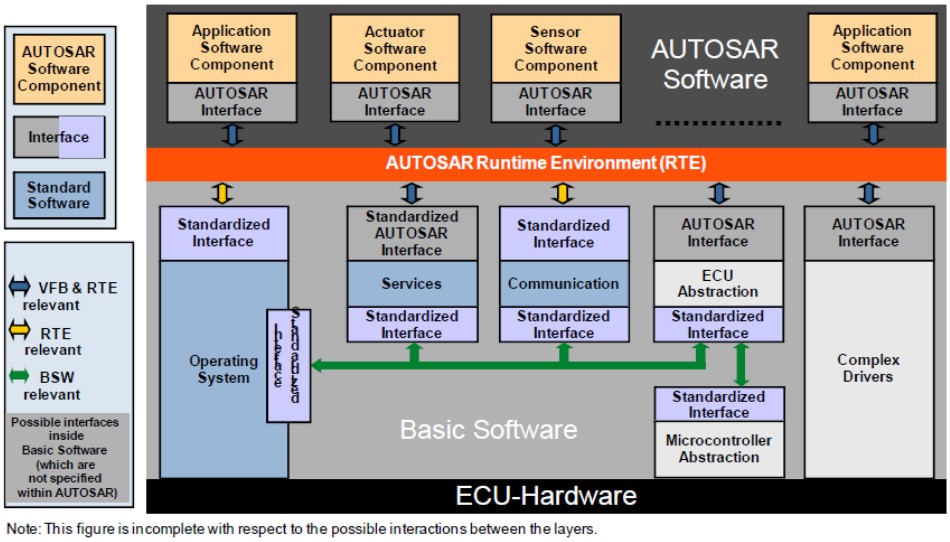


FIGURE 4 – Architecture en Couches d'AUTOSAR

que les architectures actuelles, notamment d'AUTOSAR, ne permettent pas de supporter ces nouvelles contraintes. La version classique d'AUTOSAR est statique et ne permet pas d'ajouter des fonctionnalités. L'analyse de la flexibilité d'AUTOSAR a été étudiée en détail dans la thèse de H. Martorell [39]. C'est pour ces raisons qu'une version Adaptive d'AUTOSAR [20] est à l'étude, pour venir compléter le standard actuel en répondant aux nouveaux besoins. L'objectif est donc de proposer une nouvelle architecture logicielle avec un système d'exploitation dédié à des hautes performances de calcul, accompagné des propriétés de Sûreté de Fonctionnement qui sont nécessaires à l'exécution de ces fonctions qui seront critiques pour la voiture autonome, mais aussi et surtout une plus grande flexibilité.

Du point de vue de la communication de données, le principe initial d'envoi en broadcast des données sur le bus partagé reste inchangé. Cela reste tout à fait pertinent pour les données de commande, qui sont de taille limitée et communiquées périodiquement. Mais l'automatisation importante des fonctions du véhicule implique un transfert de données bien plus important, et surtout en quantité non contrôlée car dépendante des capteurs et de l'environnement extérieur. Adaptive Autosar cherche donc à s'orienter vers des protocoles de communication orientés service, avec des données publiées et les applications concernées y souscriraient. C'est le principe du publish-Subscribe [18] que l'on retrouve dans ROS [47]. La flexibilité de ROS a été démontrée récemment dans les travaux en relation avec l'automobile [35].

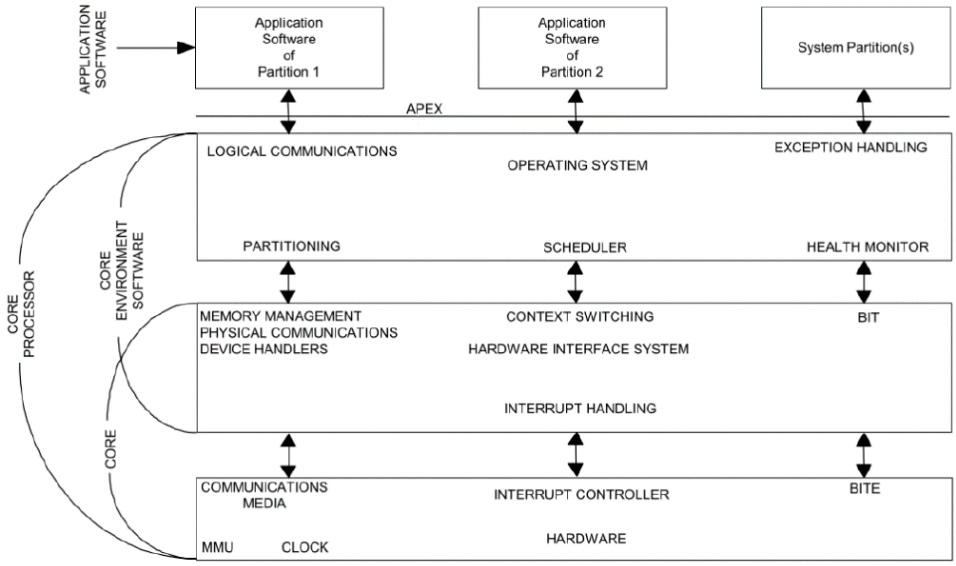


FIGURE 5 – Architecture logicielle définie pour ARINC 653 - AEEC 2003

2.3.2 Architectures Avioniques - ARINC 653

Dans le domaine de l'avionique, les architectures logicielles reposent sur un principe modulaire appelé Avionique Modulaire Intégrée (*Integrated Modular Avionics - IMA*). Cette architecture repose sur le standard ARINC 653 [3] et propose le partage des ressources de calcul ainsi que des communications entre les fonctions.

