RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

Évaluation du silence sur défaillance d'un exécutif temps-réel construit à base de composants COTS

Pascal Chevochot — INRIA / IRISA
Isabelle Puaut — INSA / IRISA
{pchevoch,puaut}@irisa.fr

Campus de Beaulieu
F-35042 Rennes Cédex, France

Résumé
Principalement pour des raisons économiques, de plus en plus de systèmes tolérants aux fautes sont construits à partir de composants COTS (Commercial Off-The-Shelf). Une hypothèse communément répandue pour la construction de tels systèmes est celle de *silence sur défaillance.* L'objectif de cet article est de déterminer de manière expérimentale (par injection de fautes) dans quelle mesure des calculateurs exécutant un support d'exécution temps-réel construit à partir de composants sur étagères sont silencieux sur défaillance en présence de fautes physiques. Le support d'exécution cible est le système HADES [1], dédié à l'exécution d'applications distribuées temps-réel strict ayant des contraintes de tolérance aux fautes. HADES inclut en particulier des mécanismes de détection d'erreurs entièrement implantés par logiciel, permettant d'arrêter un calculateur quand une erreur est détectée. Le principal résultat de notre travail est de montrer que la couverture du silence sur défaillance atteint 99.1%, ce qui divise le nombre de violations de l'hypothèse de silence sur défaillance par un facteur 22 comparé à un système sans détection d'erreurs. Par ailleurs, nous évaluons les mécanismes de détection d'erreurs inclus dans HADES à l'aide de plusieurs métriques (pourcentage de premières détections, latence), permettant ainsi de sélectionner les mécanismes de détection d'erreurs les mieux adaptés à un contexte particulier.

Mots-clés : Tolérance aux fautes, temps-réel strict, silence sur défaillance, injection de fautes, détection d'erreurs.

---

1. Introduction

Principalement pour des raisons économiques, de plus en plus de systèmes temps-réel tolérants aux fautes sont construits par assemblage de composants sur étagères (en anglais COTS, pour Commercial Off-The-Shelf), c'est à dire des composants à vocation générale n'ayant pas été conçus pour un domaine d'applications donné. Comme les composants COTS ne sont généralement pas conçus pour tolérer les fautes, ces derniers doivent être complétés par des mécanismes de tolérance aux fautes (détection d'erreurs, traitement d'erreurs, traitement de fautes) de manière à les supporter.

L'hypothèse de *silence sur défaillance* est couramment utilisée pour construire des systèmes tolérants aux fautes. Un système est dit *silencieux sur défaillance* s'il fonctionne selon ses spécifications ou s'arrête sans production de résultats incorrects. Bien que le silence sur défaillance soit une hypothèse largement utilisée, peu d'études sont consacrées à l'évaluation de la couverture de cette hypothèse dans des systèmes construits à partir de composants COTS.

L'objectif principal de ce travail est de déterminer la couverture de l'hypothèse de silence sur défaillance de calculateurs exécutant un support d'exécution distribué temps-réel strict tolérant aux fautes, construit par assemblage de composants COTS (carte mère, réseau, système d'exploitation). Nous qualifions de *temps-réel strict* un support d'exécution capable d'exécuter des applications dont les échéances ne doivent en aucun cas être dépassées.

Le support d'exécution considéré est le système HADES [1]. HADES (*Highly Available Distributed Embedded System*) est un support d'exécution distribué tolérant aux fautes pour applications temps-réel strict.

HADES est un intergiciel (couche logicielle intermédiaire) qui complète un micro-noyau temps-réel COTS par des mécanismes de tolérance aux fautes : mécanismes de détection d'erreurs pour assurer le silence sur défaillance (arrêt d'un calculateur dès qu'une erreur est détectée), protocoles tolérants aux fautes. Les mécanismes de détection d'erreurs sont entièrement implantés par logiciel, de manière à ne pas recourir à l'utilisation de matériel dédié pour la tolérance aux fautes, coûteux à acquérir et maintenir.

L'évaluation du silence sur défaillance des calculateurs exécutant HADES a été effectuée en utilisant une technique expérimentale : l'*injection de fautes*. Un injecteur de fautes a été utilisé nuits et jours pendant plus de trois mois pour injecter des fautes dans la mémoire du support d'exécution et observer leurs conséquences sur le fonctionnement d'une application de charge. Les fautes injectées sont représentatives des fautes physiques transitoires. L'influence des fautes est étudiée au travers d'un ensemble complet de métriques : couverture de l'hypothèse de silence sur défaillance, et pour chaque mécanisme de détection d'erreurs, pourcentage de premières détections, latences de détection.

Contribution de l'article

La contribution essentielle de cet article est de montrer jusqu'à quel point un support d'exécution construit par assemblage de composants COTS peut être silencieux sur défaillance. Les résultats de l'évaluation montrent que malgré les mécanismes de détection d'erreurs intégrés à HADES, la couverture du silence sur défaillance ne dépasse pas 99.1%, ce qui à notre avis exclut l'utilisation de composants COTS pour la construction de systèmes à sûreté critique reposant sur le silence sur défaillance. Une autre contribution de l'article est d'évaluer les mécanismes de détection d'erreurs que nous utilisons grâce à un ensemble complet de métriques. Ceci permet notamment de sélectionner l'ensemble de mécanismes de détection d'erreurs le mieux adapté à un contexte particulier. À notre connaissance, notre étude est la seule à évaluer la couverture du silence sur défaillance pour un support d'exécution *distribué*, *temps-réel strict* et construit par *assemblage de composants COTS*.

Plan de l'article

La suite de cet article est organisée de la manière suivante. Le paragraphe 2 donne une vue d'ensemble du support d'exécution HADES, puis se concentre sur les mécanismes de détection d'erreurs qu'il inclut. Le paragraphe 3 décrit les conditions expérimentales dans lesquelles l'évaluation a été effectuée (injecteur de fautes, application de charge). Le paragraphe 4 évalue les mécanismes de détection d'erreurs de HADES au travers de différentes métriques. Notre travail est comparé à des travaux apparentés dans le paragraphe 5 avant de conclure.

