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

Analyse de temps d'exécution au pire cas du système d'exploitation temps-réel RTEMS

Antoine Colin, Isabelle Puaut

IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cédex, FRANCE
Mél: acolin/puaut@irisa.fr

Résumé
Dans les systèmes temps réel strict, la violation d'une contrainte temporelle peut avoir des conséquences catastrophiques. Il est donc incontournable dans de tels systèmes d'obtenir les temps d'exécution au pire cas (TEPCs) de tous les logiciels s'exécutant (logiciel applicatif et système d'exploitation). Traditionnellement, les TEPCs des systèmes d'exploitation temps-réel sont obtenus par un test exhaustif du système d'exploitation, en essayant de reproduire au mieux le pire scénario d'exécution. Cet article présente une approche alternative, qui consiste à utiliser une méthode d'analyse statique de programmes pour obtenir, hors-ligne, le TEPC des appels système d'un noyau temps-réel, le noyau RTEMS [14]. Nous présentons les résultats qualitatifs et quantitatifs de l'analyse statique de RTEMS, et concluons sur les possibilités d'utilisation de méthodes d'analyse statique de TEPC pour analyser du code "système".

Mots-clés : Systèmes d'exploitation temps-réel, temps d'exécution au pire cas, analyse statique

---

1. Introduction

Les systèmes temps-réel diffèrent des autres systèmes par un critère de correction de leurs applications plus strict. Dans de tels systèmes, la correction d'une application ne dépend pas uniquement du résultat délivré mais aussi de la date à laquelle il est produit. Dans le cas des systèmes temps-réel *strict* (ou *dur*), le dépassement de l'échéance d'une tâche peut avoir des conséquences catastrophiques. Ainsi, il est fortement recommandé que de tels systèmes utilisent des méthodes *d'analyse d'ordonnançabilité* pour vérifier, avant l'exécution, que toutes les échéances seront respectées. De telles méthodes nécessitent, entre autres, la connaissance du *Temps d'Exécution au Pire Cas* (TEPC) de chaque portion du code, *i.e.* le temps au pire nécessaire pour l'exécuter sur une architecture donnée. Les TEPCs peuvent être obtenus soit par test, en portant une attention particulière aux conditions de test afin de reproduire le pire scénario d'exécution, ou bien par *analyse statique de TEPC* pour laquelle le TEPC d'une portion de code est obtenu par analyse statique du code source du programme.
La figure 1, extraite de [2], présente les 3 niveaux classiques nécessaires à l'analyse d'ordonnançabilité d'un système temps-réel strict.

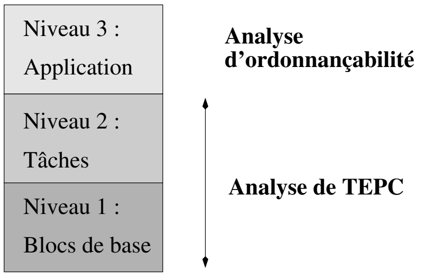


FIG. 1 – *Les différents niveaux de l'analyse d'ordonnançabilité*

– **Niveau 1 - Analyse du TEPC des blocs de base.** Le code d'un programme peut être fragmenté en plusieurs blocs de base. Un bloc de base ne comporte qu'un seul point d'entrée, qu'un seul point de sortie, et aucune boucle, ni branchement ou appel de fonction. Les blocs de base n'utilisent aucune ressource autre que le processeur. À ce niveau, l'analyse statique de TEPC donne une estimation du TEPC du bloc de base en effectuant la *somme* des temps d'exécution au pire cas des instructions le composant. La *somme* est ici l'assemblage des TEPCs des instructions en prenant en considération la micro-architecture du processeur (*e.g.* les caches qui modifient le temps de chargement des instructions et les pipelines qui introduisent du parallélisme entre les exécutions d'instructions successives).
– **Niveau 2 - Analyse du TEPC des tâches.** À ce niveau, l'analyse statique de TEPC calcule le temps nécessaire à l'exécution d'une tâche isolée, et ce à partir des TEPCs des blocs de base composant la tâche. C'est à ce niveau que sont prises en compte les possibilités de rupture de séquence dans le code d'une tâche (conditionnelles, itérations, etc).
– **Niveau 3 - Analyse d'ordonnançabilité.** À ce niveau, plusieurs tâches s'exécutent sur le processeur et sont en compétition pour l'utilisation des ressources. L'analyse d'ordonnançabilité prend en considération un ensemble de caractéristiques connues statiquement sur les tâches (périodicité, TEPCs, échéances, synchronisations et communications entre tâches) pour décider si toutes les contraintes temporelles des tâches seront respectées. Un exemple classique d'analyse d'ordonnançabilité est l'analyse Rate Monotonic [12].