Comme représenté dans la figure 5 ci-dessous, ARINC 653 décompose l'architecture en 3 niveaux. D'abord l'interface du système avec le hardware, qui inclut les systèmes de gestion mémoire, des communications physiques et autres gestions de couche basse. La couche intermédiaire dispose des différents services tel que la gestion des exceptions, les moniteurs, l'ordonnanceur et tout ce qui a trait à l'OS embarqué. Enfin la couche haute comprend les différentes partitions, chacune incluant le logiciel d'une Application. Ce standard définit l'interface normalisée du système d'exploitation temps-réel pour les logiciels avioniques. Il permet le développement de chaque composant par différents constructeurs en facilitant leur intégration. La principale caractéristique de l'architecture logicielle telle que définit dans ARINC 653 est le partitionnement de chaque application qui partage la même plateforme matérielle. Chaque partition permet alors une affectation à la fois physique et temporelle (temps de calcul) dédiée. Du point de vue de la gestion de la sûreté de fonctionnement, le domaine aéronautique se base lui aussi sur 5 niveaux de criticité, *Developement Assurance Level - DAL* [24], chacun lié à un taux de défaillance ciblé. Le DAL A étant le niveau le plus fort avec couverture du critère *fail-safe* et un taux de défaillance inférieur à 10^{-9}/a. A l'inverse

le DAL E est le moins restrictif, pour des fonctions qui n'interviennent pas dans la sûreté de fonctionnement du système (c.f. DO178C).

Le partitionnement comme présenté dans les systèmes avioniques par le biais de l'ARINC 653 [46] présente plusieurs avantages vis-à-vis des méthodes actuelles dans l'automobile. Il s'agit là d'un standard qui uniformise l'interface avec l'OS Temps-Réel pour les applications réparties dans des partitions. Chaque partition contient un ou plusieurs processus concurrents qui se partagent l'accès aux ressources processeur. Parmi les raisons qui amènent à nous intéresser à ce partitionnement on peut citer les points suivants :

— Efficacité dans l'utilisation des ressources de calcul. La mutualisation des ressources en facilite la maintenance, offre un meilleur compromis entre le poids, la consommation énergétique et l'encombrement,

— Utilisation de composants sur étagères pour réduire les coûts,

— Intégration de briques logicielles provenant de fournisseurs multiples, et portabilité de ces application sur toutes les plateformes compatibles ARINC 653 grâce à l'indépendance du logiciel applicatif avec les couches logicielles bas niveau.

— Support de la multi-criticité. L'intégration de fonctionnalités utilisateur à différents niveaux de criticité (par exemple dans l'automobile, aide à la conduite, logiciels de divertissement, partie contrôle du véhicule...) ouvre sur les problématiques liées aux systèmes multi-critiques.

Cependant, une architecture partitionnée statiquement, comme implémentée actuellement dans le standard ARINC 653, implique une sous-utilisation des ressources, due essentiellement à la sur-réservation effectuée pour la partition critique. Quelques études préliminaires récentes ont ciblé l'adaptabilité dans les systèmes partitionnés avioniques [16] en se basant sur les différentes phases d'une mission. Ce système d'ordonnancement basé sur les modes (mode-based schedules) apporte à l'ARINC 653 une architecture à deux niveaux hiérarchiques d'ordonnancement temps-réel : un premier ordonnancement entre les différentes partitions, gère le temps de calcul alloué à chacune d'entre elle. Un second niveau d'ordonnancement inférieur s'applique sur les processus au sein de chaque partition. Chaque partition possède pour cela son propre processus d'ordonnancement. Ces ordonnancements peuvent alors être réglés selon des stratégies différentes, selon les besoins fonctionnels du système. Ces besoins sont définis suivant 3 phases de fonctionnement : vol, approche et exploration. Ces différents modes d'ordonnancement doivent prendre en compte les exigences temps réel. Il est possible que durant l'exécution du système des processus dépassent leur deadline. Cela peut être dû à des fautes transitoires, une sous estimation du WCET... Cependant ces problèmes peuvent être évités avec l'utilisation d'outils hors-ligne, utilisés avant la mise en fonctionnement, pour vérifier le respect des contraintes temps-réel.

3 Temps-réel et Criticité

3.1 Définition et Problématique

Les systèmes à criticité mixte (Mixed Criticality Systems - MCS) ont mené à de nombreuses études pour gérer la répartition temporelle des tâches, et en même temps prendre en compte les conditions de partitionnement et le partage des ressources de façon à optimiser l'usage de ces dernières. La plupart des travaux dans ce domaine se basent sur le modèle d'un ensemble de tâches τ_i qui possèdent les caractéristiques suivantes :

— T_i - la période d'apparition de la tâche
— D_i - la date limite d'exécution (*deadline*)
— C_i - Le Pire Temps d'Exécution (*Worst Case Execution Time - WCET*)
— L_i - Le Niveau de Criticité de la tâche

Ces calculateurs hébergeant plusieurs applications demandent un soin particulier au cloisonnement des tâches de façon à éviter qu'une faute dans un composant logiciel ne vienne se répercuter sur un autre. Cela est d'autant plus vrai quand les applications hébergées ont des niveaux de criticité différents. En effet, sans ce type de protection chaque composant devrait être du niveau de criticité le plus élevé, ce qui augmente significativement les coûts de développement. A la suite de cette problématique de partitionnement des tâches vient celui du partage des ressources de façon efficace. Ce point est simplifié du fait que les différentes caractéristiques présentées ci-dessus ne sont pas indépendantes. En particulier, l'estimation du WCET, C_i, est dérivé du niveau de crititicé. Plus le niveau de criticité sera fort, plus ce WCET sera choisi de façon fiable et avec des contraintes fortes, et donc plus il sera élevé comme l'observe Vestal [55].