2. Le support d'exécution temps-réel distribué Hades

2.1. Vue d'ensemble

La plupart des systèmes d'exploitation temps-réel COTS ne fournit pas de support adéquat pour l'exécution d'applications ayant à la fois des contraintes de temps-réel strict et de tolérance aux fautes. Le plus souvent, ces systèmes fournissent, comme mode de communication à distance, des primitives d'échanges de messages qui garantissent rarement la livraison de messages en temps borné et/ou ne supportent pas les fautes du réseau ou des calculateurs. De plus, peu de systèmes d'exploitation COTS permettent de détecter et recouvrir les arrêts de calculateurs. Le support d'exécution HADES [1] compense ce manque de fonctionnalités des systèmes d'exploitation temps-réel COTS en fournissant dans un intergiciel intercalé entre un noyau de système d'exploitation COTS et l'application : un protocole de diffusion fiable en temps borné, un protocole de gestion de groupes, un protocole de synchronisation des horloges et un ordonnanceur distribué. De plus, afin que chaque calculateur respecte l'hypothèse de silence sur défaillance, HADES inclut un ensemble de mécanismes de détection d'erreurs (§ 2.2), qui arrêtent un calculateur dès qu'une erreur est détectée. Tous les services fournis par HADES supportent les fautes physiques temporaire et permanentes et s'exécutent en temps borné, et ce même en présence de fautes. Par ailleurs, pour éviter un surcoût financier, tous les services sont entièrement implantés par logiciel, et donc aucun matériel dédié à la tolérance aux fautes n'est utilisé.

De manière plus concrète, HADES est composé d'un petit noyau d'exécution et d'un ensemble de tâches de services. Le *noyau d'exécution*, conçu pour des raisons de portabilité, fournit un ensemble d'abstractions (espaces d'adressage, unités d'exécution, segments de mémoire), utilisées pour l'exécution de tâches temps-réel. Ces abstractions peuvent être aisément associées à celles de tout noyau temps-réel. À ce jour, HADES

a été porté sur les noyaux Chorus, RTEMS et émulé au dessus d'Unix (dans cet article, nous nous concentrons sur l'implantation au dessus de Chorus). *Les services* implantent les protocoles temps-réel distribués mentionnés plus haut (diffusion multipoint, synchronisation des horloges, etc).
Comme HADES est conçu pour exécuter des applications temps-réel strict (dont les échéances doivent être impérativement respectées), toutes les ressources sont dimensionnées et allouées statiquement, sans recourir à l'allocation dynamique. Par ailleurs, toutes les informations qui sont nécessaires hors-ligne pour vérifier le respect des échéances (temps d'exécution au pire cas des tâches, périodicité et échéance des tâches) sont conservées pendant l'exécution à des fins de détection d'erreurs. Ces informations sont regroupées dans un graphe contenant toutes les propriétés connues statiquement sur les tâches.
Le support d'exécution HADES s'exécute au dessus d'un réseau ATM à 155Mb/s interconnectant des calculateurs PCs 90 MHz exécutant le système d'exploitation temps-réel Chorus 3.1. Les machines sont également reliées par un réseau Ethernet à 10 Mb/s utilisé exclusivement pour l'injection de fautes.

2.2. Mécanismes de détection d'erreurs (MDEs)

Comme HADES est conçu pour exécuter des applications temps-réel strict, les mécanismes de détection d'erreurs (MDEs) qu'il inclut doivent s'exécuter en temps à la fois faible et borné. Par ailleurs, pour des raisons de coût, notre volonté est d'éviter l'utilisation de matériel dédié à la détection d'erreurs (e.g. [12]), et de ce fait tous les MDEs sont implantés par logiciel. Il est possible de générer différentes configurations de HADES en désactivant des ensembles quelconques de MDEs. Par exemple, il est possible de générer des configurations de HADES dans lesquelles les tâches, soit se partagent toutes le même espace d'adressage, soit s'exécutent dans des espaces d'adressage séparés.
Les MDEs de HADES sont présentés dans le tableau ci-dessous. La première catégorie de MDEs détecte des erreurs par comparaison d'informations redondantes. La deuxième catégorie de MDEs inclut des mécanismes de vérification de contrôle de flot et de temps d'exécutions, sans utilisation de redondance. Enfin, la troisième catégorie regroupe des MDEs de bas niveau (MDEs intégrés au processeur et au système d'exploitation Chorus).

Mécanisme	**Description**
	a. MDEs à base de redondance
CODING	*Encodage de données lors des communications.* Ce mécanisme ajoute à chaque message une somme de contrôle qui est vérifiée avant la livraison du message.
ROBUST	*Structures de données robustes.* Ce mécanisme est utilisé pour détecter les erreurs affectant les structures de données de type *liste* et *pile*. Une redondance (e.g. identificateur, double chaînage pour les listes) est ajoutée à la structure de données pour vérifier son intégrité à chacune de ses utilisations.
CODING_STATIC	*Encodage des structures de données statiques.* Une somme de contrôle est ajoutée à la structure de données décrivant les propriétés statiques des tâches. La conformité entre la structure de données et la somme de contrôle est vérifiée quand le support d'exécution est inactif (tâche *idle*).
	b. Vérifications temporelles et de flot de contrôle
DEADLINE	*Détection des dépassements d'échéances.* Cette vérification est effectuée grâce aux informations statiques nécessaires à l'analyse d'ordonnancement d'un système.
TIMEOUT	*Détection des temps d'exécution au pire cas des tâches.* Ce mécanisme vérifie que le temps d'exécution de chaque tâche ne dépasse pas son pire temps d'exécution.
ARRIVAL	*Vérification de périodicité.* Ce mécanisme consiste à vérifier que toute tâche arrive bien avec la loi d'arrivée prévue (strictement périodique ou sporadique).
CALLGRAPH	*Vérification du graphe de flot de contrôle.* Ce mécanisme vérifie que le flot de contrôle du support d'exécution est conforme à un graphe de flot de contrôle (enchaînements possibles entre fonctions) généré avant exécution. Ce mécanisme est implanté par réécriture du code source du support d'exécution (ajout de code de vérification en entrée et sortie des fonctions).
	continue page suivante