Cet article porte sur les 2 premiers niveaux de l'analyse d'ordonnançabilité, que nous nommons par la suite simplement *analyse statique de TEPC*.

1.1. Travaux apparentés

Systèmes d'exploitation temps-réel

Les systèmes d'exploitation ayant des caractéristiques temps-réel peuvent être répartis en trois catégories principales : les extensions temps-réel aux systèmes d'exploitation à partage de temps (¨time-sharing¨), les noyaux ciblés pour les applications embarquées, à base d'ordonnancement à priorités fixes, et les systèmes d'exploitation du domaine de la recherche. Les systèmes d'exploitation de la première catégorie (les extensions temps-réel d'Unix par exemple) peuvent souffrir de problèmes, car certains de leurs mécanismes n'ont pas été conçus dans le but de faire fonctionner des applications temps-réel. Par exemple, la plupart des files d'attente internes est gérée selon une politique FIFO, y compris dans la version temps-réel du système, ce qui peut causer des délais longs et imprévisibles. L'objectif des systèmes de la seconde catégorie (e.g. pSOS, VxWorks, RTEMS, Chorus, Nucleus) est généralement d'offrir les meilleures performances possibles, en terme de temps de réponse moyen aux événements extérieurs. Certains d'entre eux fournissent des mécanismes permettant d'éviter le phénomène d'inversion de priorité (*e.g.* VxWorks, RTEMS). L'absence de système d'exploitation commercial ayant un comportement temporel hautement prévisible a encouragé les équipes de recherche à concevoir de nouveaux paradigmes de systèmes d'exploitation (la troisième catégorie de systèmes mentionnée ci-dessus). Les systèmes de cette catégorie sont par exemple RT-Mach, Mars et Spring.
En ce qui concerne la connaissance du temps de réponse des appels système, la plupart des systèmes d'exploitation temps-réel (principalement commerciaux) n'est fournie qu'avec le temps d'exécution moyen de leurs appels système, obtenus par tests et mesures (*e.g.* Chorus [7], RTEMS [14]). Quelques uns (*e.g.* Nucleus [9]) fournissent des temps d'exécution obtenus par des tests soigneusement conçus pour être représentatifs des pires circonstantes d'utilisation du système d'exploitation. Bien que cette approche soit meilleure que la précédente pour une utilisation dans un cadre temps-réel strict, elle est sujette aux erreurs. En effet, l'identification du pire scénario d'exécution reste totalement à la charge du concepteur du système d'exploitation qui peut commettre certaines erreurs dues à la complexité du code manipulé.

Analyse statique de TEPC

Alors que l'analyse temporelle est un domaine traditionnel de la recherche en système temps-réel, l'analyse statique de TEPC (*static WCET analysis* en anglais) n'a attiré l'attention de la communauté qu'il y a seulement dix ans. Malgré tout, des progrès substantiels ont été réalisés dans ce domaine durant la dernière décennie (voir [15] pour un état de l'art). L'analyse statique de TEPC repose sur deux problèmes principaux : la caractérisation des chemins d'exécution possibles, et le calcul du temps d'exécution d'un

chemin d'exécution donné d'une manière sûre et précise en considérant la micro-architecture du processeur. Les éléments d'architecture étudiés à ce jour sont les pipelines [18, 8], les caches [11, 13], et la prédiction de branchement [3].
La cible la plus courante pour l'analyse statique de TEPC est le code utilisateur. De plus, la plupart des travaux publiés dans le domaine de l'analyse statique de TEPC sont évalués en utilisant des petites applications numériques, exception faite du travail décrit dans [6], qui étudie les propriétés de code applicatif embarqués complexe (dans les domaines des télécommunications, du contrôle de véhicule, et de l'électronique grand public). À notre connaissance, aucune autre étude ne s'est encore attachée à utiliser une méthode d'analyse statique de TEPC sur le code d'un système d'exploitation.

1.2. Contribution

Cet article présente les résultats de l'analyse statique du noyau RTEMS [14] par un analyseur statique de TEPC de notre conception (Heptane [4]), réalisé dans le but d'obtenir le TEPC de ses appels système.
Une difficulté majeure de l'analyse statique du code source d'un système d'exploitation est la présence de code ¨bas niveau¨ pour la gestion du matériel sous-jacent (processeur, périphériques). Ce type de code peut rendre le code difficilement analysable par la plupart des analyseurs statiques de TEPC actuels (présence de code assembleur, utilisation de pointeurs, etc). Le principal intérêt de notre étude est la mise en évidence des problèmes rencontrés et des solutions adoptées pour les résoudre. Un résultat intéressant est que des modifications mineures du code source de RTEMS ont permis de rendre analysable le cœur de RTEMS, à savoir le code relatif à la gestion des tâches et des sémaphores.
Un autre résultat de notre étude est l'identification des conditions expérimentales permettant d'exhiber le pire temps d'exécution des appels systèmes de RTEMS (e.g. le contenu de la file des processus RTEMS, les contenus des structures de données du mécanisme d'allocation dynamique). Ces conditions peuvent à terme être transposées à d'autres systèmes pour générer de manière plus systématique des benchmarks “pire cas” de systèmes d'exploitation temps-réel.