Malgré le fait de disposer de plateformes matérielles déterministes pour lesquelles chaque tâche aurait un seul WCET déterminé, il n'est pas évident de connaître cette valeur de façon totalement certaine. En effet, cette incertitude est principalement épidémique et non due à des évènements aléatoires. Le système en lui-même étant au maximum protégé des aléas, c'est la non maîtrise exhaustive de ce que l'on sait ou non du système qui empêche d'obtenir des calculs exacts. Il reste raisonnable de dire que plus le WCET est grand, plus la fiabilité est bonne mais ce n'est pas une règle absolue. En effet, il n'a pas été identifié de relation évidente entre une augmentation de x% du WCET et l'amélioration de y% de sa fiabilité.

3.2 Allocation de Tâches

La problématique de l'allocation des tâches dans une architecture distribuée a été adressée par Tamas-Selicean et Pop dans [54] dans le contexte d'ordonnancements statiques (par exécutions périodiques) et partitionnement temporel. On parle ici principalement d'allocation temporel de tâches,

c'est à dire de leur répartition entre les partitions et calculateurs, au regard des temps de calcul (et donc par extension leur temps d'occupation du processeur) des tâches. Il a été observé que l'ordonnancement peut être parfois amélioré en augmentant le niveau de criticité de certaines tâches de façon à mieux équilibrer des partitions contenant un seul niveau de criticité. Cette augmentation de criticité implique bien entendu un coût et il faut donc trouver un compromis pour avoir une utilisation de ressources qui reste minimisée. Une autre étude a été réalisée par Kelly & al. [29] en considérant des processeurs partitionnés homogènes (chaque partition dispose du même nombre de cœurs, et de mêmes caractéristiques), pour comparer les approches *first-fit* (la première partition acceptable est sélectionnée) et *best-fit* (la plus petite partition acceptable est sélectionnée) avec un pré-classement des tâche soit selon leur niveau de criticité soit selon leur taux d'utilisation. La conclusion a été qu'en général une politique de *first-fit* est meilleure. Pour le cas d'architectures hétérogènes, Awan & al.[5] proposent un schéma de partitionnement pour optimiser la consommation énergétique, qui n'est pas notre critère prioritaire.

Un grand nombre d'approches sont possibles pour gérer l'allocation des tâches. Une bonne partie d'entre elles ont pu être évaluées par Rodriguez & al. [48]. Ils considèrent un ordonnancement par EDF avec le framework d'analyse proposé EDF-VD. EDF-VD a été présenté par Baruah & al. [11, 7] pour les systèmes à deux niveaux de criticité. Le principe étant de réduire les deadlines des tâches à faible criticité durant l'exécution en mode critique. En diminuant leur WCET, cela revient à augmenter virtuellement leur niveau de criticité. Toutes les deadlines étant réduites du même facteur. Cette base a pu être améliorée ensuite avec l'ajout de plusieurs facteurs de réduction des deadlines, et une méthode d'application en runtime. Rogriguez montre ici l'efficacité de différencier l'allocation des tâches selon la criticité. Les tâches les plus critiques étant allouées en *worst-fit* (la plus grande partition acceptable est sélectionnés) tandis que les tâches les moins critiques sont allouées en *first-fit*. Il est cependant relevé que dans le cas de l'existence de tâche à faible criticité mais occupant un grand espace (mémoire ou puissance de calcul), alors ces dernières devaient être allouées avant les tâches critiques.

Une approche différente vise à profiter au maximums des avantages du partitionnement en se basant sur les clusters. La plateforme multi-cœur est partitionnée de façon statique en un certain nombre de clusters. Et au sein de chaque cluster les tâches s'exécutent de façon "globale" (i.e. sans cloisonnement). Ali & Kim suggèrent [1] d'exploiter ainsi de petits clusters quand le système est dans un mode de fonctionnement normal, et d'agrandir le dimmensionnement des clusters en cas de changement vers un mode critique. Cette méthode tire à la fois profit des sécurités liés au partitionnement et de la flexibilité des clusters.

3.3 Analyses d'Ordonnancement

Les premières études sur la criticité mixte dans le cadre de l'utilisation de microprocesseurs ont été menées par Anderson & al. [2] en 2009 et développées en 2010 [40]. Ces études se basent sur un modèle de tâches à 5 niveaux de criticité possibles, allant de A à E, respectivement du niveau le plus élevé au plus faible. De plus, les charges des tâches sont considérées comme harmonique. Chaque cœur/processeur mono-cœur ayant un container par niveau de criticité ainsi qu'un ordonnanceur à deux niveaux de hiérarchie. Ces études exploitent une stratégie différente par niveau de criticité. Le niveau A utilise un ordonnancement statique par exécution cyclique des tâches, le niveau B utilise un partitionnement préemptif à l'échéance la plus proche (*Earliest Deadline First - EDF*), une préemption globale d'échéance la plus proche est utilisée pour les niveaux C et D et enfin approche "best-effort" pour le niveau E. Anderson & Al. ont pu démontrer la pertinence d'utiliser des stratégies d'ordonnancement différentes selon les niveaux de criticité. Ces travaux ont pu être repris par la suite, notamment pour évaluer les dépassements liés aux plateformes à plusieurs processeurs, à l'aide de techniques d'isolation de la mémoire (LLC et DRAM) [30].