Mécanisme	Description
AUTOMATON	*Vérifications des états de protocoles.* Ce mécanisme vérifie, pour les protocoles implantés sous forme d'automates, la validité des transitions (e.g. état de chaque tâche qui peut être bloqué, prêt, actif, terminé).
ECI	*Error capturing instructions.* Ce mécanisme insère des instructions déclenchant une exception (trappes) dans les zones de code et de données qui ne sont pas utilisées en fonctionnement normal, mais qui peuvent l'être si une faute provoque une déviation du flot de contrôle.
VMEMORY	*Virtual memory.* Ce mécanisme déclenche une exception quand une tâche tente d'accéder à un segment mémoire qui n'est pas projeté dans l'espace d'adressage courant. Un espace d'adressage est associé à chaque tâche de service qui peut s'exécuter dans le mode utilisateur du processeur.
STRUCTURE	*Vérification des structures de données.* Le mécanisme STRUCTURE vérifie, sans utilisation de redondance, la validité des accès aux types de données tableau et liste (vérification d'indices de tableaux, des chaînages dans les listes).
SEMAN	*Vérifications sémantiques.* Ce mécanisme vérifie, par le biais d'assertions disposées dans le code du support d'exécution, que des invariants liés à la sémantique du fonctionnement du support d'exécution sont vérifiés.
	c. MDEs de bas niveau
CPU	CPU représente l'ensemble des mécanismes de détection d'erreurs du processeur (ici, processeur Pentium). Cette classe de MDEs ne peut pas être désactivée.
OS	OS représente l'ensemble des MDEs du noyau Chorus (codes d'erreur retournés par les appels système, paniques).

3. Conditions expérimentales

La performance des MDEs introduits au paragraphe 2.2 a été évaluée grâce à de l'injection de fautes réalisée par logiciel. Nous définissons le modèle des fautes injectées (§ 3.1), précisons quelles expériences sont utilisées pour évaluer les MDEs (§ 3.2), puis donnons quelques détails techniques sur l'injecteur de fautes (§ 3.3) et décrivons l'application de charge (§ 3.4).

3.1. Modèle de fautes

L'objectif de l'injection de fautes est de simuler l'occurrence de fautes physiques. Dans ce travail, nous simulons les fautes physiques transitoires, qui sont les fautes physiques les plus courantes dans les systèmes informatiques [10]. L'injection est effectuée par logiciel, en introduisant la manifestation des fautes (les *erreurs*) dans la mémoire du support d'exécution. L'injection est déclenchée à un instant aléatoire, et l'emplacement mémoire modifié est lui aussi sélectionné de manière aléatoire (dans le code, les données ou la pile du support d'exécution). La corruption mémoire consiste à inverser un ou plusieurs bits (un seul dans le cadre de cet article) d'un octet. Ce modèle de fautes est en accord avec la plupart des travaux utilisant l'injection de fautes par logiciel [2, 8, 3, 14]. Une bonne discussion sur la représentativité de ce mode d'injection vis-à-vis des fautes physiques transitoires peut être trouvée dans l'article [3].

3.2. Injection et activation de fautes, détections justifiées

L'injection d'une faute peut avoir deux types de conséquences :

– Le mot corrompu par l'injection n'est pas utilisé par le processeur. Cette situation, de toute évidence, ne peut pas aboutir à une défaillance du système.
– Le processeur utilise le mot corrompu par l'injection. On dit alors que la faute a été *activée*.

Pour calculer la couverture du silence sur défaillance (§ 4.1), nous considérons uniquement les fautes activées. Considérer toutes les fautes injectées donnerait à notre avis des résultats trop dépendants du système cible (dans notre cas, seules 9% des fautes injectées sont activées, ce qui est beaucoup plus faible que dans d'autres systèmes comme Mafalda [14] ou ce pourcentage atteint 34%). La même approche est utilisée dans la plupart des systèmes utilisant de l'injection de fautes par logiciel (Mafalda, FERRARI, FINE). Par ailleurs, comme ceci est suggéré dans [15], nous utilisons pour mesurer les latences et efficacités des MDEs non pas toutes les erreurs détectées, mais uniquement les erreurs détectées de manière *justifiée* (détectées et amenant à une défaillance du système). Ceci, comme argumenté dans [15], permet de ne pas surestimer certains MDEs dits MDEs *hypersensibles*, qui détectent des erreurs alors que ces dernières seraient toujours restées latentes.

3.3. Implantation de l'injecteur de fautes

La mise en œuvre de l'injecteur de fautes repose sur l'utilisation des registres de déverminage du Pentium, permettant de lever une exception dès qu'une adresse donnée est accédée par le processeur. L'injection d'une faute est effectuée à un instant aléatoire. Si au terme d'un délai borné (10 s), la faute n'a pas été activée, les effets de l'injection sont annulés et une nouvelle faute est injectée (ce processus est répété jusqu'à activation de la faute).

La figure 1 montre l'environnement dans lequel ont été réalisées les expérimentations. Le système est composé d'un ensemble de machines, reliées par ATM et Ethernet. Quatre machines exécutent l'application de charge, et échangent des messages à travers ATM. La cinquième machine (machine Unix) récolte les traces émises par l'injecteur de fautes sur Ethernet. Les fautes sont injectées dans la mémoire d'une des machines (dans les zones grisées sur la figure1).