1.3. Plan de l'article

La suite de cet article détaille les résultats obtenus et est organisée comme suit. La section 2 présente le noyau temps-réel RTEMS. La section 3 décrit brièvement l'outil d'analyse statique de TEPC Heptane. Les résultats qualitatifs et quantitatifs de l'analyse de TEPC de RTEMS sont détaillés dans la section 4. Enfin, nous concluons en section 5 par un récapitulatif des résultats obtenus, et des perspectives de travaux futurs.

2. Le noyau temps-réel RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) [14] est un micro-noyau temps-réel pour applications embarquées, dont les sources sont publics. RTEMS comporte les éléments suivants :

- des capacités multi-tâches,
- un ordonnancement préemptif, à base de priorités,
- la communication et la synchronisation entre tâches,
- l'héritage de priorité pour éviter les problèmes d'inversions de priorités,
- l'allocation dynamique de la mémoire,
- des capacités de configuration.

RTEMS offre deux interfaces de programmation : une interface conforme à la norme POSIX, et une interface spécifique RTEMS. Nous nous concentrons ici sur l'interface RTEMS. Cette dernière comporte 85 appels système (appelés *directives* dans la terminologie RTEMS) regroupés en 17 modules (initialization, task, interrupt, clock, timer, semaphore, message, event, signal, partition, region, dual ported memory, I/O, fatal error, rate monotonic, user extensions, et multiprocessing). Tous les éléments gérés par RTEMS, tels que les tâches, les sémaphores et les messages (i.e. les *objets* dans la terminologie RTEMS) sont nommés et alloués de la même façon.

3. L'analyseur de temps d'exécution au pire cas HEPTANE

HEPTANE [4] (*Hades Embedded Processor Timing ANalyzEr*) fait partie de l'environnement HADES [1] en cours de développement à l'IRISA. HEPTANE analyse des programmes écrits en C ANSI, et fournit des informations sur leur temps d'exécution sur un processeur Pentium à 90Mhz (P54C stepping B5). HEPTANE utilise deux niveaux de représentation des programmes : l'arbre syntaxique, obtenu par analyse du code source, et le graphe de flot de contrôle, obtenu par l'analyse du code assembleur du programme.

3.1. Recherche des chemins d'exécution

Le calcul du TEPC d'un programme par HEPTANE repose principalement sur la syntaxe du programme. Le calcul est effectué grâce à un système d'équations temporelles semblables à celles proposées par Puschner et Koza dans [16]. Par exemple, l'équation simplifiée pour le calcul du TEPC d'une structure conditionnelle *"CO"* dont le code est *"if (C) S; else E;"*, sera *"TEPC(CO) = TEPC(C) + max(TEPC(S),TEPC(E))"*. Le TEPC est calculé de manière récursive en respectant l'arbre syntaxique du programme.

Le calcul du TEPC d'une boucle nécessite la connaissance de son nombre maximum d'itérations. HEPTANE suppose que cette information est fournie par l'utilisateur sous forme d'annotations présentes dans le code source du programme. Ces annotations permettent de prendre en compte les boucles imbriquées dont le nombre d'itérations dépend des valeurs des variables de boucle des boucles englobantes (cf. [3]).

Quelques restrictions sur le langage source, classiques dans le domaine de l'analyse statique de TEPC, sont nécessaires pour pouvoir identifier le pire chemin d'exécution dans le programme. Tout d'abord, l'utilisation de pointeurs sur fonctions est interdite afin que tous les appels de fonctions soit identifiables statiquement. De plus, tout ce qui peut conduire à un graphe de de flot de contrôle *non structuré* (*i.e.* un graphe de flot de contrôle incompatible avec l'arbre syntaxique) est interdit. Par exemple, les *goto* et l'inclusion de code assembleur contenant des instructions de branchement ne sont pas autorisés.