3.3.1 Ordonnancements mono-processeur

Les premiers travaux sur l'ordonnancement multi-critique s'est principalement reposé sur les plateformes mono-processeur (e.g. [10, 14]), ayant pour conséquence de ne pas être directement applicable pour les plateformes multi-cœurs. Sur ces dernières, plusieurs approches existent, qui présupposent que le groupe de tâches est ordonnançable a minima pour le haut niveau de criticité. Dans [13], à la fois les systèmes temps-réel dur et temps-réel souple sont ordonnancés en utilisant la méthode de l'*Earliest Deadline First* (priorité aux tâches ayant la deadline la plus proche) en considérant que les tâches temps-réel critiques sont ordonnançables statiquement. Quand un cœur termine son exécution avant le *WCET* estimé, ce temps libre (dit *slack-time*) est ré-alloué aux tâches non critiques. L'ordonnancement multi-critique à deux niveaux de [2, 40] ordonnance les tâches de chaque niveau de criticité avec des approches différentes. Les tâches de niveau de criticité les plus faibles ne sont autorisées à être exécutées que s'il n'y a pas de tâches de criticité supérieure en cours d'exécution, i.e. les tâches critiques sont exécutées en isolation. Pour soutenir cette approche, l'outil LITMUS^{RT} met à disposition diverses politiques d'ordonnancement multi-critique [23]. Ainsi, cette méthode permet de garantir le temps d'exécution des tâches critiques tout en optimisant l'utilisation des ressources : le système s'exécute avec toutes les tâches en parallèle, et les propriétés temporelles des applications critiques sont évaluées régulièrement. Dès lors que le système de monitoring détecte une surcharge pouvant mener à un dépassement d'échéance, les tâches non

critiques sont suspendues, garantissant une exécution des tâches critiques dans leur délai maximum attendu.

Pour dépasser ce cadre simplifié, d'autres stratégies nécessitant une adaptation plus fine du système [36, 53] sont envisagées. Notamment, d'autres approches associent plusieurs WCETs à chaque tâche et considèrent alors un scénario d'ordonnancement par niveau de criticité [9, 22, 44]. En général on identifiera un WCET de haute criticité, qui est estimé le plus sûr possible et donc grand car pessimiste. Et un second WCET de criticité moindre (mais respecté dans la majeure partir des situations) est estimé, moins pessimiste et donc plus court.

Plus le niveau de criticité est haut, plus les WCETs seront allongés et prudents [15], au sens sûr de ne pas être dépassés. Initialement, toutes les tâches sont assignées par leur WCET de faible criticité. Durant l'exécution, ils observent si les tâches signalent leur fin d'exécution à l'instant prédéfini par leur WCET de faible criticité. Si aucun signal de terminaison n'a été reçu, le passage vers un WCET de plus haute criticité se fait et le système passe donc sur un ordonnancement de plus haut niveau de criticité en court d'exécution en modifiant les WCET pour les valeurs de haute criticité. Les tâches avec un plus faible niveau de criticité sont abandonnées [8] ou réallouées pour être ré-exécutées quand le système reviendra au mode de faible criticité [19].

Dans [6], les systèmes multi-critiques utilisent un ordonnancement déclenché temporellement où le WCET estimé peut être dépassé sans provoquer d'erreur critique au fonctionnement. Un moniteur détecte en cours d'exécution les dépassements, et change d'ordonnancement sur la base d'ordonnancements pré-calculés. Dans [21], un ordonnancement déclenché temporellement permet aux tâches de même niveau de criticité d'être exécutées en même temps, permettant une isolation temporelle des tâches.

3.3.2 Ordonnancements multi-processeurs et multi-cœurs

Dans [22] est décrit la généralisation de l'algorithme préemptif sur mono-processeur EDF avec Deadlines Virtuelles (*Earliest Deadline First-Virtual Deadlines - EDF-VD*), (c.f. 3.2), aux plateformes multi-processeurs. Pour des ordonnancements dit "globaux", ils exploitent un modèle "standard" de multiprocesseur, fpEDF, et le combinent avec l'approche sus-mentionnée EDF-VD. Les mesures indiquent que la combinaison est efficace. Des extensions de cette étude [9] mettent en perspective l'utilisation d'ordonnancements partitionnés vis-à-vis de d'ordonnancements globaux pour les systèmes à criticité mixte. Il en résulte que l'ordonnancement par partitionnement est bien plus efficace.

Malgré ce résultat, diverses études de partitionnements globaux, tel que celle de Pathan [44] qui dérive de l'ordonnancement à Priorités Fixes. Ici, les méthodes pour mono-processeur sont reprises, avec l'intégration d'une

analyse pour multi-processeur de façon à rechercher un ordonnancement optimal. L'approche arrive à démontrer son efficacité par la mesure d'un ratio de succès d'ordonnancement, et l'analyse bénéficie même d'améliorations par Jung & Lee [28].