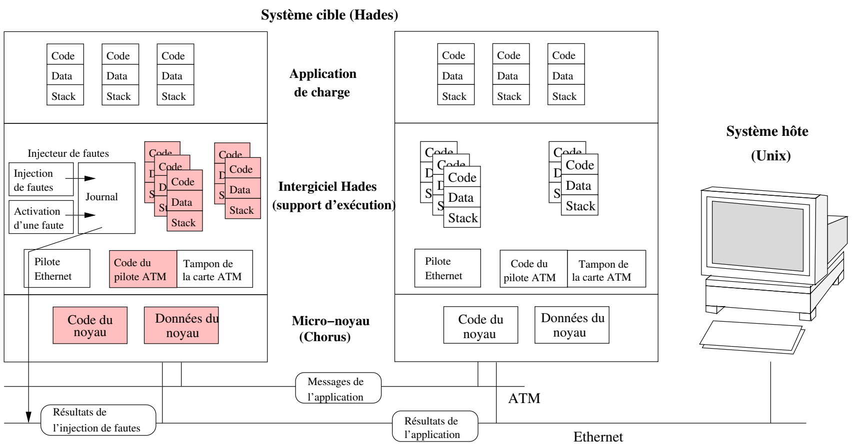

FIG. 1 – *Environnement expérimental*

Une campagne d'injections de fautes est constituée d'une suite d'expériences. Pour chaque expérience, (i) une application de charge (§ 3.4) est démarrée, et envoie régulièrement ses résultats sur Ethernet pour pouvoir déterminer la correction de son exécution; (ii) des fautes sont injectées jusqu'à ce que l'une d'elles soit activée; (iii) au terme d'un délai borné (ici, 20 secondes), l'application est stoppée et les traces récoltées par l'injecteur de fautes sont transmises via Ethernet à la machine Unix et les machines HADES sont alors redémarrées pour l'expérimentation suivante.

Comme la durée d'une expérimentation après activation d'une faute est bornée (ici 20 s), une absence de défaillance pendant cet intervalle est ambiguë. Cela peut indiquer que la faute a été masquée, mais aussi que la durée de l'expérimentation a été trop courte pour que l'erreur ait eu des conséquences (erreur *latente*). Nous avons expérimenté plusieurs intervalles plus longs que 20 s (jusqu'à 5 mn) et n'avons pas observé de modification du comportement de l'application de charge, ce qui lève l'ambiguïté.

3.4. Application de charge

L'application de charge est la partie logicielle d'une application de suivi d'objet. L'objectif est pour un ensemble d'objets esclaves de suivre la trajectoire d'un objet maître. La position des objets est donnée par des capteurs, et leur déplacement géré par des actionneurs (ici, les capteurs et actionneurs sont émulés par logiciel). Le calcul des trajectoires des esclaves est réalisé par une tâche distribuée sur les quatre machines. Cette tâche est exécutée périodiquement toutes les 250 ms, et son échéance est 250 ms. Pour pouvoir aisément déterminer la correction du déroulement de l'application de charge, l'objet maître est déplacé selon une trajectoire prédéfinie.

4. Résultats des expérimentations

Les MDEs introduits dans le paragraphe 2.2 sont évalués selon différentes métriques : couverture du silence sur défaillance (§ 4.1), pourcentage de premières détections (§ 4.2), et latence de détection (§ 4.3).

4.1. Couverture de l'hypothèse de silence sur défaillance

Le silence sur défaillance d'un calculateur exécutant le support d'exécution HADES est mesuré en injectant des fautes dans son image mémoire et en examinant leurs conséquences sur l'application de charge décrite dans le § 3.4 (il serait trop coûteux et complexe de tester le silence sur défaillance directement au niveau du support d'exécution). Une faute activée peut aboutir aux manifestions suivantes sur les résultats de l'application de charge :

STOP	Pas de résultats (arrêt d'au moins un calculateur)
INCORRECT	Résultats incorrects (valeur et/ou instant de production du résultat incorrects)
CORRECT	Résultats corrects dans le domaine des valeurs *et* le domaine temporel

La correction de l'application de charge est jugée de manière globale à l'architecture distribuée : son résultat est correct s'il l'est sur tous les calculateurs. Il est incorrect s'il l'est sur au moins un des calculateurs. Les deux premières classes de manifestations données ci-dessous peuvent être subdivisées en deux, selon que l'erreur s'est propagée ou non sur un autre calculateur. On appelle par la suite les catégories résultantes STOP-NOPROPAG, STOP-PROPAG, INCORRECT-NOPROPAG, INCORRECT-PROPAG et CORRECT. Un calculateur est silencieux sur défaillance si les résultats de l'application de charge sont corrects (catégorie CORRECT) ou s'il s'arrête sans propager d'erreur (catégorie STOP-NOPROPAG). La couverture est calculée sur l'ensemble des erreurs activées, de cardinal noté N_{act} :

$$coverage_{fail-silence} = \frac{(N_{CORRECT} + N_{STOP-NOPROPAG})}{N_{act}}$$

Le tableau 1 donne les manifestations observées par injection de fautes, ainsi que les couvertures de silence sur défaillance correspondantes. Deux configurations de HADES sont distinguées : (a) sans MDEs (i.e. avec uniquement CPU et les paniques noyau qui ne peuvent pas être désactivées) ; (b) avec tous les MDEs du § 2.2.