3.2. Modélisation de l'architecture matérielle

HEPTANE inclut une modélisation de l'architecture matérielle cible, ce qui permet de réduire le pessimisme de l'analyse statique de TEPC. HEPTANE comporte des mécanismes permettant la prise en compte des effets du cache d'instruction, du pipeline et de la prédiction de branchement présents dans les processeurs Pentium. La modélisation du flot d'instructions au travers du pipeline permet de simuler son effet. La prise en compte du cache d'instructions repose sur la technique dite de simulation statique de cache [13] : chaque instruction est classée en fonction de son comportement par rapport au cache d'instruction. Le classement des instructions nécessite à la fois l'arbre syntaxique et le graphe de flot de contrôle du programme. Une technique similaire permet de prendre en compte les effets du mécanisme de prédiction de branchement (cf. [3] pour plus de détails).

4. Résultats de l'analyse de RTEMS

Les conditions expérimentales sont décrites au paragraphe 4.1. Puis, nous présentons les résultats qualitatifs de cette étude en énumérant toutes les difficultés rencontrées lors de l'analyse : graphes de flot de contrôle non structurés (§ 4.3), appels de fonctions (§ 4.4), boucles (§ 4.5), appels bloquants et changements de contexte (§ 4.6), et interruptions (§ 4.7). Enfin, nous présentons les résultats quantitatifs de l'analyse (§ 4.8).

4.1. Conditions expérimentales

Cette étude porte sur le code source de RTEMS 4.0.0 pour une architecture i386. On suppose une architecture mono-processeur (le code source a été spécialisé, quand cela était nécessaire, pour supprimer le code relatif à la gestion des multiprocesseurs). Nous présentons ici les résultats de l'étude de 12 directives clés de gestion des tâches et de synchronisation du noyau RTEMS (voir tableau ci-dessous).

Nom	Description
rtems_clock_get	Informations de date et heure du système
rtems_semaphore_create	Création de sémaphore
rtems_semaphore_obtain	Prise de sémaphore
rtems_semaphore_release	Libération de sémaphore
rtems_task_create	Création de tâche
rtems_task_delete	Destruction de tâche
rtems_task_ident	Renvoi de l'identifiant d'une tâche
rtems_task_start	Démarrage d'une tâche
rtems_task_suspend	Suspension d'une tâche
rtems_task_resume	Reprise d'une tâche
rtems_task_set_priority	Affectation d'une priorité à une tâche
rtems_task_wake_after	Réveil d'une tâche après un certain délai

Le code source des 12 directives analysées est réparti entre 91 fichiers sources dont : 28 fichiers C, un fichier assembleur, 38 fichiers de définition de constantes, de macro-commandes et de types, et 24 fichiers de fonctions "en-ligne". Le code source (hors commentaires) comporte 14532 lignes, et définit 355 fonctions.

4.2. Problèmes posés par la structure modulaire de RTEMS

Dans une large majorité des cas (89%), les fonctions appelées et appelantes sont déclarées dans des fichiers différents. Ainsi, pour être facilement utilisable, un outil d'analyse statique de TEPC devrait pouvoir traiter des fonctions dispersées dans différents fichiers et répertoires. Comme HEPTANE ne travaille actuellement que sur un fichier source unique, on doit au préalable opérer la fusion de tous les fichiers requis pour l'analyse de chacune des directives.

En outre, nous avons observé que la plupart des fichiers sont utilisés pour l'analyse de plusieurs directives de RTEMS. En moyenne, un fichier de code (*c.-à-d.* un fichier contenant du code de C ou des fonctions "en-ligne") est utilisé pour l'analyse de 6 directives. Par conséquent, puisque l'analyse statique de TEPC peut être longue (jusqu'à 10 minutes au pire pour une directive de RTEMS), nous pensons qu'il serait avantageux de pouvoir garder des résultats partiels de TEPC afin de les réutiliser plus tard. Une question importante est alors celle du choix des informations qui doivent constituer un résultat partiel de TEPC permettant l'obtention de TEPCs qui ne soient pas trop pessimistes.

4.3. Graphe de flot de contrôle non structurés

Un seul fichier source assembleur a été nécessaire pour l'analyse des douze directives choisies. Il contient le code de sauvegarde et de restauration du contexte d'une tâche (*c.-à-d.* sauvegarder et restaurer les registres). Ce code n'a posé aucun problème d'analyse car il ne contient pas d'instructions de branchement.

Quatre boucles contenant des *goto* ont été rencontrées pendant l'étude des directives de RTEMS. Dans tous les cas, il a été possible de restructurer ces boucles.