Une approche récente de l'ordonnancement de systèmes multi-cœurs est fournie par Kritikakou [31, 32]. Le principe repose sur le monitoring en cours d'exécution de l'utilisation des ressources, de façon à garantir le respect des WCET. Kritikakou identifie ainsi quand une tâche critique va souffrir d'interférences de tâches moins importantes, de part le partage de la plateforme physique (utilisation des bus de communication, espaces mémoire...). En monitorant la tâche critique, il est possible d'identifier le moment au delà duquel il y a risque qu'elle dépasse son échéance d'exécution. A ce point donné, il est possible de bloquer la tâche non critique de façon à stopper les interférences. Cette approche d'abord statique mène à une amélioration plus dynamique.

4 Adaptation Dynamique

4.1 Gestion temps-réel dynamique

Plusieurs approches proposent la ré-allocation des ressources basée sur les informations qui dérivent du monitoring de leur utilisation, par exemple les accès mémoire. Dans [42] les *interférence-sensitive* WCETs sont déterminés selon une analyse préliminaire de l'utilisation des ressources par les tâches. Les ressources partagées sont ensuite partitionnées hors-ligne avec les tâches. Un moniteur observe durant l'exécution l'utilisation en ressource des tâches et suspend celles qui sur-utilisent leur capacité allouée. Dans [41], le partitionnement des ressources peut être modifié de façon dynamique, quand elles sont sous utilisées.

Dans [58, 59] les accès mémoire sont réservés aux tâches critiques. Un contrôleur régule pendant l'exécution l'accès aux mémoires partagées et s'assure de l'isolation temporelle entre les tâches. Une solution de profilage hors-ligne a été proposée dans [38], qui trouve les pages mémoire les plus fréquemment utilisées par une tâche donnée. Ces informations sont utilisées pour modifier la localisation des variables dans les caches partagés de façon à réduire les interférences.

L'approche hardware de [37] permet de monitorer uniquement quand on dispose d'un temps-mort de calcul, garantissant que le monitoring n'impacte pas la validation des contraintes de temps-réel. Si aucun temps-mort n'existe, une opération d'abandon de tâches non critiques minimise le dépassement généré par le monitoring.

Des travaux récents menés au LAAS [33, 34] proposent une approche innovatrice pour gérer l'exécution sûre d'un système à niveau de criticité mixte sur une plateforme multi-cœurs. Cette approche d'abord appliquée avec des calculs statiques lors de la conception a été étendue pour effectuer

des calculs en ligne pour gérer l'ordonnancement de façon dynamique. Pour les tâches critiques, deux valeurs de WCET sont calculées à différents points d'observation :

- Cas 1 : La tâche critique est exécutée en isolation, ce qui correspond au plus petit *WCET*,
- Cas 2 : La tâche critique est exécutée en parallèle d'une charge maximale (*WCET* maximal) : le processeur est sollicité au maximum,

Remarquant que ce dernier *WCET* en charge maximale correspond à la définition usuelle d'un *WCET* sur une plateforme multi-cœurs, il s'avère que ce calcul donnera en général une sur-réservation des ressources et donc une sous-utilisation à l'exécution. Il est alors possible d'exécuter les tâches via un ordonnancement classique basé sur les WCET usuels, et via les calculs aux différents points d'observation, de basculer sur un fonctionnement critique où toutes les tâches non critiques sont stoppées, de façon à se retrouver dans la configuration connue où les tâches critiques s'exécutent en isolation. De cette façon, il est possible de garantir de façon sûre que les tâches critiques s'exécutent dans les délais, tout en optimisant la puissance de calcul pour exécuter les tâches non critiques.

4.2 Virtualisation

4.2.1 Présentation

L'utilisation de plateformes physiques multi-cœurs est en grande partie liée à la possibilité de pouvoir héberger des systèmes qui jusqu'alors étaient dans des calculateurs séparés. Une problématique qui survient alors est de pouvoir héberger des tâches qui reposent sur une base logicielle différente. En effet, là où les tâches critiques vont évoluer soit sur une architecture type AUTOSAR, soit sur un OS temps-réel, d'autres tâches à temps-réel "mou" s'exécutent sur des systèmes d'exploitation d'usage général comme Linux. De façon à associer les tâches sur une même plateforme, il est donc nécessaire de fournir une interface qui permette la cohabitation de plusieurs systèmes d'exploitation. C'est ainsi que le principe de Virtualisation a vu le jour. Il s'agit d'une couche logicielle nommée Moniteur de Machines Virtuelles (*Virtual Machine Manager - VMM*) ou encore Hyperviseur qui fait l'interface entre la couche matérielle et les systèmes d'exploitation. L'Hyperviseur gère alors des Machines Virtuelles, qui servent de support pour héberger un type de système d'exploitation donné.