Manifestation	**(a) Pas de MDE**		**(b) Tous les MDEs**	
Fautes injectées	39025		29591	
Fautes activées	3152		3012	
CORRECT	1839	(58.3%)	1122	(37.3%)
STOP-NOPROPAG	703	(22.3%)	1862	(61.8%)
STOP-PROPAG	102	(3.2%)	22	(0.7%)
INCORRECT-NOPROPAG	33	(1.1%)	2	(0.1%)
INCORRECT-PROPAG	475	(15.1%)	4	(0.1%)
Couverture silence sur défaillance	80.64 %		99.07 %	
Intervalle de confiance (niveau de confiance de 95 %)	[79.49 %, 81.79 %]		[98.80 %, 99.33 %]	

TAB. 1 – *Couverture du silence sur défaillance*

Quand aucun MDE n'est utilisé, 19.4% des fautes activées aboutissent à une violation du silence sur défaillance, ce qui montre que sans MDE, le système n'est clairement pas silencieux sur défaillance. Quand tous les mécanismes de détection d'erreurs sont utilisés, 99.1% des fautes activées aboutissent à un comportement silencieux sur défaillance. 99.1% est indéniablement plus faible que les taux obtenus dans des systèmes où du matériel dédié à la tolérance aux fautes est utilisé. Néanmoins, les MDEs intégrés à HADES sont très efficaces, car ils divisent le nombre de violations du silence sur défaillance par un facteur 22.

Le tableau 1 montre qu'un grand pourcentage de violations du silence sur défaillance provient de propagations d'erreurs entre calculateurs. Un examen détaillé des expériences ayant produit ces violations [4] nous a amené aux deux conclusions suivantes : (i) dans un large pourcentage de ces expériences (90%), l'application n'a pas fourni de résultat incorrect (elle a été détectée sur un calculateur distant avant de se propager à l'application) ; (ii) nous ne voyons pas comment éviter les propagations sans utiliser de matériel spécifique.

4.2. Pourcentage de premières détections

Une métrique pour évaluer l'efficacité d'un MDE m est le pourcentage de premières détections effectuées par ce mécanisme sur un ensemble d'expériences. Soit m un MDE et soit E un ensemble d'expériences pour lesquelles une erreur a été détectée (par un MDE quelconque) de manière justifiée. Si $det(m)$ nomme le nombre d'expériences pour lesquelles m a été le premier mécanisme à détecter une erreur, alors le pourcentage de premières détections effectuées par m est $\frac{det(m)}{|E|}$.

La figure 2 montre la valeur de cette métrique pour un ensemble de 2086 expériences représentant des détections d'erreurs justifiées.

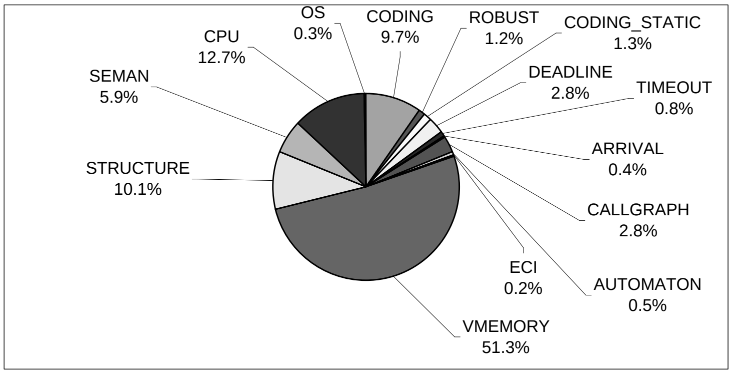

FIG. 2 – *Pourcentage de premières détections de MDEs*

La figure montre que la majorité des erreurs (51.3%) est détectée par le mécanisme VMEMORY. Un pourcentage plus faible d'erreurs est détecté par les MDEs CODING, STRUCTURE et CPU (approximativement 10% chaque). L'efficacité de STRUCTURE est due au fait que le support d'exécution HADES utilise de manière intensive les listes et les tableaux. Le mécanisme CALLGRAPH détecte environ 3% des erreurs. Les autres mécanismes, et en particuliers ceux qui contrôlent les temps d'exécution des tâches sont très peu efficaces puisqu'ils détectent moins de 2.5% d'erreurs.

4.3. Latence de détection d'erreurs

La *latence* de détection d'erreurs d'un MDE est le délai entre l'activation de la faute origine de l'erreur et sa détection par le MDE. Nous avons mesuré les latences de détection d'erreurs sur un ensemble de 2086 erreurs détectées de manière justifiée. Les latences ont été mesurées en lisant le compteur de cycles du Pentium, permettant ainsi de mesurer les durées avec une très bonne précision.

La latence moyenne de détection d'erreurs est de 47.5 ms. Un examen des répartitions des latences a montré l'existence d'un petit nombre de latences très importantes (de 1 à 20 s). Si l'on élimine les 10% de valeurs extrêmes, la latence de détection moyenne est de 7.8 ms. Pour 90% des expériences, la latence de détection est inférieure à 10 ms.

La figure 3 donne la latence de détection d'erreurs moyenne de chaque MDE. Sur la figure, les MDEs marqués d'une astérisque sont ceux qui détectent le plus petit nombre d'erreurs (moins de 20 sur l'ensemble des expériences), et par conséquent sont ceux dont le calcul de latence moyen souffre du plus d'imprécisions.

Les mécanismes de détection d'erreurs peuvent être divisés en trois grandes catégories selon leur latence de détection d'erreur. Les mécanismes de la première classe (DEADLINE, TIMEOUT, ARRIVAL) sont ceux dont la latence de détection d'erreurs est dépendante des caractéristiques temporelles des tâches exécutées. Par exemple, le mécanisme DEADLINE ne peut pas détecter une erreur avant le dépassement des échéances de tâches. Ici, étant données les caractéristiques temporelles des tâches (périodes de quelques dizaines de milli-secondes pour les tâches périodiques, temps d'exécution au pire cas calculés de manière sûre, donc surestimés) cette classe de mécanismes possède une latence de détection importante (de 20 à