4.4. Appels de fonction

Appels dynamiques de fonction

Les appels de fonctions via l'utilisation de pointeurs (ci-après appels *dynamiques* de fonctions) posent problème pour l'analyse statique de TEPC car la fonction effectivement appelée n'est connue qu'à l'exécution. Parmi les 368 appels de fonctions présents dans les 12 directives étudiées, nous n'avons trouvé que 15 appels dynamiques de fonctions, répartis en deux catégories :

- *Appels des extensions utilisateurs.* RTEMS permet aux concepteurs d'applications d'ajouter des routines d'extension au noyau. Ces routines seront appelés lors de l'occurrence d'événements spécifiques du système, comme par exemple la création de tâches ou le changement de contexte. Comme ces routines d'extension peuvent être ajoutées dynamiquement, on doit apporter une attention particulière à l'utilisation des directives de RTEMS impliquant des extensions utilisateur. Nous aurions pu interdire l'utilisation de ces extensions. L'approche que nous avons préférée consiste à demander aux concepteurs d'applications, avant exécution, une identification de toutes les routines d'extension à prendre en compte. Notons que RTEMS lui-même utilise ces extensions pour effectuer certains traitements lors de la création, de la suppression, du démarrage et du redémarrage des tâches. Heureusement, les extensions ajoutées par le noyau sont connues avant exécution, et les appels dynamiques ont pu être aisément remplacés par des appels statiques.

– *Appels des extensions d'API.* L'interface POSIX et l'interface RTEMS reposent toutes deux sur une couche logicielle commune appelée *score*. Les extensions d'API permettent à une API donnée de fournir des routines d'extension qui seront invoquées par la couche *score* à certains points spécifiques : après une commutation de contexte, avant/après un appel à un gestionnaire de périphérique. Heureusement, pour une API donnée (ici l'API RTEMS) les fonctions appelées sont connues statiquement.

Nous avons retrouvé la même situation concernant les appels dynamiques de certaines fonctions dépendant de l'architecture. Dans le cas de l'architecture i386, on rencontre deux de ces fonctions appelés respectivement lors de l'allocation et de la libération d'une pile. Puisque l'architecture cible est connue statiquement, ces appels dynamiques ont pu être aisément remplacés par des appels statiques.

Appels récursifs
Pour certains outils d'analyse statique de TEPC, la présence de récursivité dans le code peut rendre problématique l'identification du chemin d'exécution au pire cas. Aucune fonction récursive n'a été rencontrée parmi les fonctions analysées durant notre étude. Cette conclusion diffère de celle donnée dans [6], qui se base sur l'étude de code applicatif (dans cette étude, 0.3% des fonctions sont récursives).

4.5. Les boucles

L'analyse des boucles est d'une grande importance pour l'analyse statique de TEPC. Elle permet d'identifier le pire chemin d'exécution dans le programme. Le code analysé contient relativement peu de boucles (seulement 21 boucles pour 14532 lignes de code analysées).

Boucles emboîtées
Les boucles emboîtées (boucles dont le nombre d'itérations dépend des compteurs des boucles l'englobant) peuvent être un problème pour l'analyse statique de TEPC, car une évaluation pessimiste du nombre maximum d'itérations d'une boucle emboîtée peut avoir une influence importante sur le TEPC estimé d'un programme [3]. Nous n'avons rencontré aucune boucle de ce type dans le code analysé. Là encore, nos conclusions diffèrent de celles données dans [6], rapportant une moyenne de 9% de boucles emboîtées.

Boucles sans fin
Une seule boucle sans fin a été rencontrée dans le code source de RTEMS. C'est le corps de la tâche *idle*, qui n'est exécutée que lorsqu'aucune autre tâche n'est prête à être exécutée. La fonction dans laquelle cette boucle apparaît n'appartient pas au graphe d'appel des directives analysées, et n'a donc aucune influence sur leur TEPC.

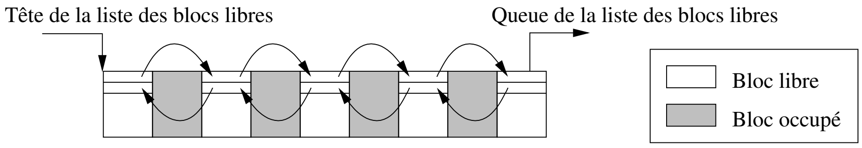
Détermination du nombre maximum d'itérations des boucles
Comme nous l'avons dit au § 3.1, HEPTANE nécessite que le code source soit annoté pour fournir le nombre maximum d'itérations pour chaque boucle. Bien que les boucles soient peu nombreuses, l'obtention du nombre maximum d'itérations des boucles n'a été aisé que pour quatre d'entre elles (boucles *for* dont les nombres maximum d'itérations étaient directement fournis par le test de boucle). Les 17 autres boucles ont exigé un examen plus fin du code source, dont nous présentons les points difficiles ci-dessous.