Les Machines Virtuelles doivent respecter trois propriétés clé, selon Popek & Goldberg [45]. Tout d'abord la propriété d'**équivalence** qui implique que les performances d'un programme soient équivalentes au sein d'une machine virtuelle et nativement. Cette propriété a deux exceptions, qui sont d'une part l'écart dû à l'exécution du Moniteur de Machines Virtuelles et d'autre

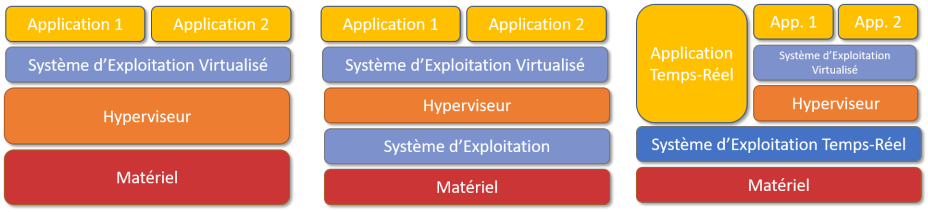


FIGURE 6 – Types d'Hyperviseurs

part des écarts dûs à l'utilisation de ressources (mémoire notamment) partagées. Il faut pour ce dernier point maîtriser la gestion des ressources partagées entre machines virtuelles afin de limiter au maximum cet écart. La seconde propriété est que l'hyperviseur doit avoir un contrôle total de l'utilisation des ressources matérielles. Et enfin la dernière propriété requiert que le processeur exécute directement l'ensemble des instructions des tâches qui sont le plus utilisées, de façon à garantir les bonnes performances.

4.2.2 Types d'Hyperviseurs

On peut caractériser deux types d'Hyperviseurs, qui ont chacun leurs avantages et inconvénients :

— Type 1 - Les hyperviseurs de type 1, dit natifs, s'exécutent directement sur la plateforme matérielle. Ils lancent alors des systèmes d'exploitation distincts dans des machines virtuelles.

— Type 2 - Les hyperviseurs de type 2, s'exécutent au sein d'un système d'exploitation dit système hôte. On a donc un premier système d'exploitation installé sur le matériel, qui exécute l'hyperviseur. C'est ce que l'on retrouve classiquement sur un ordinateur personnel.

On peut noter aussi un troisième type d'hyperviseur dit hybride qui dérive du type 1. Ce dernier exécute l'hyperviseur au sein d'un système d'exploitation, mais qui a pour spécificité d'être temps-réel. Ce système hôte exécute alors en concurrence l'hyperviseur, et des tâches critiques. De cette façon, il est possible sur le même support physique d'avoir une tâche critique qui s'exécute nativement sur un système d'exploitation temps-réel, et en concurrence, l'hyperviseur qui pourra exécuter des tâches moins critiques dans des machines virtuelles. L'ensemble de ces architectures sont représentées en figure 6.

Dans les systèmes embarqués, on constate le plus souvent l'utilisation d'hyperviseurs de type 1, pour des raisons de performances. On a notamment une diminution du volume de code occupé par l'hyperviseur sur la plateforme physique. Cela offre plusieurs avantages comme le gain d'embarquabilité (qui laisse donc un espace mémoire plus important pour les applications) et cela

facilite les tests et la certification de par la simplification de l'architecture. Grâce à l'utilisation d'hyperviseur dans le cadre des architectures automobiles, il devient possible d'assurer les contraintes de séparation spatiale et temporelle des applications sur le même calculateur, en apportant une vision globale du système afin de le superviser et de s'assurer du respect des contraintes de temps-réel. Le fait de déporter sur du logiciel la configuration de l'architecture logicielle permet une flexibilité du système et donc sa mise à jour et son amélioration de façon simplifiée. Les hyperviseurs sont donc une très bonne piste à explorer dans le cadre d'une exploitation de calculateurs pour héberger des systèmes multi-critiques.

5 Conclusion

L'évolution constante des systèmes industriels amène à l'utilisation d'architectures électriques, électroniques et logicielles innovantes qui nécessitent de nouvelles méthodes afin de continuer à garantir les contraintes propre aux systèmes critiques. Dans le domaine automobile, cela se traduit par l'arrivée des voitures connectées et autonomes, qui demande des architectures à forte adaptabilité tout en respectant les contraintes de temps-réel et de sûreté de fonctionnement.

L'objet de la thèse consiste à proposer une méthode de conception et de validation d'une architecture E/E et de l'architecture logicielle associée, basées sur les nouvelles technologies embarquées (processeurs multi-cœurs, Hyperviseur..) pour la génération 2025-2030 des véhicules. Les travaux de cette thèse viseront plus particulièrement à maîtriser le dimensionnement, l'allocation et l'ordonnancement des ressources que de telles architectures E/E et logicielle devront mettre à disposition des applications embarquées. Ces architectures devront garantir le respect d'exigences de performance, de disponibilité, de sûreté de fonctionnement et de cyber-sécurité, tout en offrant l'évolutivité requise dans le contexte du développement de véhicules électriques, connectés et autonomes. Un point critique de ce travail sera de maîtriser la complexité et la non-predictibilité des architectures multi-cœurs envisagées pour la génération 2025-2030.

Références

[1] T. Ali et K. Kim. « Cluster-based multicore real-time mixed-criticality scheduling ». In : *Journal of Systems Architecture*. 2017, p. 45–58.

[2] J. Anderson, S. K. Baruah et B. B. Brandenburg. « Multicore Operating-System Support for Mixed Criticality ». In : *WMC*. Avr. 2009.

[3] ARINC 653. *ARINC specification 653 - Avionics Application Software Standard Interface*. 2003.

[4] A. Avizienis, J.-C. Laprie, B. Randell et C. Landwehr. « Basic concepts and taxonomy of dependable and secure computing ». In : *IEEE transactions on dependable and secure computing* 1.1 (2004), p. 11–33.