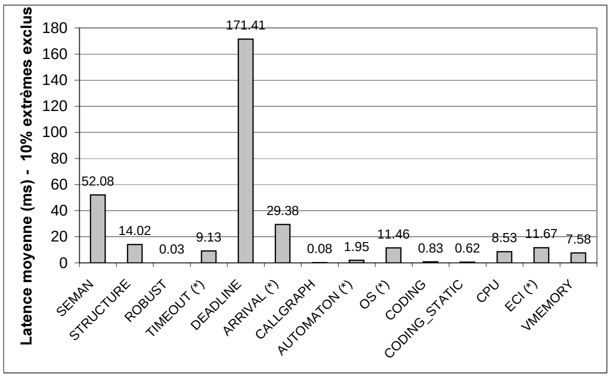

FIG. 3 – *Latence moyenne de détection d'erreurs par MDE*

140 ms en moyenne selon les mécanismes). Les MDEs de la seconde catégorie (VMEMORY, CPU) sont les MDEs implantés par matériel. Leur latence de détection moyenne est de 8 ms. Enfin, tous les autres MDEs peuvent être regroupés dans une troisième catégorie, exhibant des latences de détection plutôt faibles (en moyenne moins d'1 ms, sauf pour les mécanismes STRUCTURE et SEMAN).
La figure 4 montre pour chaque intervalle de latences de détections d'erreurs le nombre d'erreurs ayant été détectées dans cet intervalle.

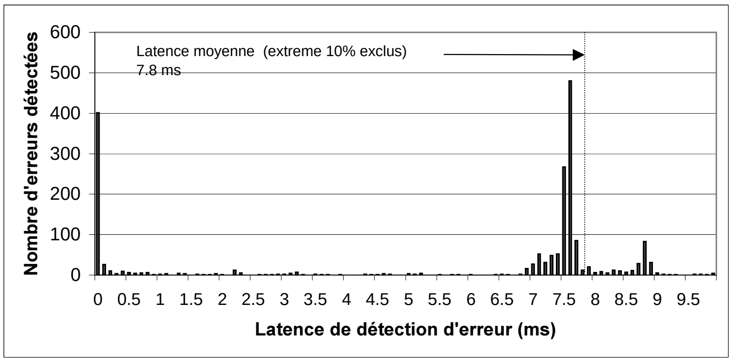

FIG. 4 – *Répartition des latences de détection d'erreurs*

Deux pics peuvent être observés dans cet histogramme. Le premier correspond aux MDEs de très faible latence (troisième catégorie introduite plus haut). Le deuxième pic correspond à des latences de détection entre 7 et 9 ms (deuxième catégorie de mécanismes introduite plus haut). La première catégorie de mécanismes n'apparaît pas de manière explicite, les latences de détection d'erreurs associées aux mécanismes TIMEOUT, DEADLINE et ARRIVAL étant très dispersées.

4.4. Coût en espace et en temps des MDEs

La figure 5 détaille les coûts en espace et en temps d'exécution des MDEs. Ces coûts ont été mesurés en ajoutant les MDEs un par un, par ordre décroissant de pourcentages de premières détections (voir figure 2). Le coût mémoire du noyau Chorus (196Ko à 191Ko selon que l'on utilise ou non des espaces d'adressage séparés) ne sont pas indiqués sur la figure 5.
La partie gauche de la figure détaille l'occupation mémoire additionnelle occasionnée par l'ajout de chaque MDE. La taille de code du support d'exécution (Chorus exclus) est multipliée par un facteur 6.7 lors de l'introduction de tous les MDEs. Le mécanisme VMEMORY multiplie le volume de code par un facteur 3.5. Ceci s'explique par la duplication du code de certaines bibliothèques en mémoire (e.g. libc), cette duplication étant due au fait que la version de Chorus que nous utilisons n'offre pas de support direct pour les librairies partagées. Le mécanisme STRUCTURE ajoute également un surcoût en espace significatif. Ce surcoût provient du fait de l'utilisation massive de fonctions C *inline* pour la gestion des structures

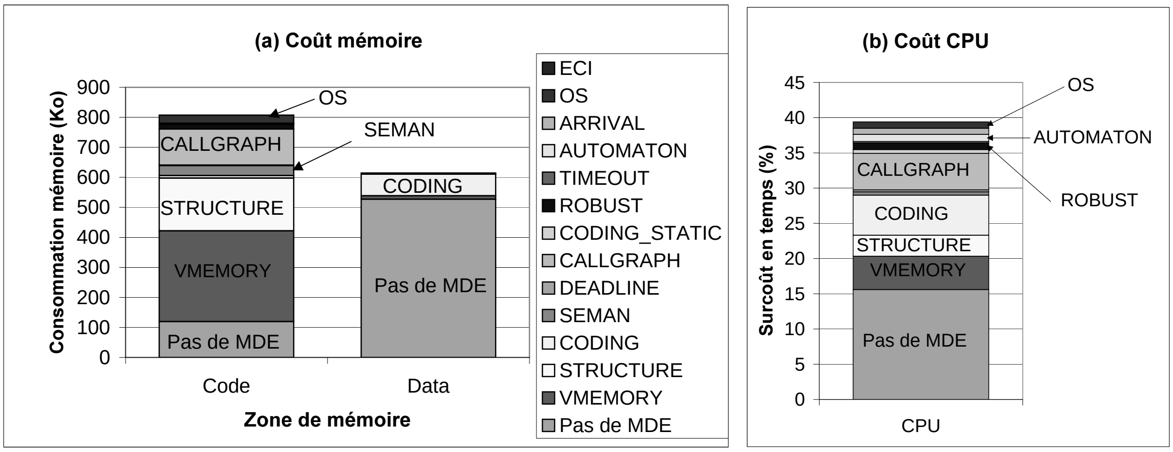

FIG. 5 – *Coûts mémoire et CPU des MDEs*

de données pour des raisons de performances. Le mécanisme CALLGRAPH augmente la taille du code de manière non négligeable. Cette augmentation est due à la présence dans HADES d'un grand nombre de fonctions de petites tailles, introduites pour des raisons de modularité. Concernant l'augmentation de la taille des données, seul le mécanisme CODING occasionne une augmentation perceptible du volume de données. Tous les autres mécanismes utilisent des variables allouées en pile, ce qui n'a pas d'impact sur la taille de la zone de données du support d'exécution.