– *Gestion des noms d'objets.* Dans RTEMS, il existe deux types de nommage des objets : chaînes de caractères de type C ou chaînes de valeurs 32 bits. Deux ensembles de fonctions de gestion des noms (comparaison, copie et suppression de noms) sont fournis. Chacune de ces fonctions comporte une boucle dont le nombre d'itérations dépend de la longueur du nom. Les objets créés en utilisant l'API RTEMS (*e.g.* tâches, sémaphores) sont identifiés par des chaînes de valeurs 32 bits de longueur. De même, les noms créés par le noyau lui-même ne dépassent pas 32 bits. Ainsi, les boucles concernées par ce type de nom ont été annotées avec la valeur *un*.
– *Conversion des noms en identificateur.* La directive RTEMS *rtems_task_ident* fournit, pour un nom 32-bits donné, l'identificateur de la tâche associée. La recherche de l'identificateur de tâche est réalisée par une boucle qui balaye la table des tâches du système à la recherche du nom indiqué. Le nombre maximum d'itérations de cette boucle n'apparaît pas explicitement dans son test de fin d'itération. Cependant, la gestion de la table des tâches du système implique que cette boucle itère au pire *MAX_TASK* fois, où *MAX_TASK* est le nombre maximum de tâches présentes dans

le système à un moment donné. RTEMS impose que cette valeur soit connue statiquement : elle est imposée par la configuration de RTEMS utilisée pour la réservation statique de l'espace des descripteurs de tâche.

– *Gestion des extensions utilisateur.* Pour chaque événement auquel un ensemble d'extensions peut être associé (par exemple, la commutation de contexte), le code comporte une boucle pour l'appel de toutes ses extensions. Puisque le concepteur fournit la liste de ses extensions (cf § 4.4) et que le nombre d'extensions d'utilisateur ajoutées par RTEMS lui-même est connu, l'annotation de ces boucles est possible.

– *Gestion de la mémoire.* RTEMS utilise l'allocation dynamique de mémoire lors de la création des tâches et de leurs suppressions afin d'allouer et de libérer leur pile. RTEMS utilise la stratégie d'allocation de mémoire *first-fit* avec une taille minimum des blocs alloués de *MIN_BLOCK_SIZE*. Les blocs libres sont gérés par une liste doublement chaînée. La fonction d'allocation de mémoire contient une boucle qui effectue une recherche sur cette liste de blocs libres jusqu'à ce qu'un bloc dont la taille est au moins celle demandée soit trouvé. Dans le pire cas, la mémoire est constituée d'une succession de blocs libres et de blocs occupés de taille *MIN_BLOCK_SIZE* (voir la figure ci-dessous). Le pire nombre d'itérations de cette boucle est alors *HEAP_SIZE* / (2 * *MIN_BLOCK_SIZE*), où *HEAP_SIZE* est la taille de la zone mémoire consacrée à l'allocation dynamique.



Bien que l'on ait pu exhiber une borne supérieure pour le TEPC des routines d'allocation dynamique de mémoire, nous ne pouvons que constater que la stratégie d'allocation *first-fit* n'est pas adaptée à l'analyse statique de TEPC, et plus généralement, aux systèmes temps-réel strict. La surestimation du temps d'exécution de la fonction d'allocation dynamique a un impact important sur l'estimation du TEPC de la directive de création de tâche.

– *Gestion des files d'exécution.* L'attribution du processeur aux tâches par l'ordonnanceur de RTEMS est réalisée par un algorithme préemptif à base de priorité, les tâches de même priorité étant gérées par la méthode du tourniquet (*round-robin*). Le but de cet algorithme est de garantir que la tâche en cours d'exécution est à tout moment celle dont la priorité est la plus élevée parmi toutes les tâches prêtes. Le code de l'ordonnanceur comporte des boucles pour l'ajout des descripteurs des tâches dans la file d'exécution, selon leurs niveaux de priorité. Il parcourt la file d'exécution jusqu'à ce qu'un descripteur de tâche avec un niveau de priorité donné soit trouvé. Ces boucles itèrent au pire *MAX_READY_TASK* fois, où *MAX_READY_TASK* est le nombre maximum de tâches prêtes (au pire, *MAX_READY_TASK* est égal au nombre maximum des tâches *MAX_TASK*). On peut réduire le pessimisme si l'utilisateur connaît statiquement la répartition des tâches dans la file d'exécution.

– *Boucle principale de l'ordonnanceur.* La partie la plus difficile à analyser dans le code source de RTEMS a été le code de commutation de contexte (fonction *_Thread_Dispatch*, dont une vue simplifiée du code est présenté en figure 2).

  La fonction *_Thread_Dispatch* est appelée par toutes les directives et gestionnaires d'interruption de RTEMS qui peuvent modifier l'attribution de processeur. Chaque fois que les files d'exécution sont modifiées, la variable partagée *_Context_Switch_necessary* devient *VRAI* mais l'attribution du processeur n'est pas changée immédiatement. L'attribution du processeur est modifiée exclusivement par la fonction *_Thread_Dispatch*.