[5] M. A. Awan, D. Masson et E. Tovar. « Energy efficient mapping of mixed criticality applications on unrelated heterogeneous multicore platforms ». In : *Industrial Embedded Systems (SIES), 2016 11th IEEE Symposium on*. IEEE. 2016, p. 1–10.

[6] S. K. Baruah et G. Fohler. « Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems ». In : *RTSS*. 2011, p. 3–12.

[7] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster et L. Stougie. « Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems ». In : *Journal of the ACM (JACM)* 62.2 (2015), p. 14.

[8] S. Baruah, V. Bonifaci, G. D'Angelo, H. L., A. Marchetti-Spaccamela, N. Megow et L. Stougie. « Scheduling Real-Time Mixed-Criticality Jobs ». In : *Trans. Computers* 61.8 (2012), p. 1140–1152.

[9] S. Baruah, B. Chattopadhyay, H. Li et I. Shin. « Mixed-criticality scheduling on multiprocessors ». In : *Real-Time Systems* (2013), p. 1–36.

[10] S. Baruah, L. Haohan et L. Stougie. « Towards the Design of Certifiable Mixed-criticality Systems ». In : *RTAS*. 2010, p. 13–22.

[11] S. Baruah et S. Vestal. « Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications ». In : *ECRTS*. 2008, p. 147–155.

[12] M. Blanchet. « Industrie 4.0 : nouvelle donne industrielle, nouveau modèle économique ». FR. In : *Géoéconomie* 82.5 (2016), p. 37–53. url : `https://www.cairn.info/revue-geoeconomie-2016-5-page-37.htm`.

[13] B. Brandenburg et J. Anderson. « Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors ». In : *ECRTS*. 2007, p. 61–70.

[14] A. Burns et B. Baruah. « Towards A More Practical Model for Mixed Criticality Systems ». In : *Proc. 1st Workshop on Mixed Criticality Systems (WMC), RTSS*. 2013, p. 1–6.

[15] A. Burns et S. Baruah. « Timing Faults and Mixed Criticality Systems ». In : *Dependable and Historic Computing*. T. 6875. Lecture Notes in Computer Science. 2011, p. 147–166.

[16] J. Craveiro et J. Rufino. « Adaptability support in time-and space-partitioned aerospace systems ». In : *ADAPTIVE 2010, The Second International Conference on Adaptive and Self-Adaptive Systems and Applications*. 2010, p. 152–157.

[17] G. Durrieu, M. Faugere, S. Girbal, D. G. Pérez, C. Pagetti et W. Puffitsch. « Predictable flight management system implementation on a multicore processor ». In : *Embedded Real Time Software (ERTS'14)*. 2014.

[18] P. T. Eugster, P. A. Felber, R. Guerraoui et A.-M. Kermarrec. « The Many Faces of Publish/Subscribe ». In : *ACM Comput. Surv.* 35.2 (juin 2003), p. 114–131. issn : 0360-0300. url : `http://doi.acm.org/10.1145/857076.857078`.

[19] T. Fleming et A. Burns. « Extending Mixed Criticality Scheduling ». In : *RTSS*. 2013.

[20] S. Fürst et M. Bechter. « AUTOSAR for Connected and Autonomous Vehicles : The AUTOSAR Adaptive Platform ». In : *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*. IEEE. 2016, p. 215–217.

[21] G. Giannopoulou, N. Stoimenov, P. Huang et L. Thiele. « Scheduling of Mixed-criticality Applications on Resource-sharing Multicore Systems ». In : *EMSOFT*. Montreal, Canada : IEEE, 2013, 17 :1–17 :15.

[22] L. Haohan et S. Baruah. « Global Mixed-Criticality Scheduling on Multiprocessors ». In : *ECRTS*. 2012, p. 166–175.

[23] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson et D. M. Johnson. « RTOS Support for Multicore Mixed-Criticality Systems ». In : *RTASum*. 2012, p. 197–208.

[24] V. Hilderman et T. Baghi. *Avionics certification : a complete guide to DO-178 (software), DO-254 (hardware)*. Avionics Communications, 2007.

[25] IEC 61508. *Functional safety of electrical/electronic/programmable electronic safety-related systems*. Rapp. tech. IEC 61508. The International Electrotechnical Commission, 2010.

[26] C. Insaurralde, M. Seminario, J. Jimenez et J. Giron-Sierra. « Model-based development framework for distributed embedded control of aircraft fuel systems. » English. In : *Digital Avionics Systems Conference (DASC)*. IEEE/AIAA 29th, 2010, p. 2–14.

[27] *ISO 26262, Road Vehicle Safety*. 2011.

[28] N. Jung et J. Lee. « Improved Scedulability Analysis of Fixed Priority for Mixed-criticality Real-Time Multiprocessor Systems ». In : *Springer Singapore*. Singapore, 2018, p. 1403–1409.

[29] O. Kelly, H. Aydin et B. Zhao. « On partitionned scheduling of fixed-priority mixed-riticality task sets ». In : *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. 2011, p. 1051–1059.

[30] N. Kim, B. C. Ward, M. Chisholm, C.-Y. Fu, J. H. Anderson et F. Smith. « Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning ». In : *Proc. RTAS*. 2016, p. 1–12.

[31] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange et M. Roy. « Run-time Control to Increase Task Parallelism in Mixed-Critical Systems ». In : *ECRTS*. 2014.

[32] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, M. Roy et F. Vargas. « Monitoring On-Line Timing Information To Support Mixed-Critical Workloads ». In : *IEEE Real-Time Systems Symposium, RTSS*. 2013, p. 9–10.