Le surcoût en temps des MDEs (pourcentage de temps passé à s'exécuter dans le support d'exécution) est donné sur la partie droite de la figure 5. Le temps total passé dans le support d'exécution est multiplié par un facteur 2.5 par l'introduction de mécanismes de détection d'erreurs. Les MDEs les plus consommateurs de temps CPU sont, par ordre décroissant de consommation CPU : CODING (14.4%), CALLGRAPH (13.1%), VMEMORY (12.0%) et STRUCTURE (7.5%). Les autres MDEs ont un coût inférieur à 3%. Si l'on met en regard le pourcentage de premières détections des mécanismes et leur coût processeur, les mécanismes SEMAN, VMEMORY, DEADLINE, TIMEOUT, et STRUCTURE offrent le meilleur compromis efficacité/coût.

Un facteur 2.5 est à notre avis plutôt satisfaisant pour des MDEs entièrement implantés par logiciel. Une alternative aurait été d'utiliser de la redondance temporelle pour les activités du support d'exécution (les exécuter successivement plusieurs fois et comparer leurs résultats pour détecter les erreurs). Néanmoins l'utilisation de redondance temporelle pour des activités système est non trivial à implanter, les résultats de ce type d'activités étant principalement des effets de bord (entrées/sorties, modification de structures de données comme les files d'exécution). Par ailleurs, nous ne sommes pas convaincus que le coût induit par l'utilisation d'une telle technique soit moins important que celui de nos mécanismes, à cause de la difficulté pour comparer les résultats des exécutions successives.

Afin de diminuer le coût induit par la détection d'erreurs, nous avons analysé les redondances entre MDEs. Nous avons ainsi montré qu'en éliminant certains d'entre eux, il est possible de diminuer les coûts en mémoire et processeur (x1.1 et x1.7 respectivement) avec seulement une faible diminution de la couverture du silence sur défaillances (-0.8%). Plus de détails sur ces résultats pourront être trouvés dans [4].

5. Travaux apparentés

Il existe un large éventail de méthodes de détection d'erreurs, que ces dernières soient implantées par matériel ou par logiciel. Elles peuvent être classées entre deux catégories : (i) les méthodes utilisant de la *redondance*, au niveau des données ou des calculs (e.g. codes détecteurs/correcteurs d'erreurs, réplication spatiale ou temporelle de processus, structures de données robustes) ; (ii) les méthodes vérifiant le *comportement* des programmes (e.g. vérification de flot de contrôle, chiens de charge, contrôles sémantiques). Dans l'étude qui est présentée dans ce document, nous utilisons les deux catégories de mécanismes. Par ailleurs, pour des raisons de coût, nous nous sommes limités à l'utilisation de mécanismes réalisés par

logiciel, sans modification du système d'exploitation ou du compilateur, en les sélectionnant pour qu'ils aient un coût faible et borné du fait du contexte temps-réel strict.
L'injection de fautes a été utilisée dans de multiples études pour évaluer la sûreté de fonctionnement des systèmes logiciels ou matériels. Certaines d'entre elles évaluent les modes de défaillances des systèmes de manière générale, alors que d'autres se concentrent sur la validation de l'hypothèse de silence sur défaillance.
Dans la première classe de travaux, l'étude décrite par Damm dans [6] compare les performances d'un ensemble de MDEs intégrés dans un système d'exploitation temps-réel en présence de fautes physiques, la métrique de performance permettant de comparer les MDEs étant le pourcentage de premières détections. Contrairement aux résultats de notre étude, Damm observe un plus fort pourcentage de détections par les MDEs matériels. En revanche, les deux études concordent dans l'observation d'une faible efficacité des mécanismes de vérification temporelle (moins de 4% dans les deux cas). Salles et al [14] proposent une méthodologie (utilisation de réflexivité et de modules d'emballage – *wrappers*) pour le confinement des effets de fautes internes et externes pouvant affecter les micro-noyaux COTS. Un ensemble de modules d'emballage ont été construits au dessus de Chorus. La couverture du silence sur défaillance résultante, obtenue par injection de fautes (par logiciel) dans le code de Chorus, est de 97.7% (injection dans l'ordonnanceur) à 99.5% (injection dans le code de synchronisation). Contrairement à cette étude, notre travail prend en compte les systèmes distribués, et donne non seulement l'efficacité, mais aussi les coûts des MDEs mis en place. Koopman et al. [9] s'attachent à évaluer la robustesse de systèmes POSIX (et plus récemment Windows-NT) en présence de fautes externes, modélisées par des paramètres d'entrée incorrects. Les résultats de cette étude sont difficilement comparables aux nôtres car le modèle de fautes considéré est différent (fautes d'origine logicielle dans un cas, et d'origine physique dans le nôtre). Dans un autre domaine d'application, très récemment, une évaluation de robustesse de système de gestion de base de données COTS a été effectuée, en présence de fautes d'origine matérielle et logicielle [5].
Les études présentées dans [11, 7, 13] se concentrent sur la couverture l'hypothèse de silence sur défaillance en présence de fautes d'origine physique. Madeira et Silva [11] évaluent la couverture du silence sur défaillances dans les systèmes sans redondance ni de matériel, ni de logiciel. Les seuls MDEs utilisés (en plus de ceux embarqués dans le processeur) sont des mécanismes vérifiant le comportement des exécutions (ECI, vérification de flot de contrôle par matériel dédié). La couverture, obtenue par injection de fautes au niveau physique est de 98.1% sur une architecture 68K, et de 99.6% sur un Z80. Une étude effectuée dans le même laboratoire, et cette fois en n'utilisant pas de matériel dédié a montré une couverture du silence sur défaillance de 98.2% sur un processeur 68K [13]. Les résultats quantitatifs de ces deux études sont difficiles à comparer aux nôtres car les cibles de l'évaluation sont très différentes : des applications orientées *données* dans un cas (calcul numérique), et des applications orientées *contrôle* dans l'autre (système d'exploitation). Fuchs [7] a évalué la couverture du silence sur défaillance de l'architecture temps-réel Mars, qui inclut des MDEs eu niveau du matériel (processeur et matériel dédié), du système d'exploitation (assertions) et des applications (CRC, redondance temporelle). Une couverture de 92% du silence sur défaillance est observée en utilisant les MDEs matériels, la couverture montant à 100% quand tous les MDEs sont utilisés. Comparé à notre travail, cette étude démontre une bien meilleure couverture du silence sur défaillance, mais en revanche impose l'utilisation de matériel dédié à la tolérance aux fautes.