  Les interruptions sont masquées pendant le choix de la tâche à activer (zone de code **(I-dis1)**–**(I-ena1)**, **(I-dis2)**–**(I-ena1)** et **(I-dis2)**–**(I-ena2)**). Une variable appelée *_Thread_Dispatch_disable_level* permet de retarder la commutation de contexte demandée par les gestionnaires d'interruptions exécutés pendant la boucle de dispatch (*i.e.* pendant **(D-dis)**–**(D-ena)**). Chaque gestionnaire d'interruption qui souhaite modifier l'attribution du processeur ne fait appel à la fonction *_Thread_Dispatch* que si la variable *_Thread_Dispatch_disable_level* est nulle (*i.e.* si aucun changement de contexte n'est en cours). Le changement de contexte est alors effectué par la fonction interrompue *_Thread_Dispatch* par une itération supplémentaire de sa boucle principale.

```
void _Thread_Dispatch( void ) {

  Thread_Control *executing;   /* the thread which executes the dispatch */
  Thread_Control *heir;        /* the thread to switch to */

  executing = _Thread_Executing;
  _ISR_Disable( level );                      (I−dis1)
  while ( _Context_Switch_necessary == TRUE ) {

    heir = _Thread_Heir;
    _Thread_Dispatch_disable_level = 1;   (D−dis)
    _Context_Switch_necessary = FALSE;
    _Thread_Executing = heir; /* Select the thread to be dispatched */
    _ISR_Enable( level );                   (I−ena1)        /* Switch to the thread */
    _Context_Switch(&executing→Registers, &heir→Registers );
    executing = _Thread_Executing;
    _ISR_Disable( level );                  (I−dis2)
    }
    _Thread_Dispatch_disable_level = 0;     (D−ena)
    _ISR_Enable( level );                   (I−ena2)
  }
```

FIG. 2 – *Code du changement de contexte de RTEMS*

La figure 3 présente une exécution possible de la fonction *_Thread_Dispatch*. Lors de cette exécution, un processus *P1* demande un sémaphore déjà pris par un autre processus.

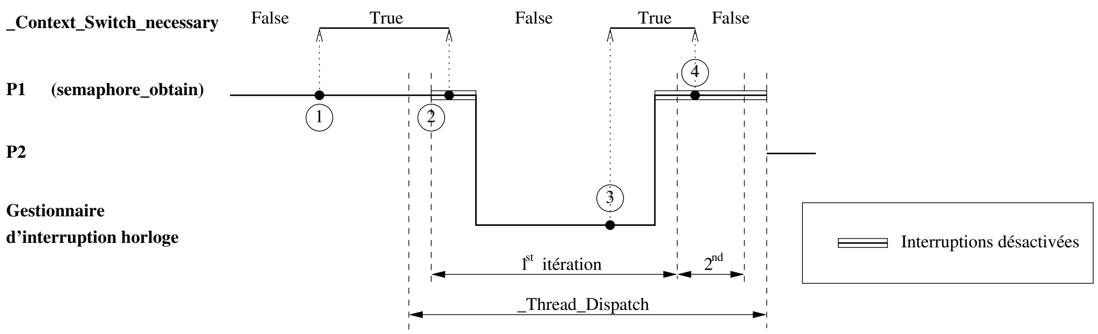


FIG. 3 – *Illustration of function* _Thread_Dispatch

Quand *rtems_semaphore_obtain* découvre que le sémaphore n'est pas libre, la file d'exécution de l'ordonnanceur ainsi que la file d'attente du sémaphore sont modifiées, et la variable *_Context_Switch_necessary* est placée à *VRAI* pour demander un changement de contexte (①). Puis, la fonction *_Thread_Dispatch* est appelée pour effectuer le changement de contexte, et re-boucle une première fois (②).

Supposons qu'une interruption d'horloge se produise pendant la première itération de la boucle et induise une modification de l'attribution du processeur à cause de l'expiration d'un quantum de temps (③). Puisque la variable *_Thread_Dispatch_disable_level* vaut un, le changement de contexte est retardé et sera effectué lors d'une deuxième itération de la fonction interrompue *_Thread_Dispatch* (④). La tâche dont la priorité est la plus élevée (ici *P2*) est alors exécutée.