[33] A. Kritikakou, T. Marty, C. Pagetti, C. Rochange, M. Lauer et M. Roy. « Multiplexing Adaptive with Classic AUTOSAR ? Adaptive Software Control to Increase Resource Utilization in Mixed-Critical Systems ». In : *CARS 2016*. URL : `https://hal.archives-ouvertes.fr/hal-01375576`.

[34] A. Kritikakou, T. Marty et M. Roy. « DYNASCORE : DYNAmic Software COntroller to increase REsource utilization in mixed-critical systems ». In : *ACM Transactions on Design Automation of Electronic Systems (TODAES)* To appear (2017).

[35] M. Lauer, M. Amy, J.-C. Fabre, M. Roy, W. Excoffon et M. Stoicescu. « Resilient computing on ROS using adaptive fault tolerance ». In : *Journal of Software : Evolution and Process* ().

[36] M. Lauer, M. Amy, J.-C. Fabre, M. Roy, W. Excoffon et M. Stoicescu. « Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS ». In : *IEEE HASE 2016*. URL : https://hal.archives-ouvertes.fr/hal-01288098.

[37] D. Lo, M. Ismail, T. Chen et G. E. Suh. « Slack-aware opportunistic monitoring for real-time systems ». In : *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE. 2014, p. 203–214.

[38] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo et R. Pellizzoni. « Real-time cache management framework for multi-core architectures ». In : *RTAS*. 2013, p. 45–54.

[39] H. Martorell. « Architecture et processus de développement permettant la mise à jour dynamique de systèmes embarqués automobiles ». Thèse de doct. 2014. URL : http://www.theses.fr/2014INPT0108.

[40] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah et J. A. Scoredos. « Mixed-Criticality Real-Time Scheduling for Multicore Systems. » In : *CIT*. 2010, p. 1864–1871.

[41] J. Nowotsch et M. Paulitsch. « Quality of Service Capabilities for Hard Real-time Applications on Multi-core Processors ». In : *RTNS*. 2013, p. 151–160.

[42] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener et M. Schmidt. *Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement*. Rapp. tech. Univ. Augsburg, Germany, 2013.

[43] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone et J. C. Phillips. « GPU computing ». In : *Proceedings of the IEEE* 96.5 (2008), p. 879–899.

[44] R. M. Pathan. « Schedulability Analysis of Mixed-Criticality Systems on Multiprocessors ». In : *ECRTS*. 2012, p. 309–320.

[45] G. J. Popek et R. P. Goldberg. « Formal Requirements for Virtualizable Third Generation Architectures ». In : *Commun. ACM* 17.7 (juil. 1974), p. 412–421. ISSN : 0001-0782. URL : http://doi.acm.org/10.1145/361011.361073.

[46] P. J. Prisaznuk. « ARINC 653 role in Integrated Modular Avionics (IMA) ». In : *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. 2008 IEEE/AIAA 27th Digital Avionics Systems Conference. Oct. 2008, 1.E.5–1–1.E.5–10.

[47] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler et A. Y. Ng. « ROS : an open-source Robot Operating System ». In : *ICRA workshop on open source software*. T. 3. 3.2. Kobe, Japan. 2009, p. 5.

[48] P. Rodriguez, L. Georges, Y. Abdeddaim et J. Goosens. « Multi-criteria evaluation of partitioned EDF-VD for mixed criticality systems upon identical processors ». In : *Proc. 1st WMC, RTTS*. 2013, p. 49–54.

[49] A. Schmidt, A. K. Dey, A. L. Kun et W. Spiessl. « Automotive user interfaces : human computer interaction in the car ». In : *CHI'10 Extended Abstracts on Human Factors in Computing Systems*. ACM. Atlanta, Georgia, USA, 2010, p. 3177–3180.

[50] T. Schoofs, S. Santos, C. Tatibana et J. Anjos. « An integrated modular avionics development environment ». English. In : 2009. isbn : 1424440785.

[51] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin et al. « Larrabee : a many-core x86 architecture for visual computing ». In : *ACM Transactions on Graphics (TOG)* 27.3 (2008), p. 18.

[52] M. Smotherman. « History of multithreading ». In : *Retrieved on* (2005), p. 12–19.

[53] M. Stoicescu, J.-C. Fabre et M. Roy. « Architecting resilient computing systems : A component-based approach for adaptive fault tolerance ». In : *Journal of Systems Architecture* 73 (fév. 2017), pp.6–16. url : `https://hal.archives-ouvertes.fr/hal-01472877/`.

[54] D. Tamas-Selicean et P. Pop. « Task mapping and partition allocation for mixed criticality real-time systems ». In : *IEEE Pacific Rim Int. Sym. on Dependable Computing* (2011), p. 282–283.

[55] S. Vestal. « Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance ». In : *RTSS*. 2007, p. 239–243.

[56] R. Warschofsky. « AUTOSAR Software Architecture ». In : 2009.

[57] M. V. Wilkes. « Slave memories and dynamic storage allocation ». In : *IEEE Transactions on Electronic Computers* 2 (1965), p. 270–271.

[58] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo et L. Sha. « Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality ». In : *ECRTS*. Juil. 2012, p. 299–308.

[59] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo et L. Sha. « MemGuard : Memory bandwidth reservation system for efficient performance isolation in multi-core platforms ». In : *RTAS*. 2013, p. 55–64.