6. Conclusion

Nous avons présenté dans cet article une évaluation du comportement de calculateurs exécutant un support d'exécution temps-réel tolérant aux fautes en présence de fautes physiques. Ce support d'exécution inclut des mécanismes de détection d'erreurs (MDEs) entièrement implantés par logiciel, utilisés pour garantir le silence sur défaillance des calculateurs. La couverture du silence sur défaillance a été évaluée par une méthode logicielle d'injection de fautes.
Le principal résultat de notre étude est que la couverture du silence sur défaillance est au mieux de 99.1%. Comparée à la couverture initiale de 80.6 %, obtenue quand aucun MDE n'est utilisé, ces résultats sont très bons, puisque le nombre de violations du silence sur défaillance est divisé par un facteur 22. Ils ne sont toutefois pas suffisants à notre avis pour supporter l'exécution d'applications à sûreté critique.
Concernant le coût en mémoire et processeur des MDEs, le pourcentage de temps passé à exécuter le support d'exécution est multiplié par 2.5 suite à l'introduction de détection d'erreurs, et sa taille mémoire

est multipliée par un facteur 1.8. Par ailleurs, la latence de détection des erreurs est en moyenne de 7.8 ms. Les coûts induits par la détection d'erreur peuvent toutefois être diminués de manière significative avec seulement un très léger infléchissement de la couverture du silence sur défaillances [4].

Remerciements

Ce travail a été partiellement financé par la Direction Générale de l'armement (DGA/DSP), #98.34.375. Nous tenons à remercier David Powell ainsi que les relecteurs de l'article pour leurs remarques avisées sur des versions préliminaires de l'article.

Bibliographie

1. Anceaume (E.), Cabillic (G.), Chevochot (P.), Colin (A.), Decotigny (D.) et Puaut (I.). – Un support d'exécution flexible pour applications distribuées temps-réel dur. *In : Première Conférence française sur les systèmes d'exploitation*, pp. 109–120. – Rennes, France, juin 1999.
2. Barton (J. H.), Czeck (E. W.), Segall (Z. Z.) et Siewiorek (D. P.). – Fault injection experiments using FIAT. *IEEE Transactions on Computers*, vol. 39, n4, avril 1990, pp. 575–581.
3. Carreira (João), Madeira (Henrique) et Silva (João Gabriel). – Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, vol. 24, n2, février 1998, pp. 125–136.
4. Chevochot (P.) et Puaut (I.). – Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from COTS components. *In : Proc. of the 2001 International Conference on Dependable Systems and Networks*. – Göteborg, Sweden, juillet 2001.
5. Costa (D.), Rilho (T.) et Madeira (H.). – Joint evaluation of performance and robustness of a COTS DBMS through fault injection. *In : Proc. of the 2000 International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, pp. 251–260. – New York, USA, juin 2000.
6. Damm (A.). – The effectiveness of software error-detection mechanisms in real-time operating systems. *In : Proc. of the 16th International Symposium on Fault-Tolerant Computing Systems*, pp. 171–176. – Vienna, Austria, juin 1986.
7. Fuchs (E.). – An evaluation of the error detection mechanisms in MARS using software-implemented fault injection. *In : Proceeding of the 2nd European Dependable Computing Conference*, éd. par Springer-Verlag, pp. 73–90. – Taormina, Italy, octobre 1996.
8. Kanawati (G.A.), Kanawati (N.A.) et Tang (D.). – FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, vol. 44, n2, 1995, pp. 248–260.
9. Koopman (P.) et DeVale (J.). – Comparing the robustness of POSIX operating systems. *In : Proc. of the 29th International Symposium on Fault-Tolerant Computing Systems*, pp. 30–37. – Madison (WI), juin 1999.
10. Lala (P.K.). – *Fault Tolerant and Fault Testable Hardware Design*. – Prentice Hall, 1985.
11. Madeira (H.) et Silva (J. G.). – Experimental evaluation of the fail-silent behavior in computers without error masking. *In : Proc. of the 24th International Symposium on Fault-Tolerant Computing Systems*, pp. 350–359. – Austin, USA, juin 1994.
12. Mahmood (A.) et McCluskey (E. J.). – Concurrent error detection using watchdog processors - a survey. *IEEE Transactions on Computers*, vol. 37, n2, février 1988, pp. 160–174.
13. Rela (M. Z.), Madeira (H.) et Silva (J. G.). – Experimental evaluation of the fail-silent behavior in programs with consistency checks. *In : Proc. of the 26th International Symposium on Fault-Tolerant Computing Systems*, pp. 394–403. – Sendai, Japan, juin 1996.
14. Salles (F.), Moreno (M. Rodriguez), Fabre (J.C.) et Arlat (J.). – Metakernels and fault containment wrappers. *In : Proc. of the 29th International Symposium on Fault-Tolerant Computing Systems*, pp. 22–29. – Madison (WI), juin 1999.
15. Steininger (A.) et Scherrer (C.). – On finding an optimal combination of error detection mechanisms based on results of fault injection experiments. *In : Proc. of the 27th International Symposium on Fault-Tolerant Computing Systems*, pp. 238–247. – Seattle, WA, USA, juin 1997.