Si la seule source d'interruption dans le système est l'interruption d'horloge, alors, grâce à l'intervalle de temps entre deux interruptions d'horloge, il ne peut y avoir au plus qu'une interruption d'horloge pendant la durée d'exécution de la fonction *_Thread_Dispatch*. Cette boucle peut alors être annotée avec la valeur 2. S'il existe d'autres sources d'interruption, l'estimation du nombre maximum d'itérations de cette boucle n'est possible que si l'on dispose de suffisamment d'informations pour majorer le nombre de demandes de changement de contexte effectuées par les gestionnaires d'interruption pendant l'exécution de la fonction *_Thread_Dispatch*. Pour cela, la connaissance du temps d'exécution du gestionnaire d'interruption, ainsi que l'intervalle minimum entre deux interruptions successives suffisent.

4.6. Gestion des appels bloquants et des changements de contexte

L'analyse d'ordonnançabilité nécessite l'identification exhaustive des blocages potentiels des tâches ainsi qu'une estimation de la durée de ces blocages.

La détermination des durées de blocage des tâches requiert une connaissance statique de l'ensemble des tâches à exécuter et de leurs synchronisations. Comme indiqué au paragraphe 1, la détermination des durées de blocages est du ressort de l'analyse d'ordonnançabilité et non de l'analyse statique de TEPC (qui analyse du code ne comportant pas de synchronisation interne). Nous avons dû simplement vérifier que le noyau ne fait pas appel à des primitives bloquantes, c'est à dire qu'il n'utilise ni sémaphore ni événement pour synchroniser les accès concurrents à ses structures de données internes. Cette vérification a été réalisée par une analyse manuelle du code source du noyau. Le contrôle des accès concurrents aux structures de données de RTEMS ne fait pas appel à des primitives bloquantes, mais utilise la désactivation des interruptions.

En ce qui concerne les directives de RTEMS qui peuvent provoquer un changement de contexte (*par exemple rtems_semaphore_obtain*), la structure du code de changement de contexte (fonction *_Thread_Dispatch* de la figure 2) impose que l'outil d'analyse statique de TEPC prenne en compte le temps nécessaire à la sauvegarde et à la restauration du contexte de la tâche (fonction *_Context_Switch* dans le code de *_Thread_Dispatch*).

Le TEPC d'une directive comprend donc le temps requis pour passer à une autre tâche dans le cas où la tâche appelante se bloque.

Cependant, lors de l'analyse statique de TEPC, l'analyseur ne sait pas si *_Context_Switch* effectue réellement le changement de contexte car il ne connaît pas la sémantique du code analysé. Il considère donc que *_Context_Switch* est une fonction ordinaire et ne prend pas en compte le surcoût dû au cache, à la prédiction de branchement et au pipeline lors du changement de contexte. Au pire, il faut supposer que le cache d'instructions et le tampon de cibles de prédiction de branchement sont complètement vidés à chaque changement de contexte ; le surcoût correspondant doit être ajouté au TEPC de la fonction *_Context_Switch* calculé par l'analyseur de TEPC.

4.7. Influence des interruptions

Une interruption peut se produire à presque n'importe quel moment de l'exécution d'une tâche. La présence d'interruptions a une influence directe sur le TEPC d'une tâche, car elle conditionne le nombre d'itérations de la boucle principale de la fonction *_Thread_Dispatch* (voir § 4.5). C'est l'unique impact des interruptions sur l'analyse statique de TEPC, et toute autre influence concerne l'analyse d'ordonnançabilité.

Pour pouvoir prendre en compte les interruptions lors de l'analyse d'ordonnançabilité, le TEPC de tous les gestionnaires d'interruptions doit être connu, ce que permet d'obtenir l'analyse statique de TEPC. Cependant, celle-ci ne considère que du code isolé, et ne prend donc pas en compte l'impact des interruptions sur la tâche interrompue (vidage du cache d'instructions, rupture du flot d'instructions dans le pipeline). Ce surcoût doit être ajouté au TEPC du gestionnaire d'interruption avant l'analyse d'ordonnançabilité. [10] et [17] présentent des exemples de prise en compte des interruptions dans l'analyse d'ordonnançabilité.

4.8. Résultats quantitatifs

Ce paragraphe présente nos premiers résultats quantitatifs. Nous comparons les résultats obtenus par l'analyse statique des 12 directives de RTEMS étudiées aux résultats de l'exécution réelle.

Pour mesurer les temps d'exécution de chacune des 12 directives, nous avons reproduit son pire scénario d'exécution. Par exemple, concernant les directives qui utilisent l'allocation de mémoire dynamique, comme *rtems_task_create*, nous avons fragmenté la mémoire avant d'appeler la directive concernée. Les mécanismes internes du Pentium (registres de décompte d'événements) nous ont permis de compter les instructions, les erreurs de prédictions de branchement et les défauts dans le cache d'instructions. Comme notre analyseur ne prend en compte ni le cache de données ni le deuxième pipeline entier du processeur de Pentium, ces deux éléments ont été désactivés. Pour les mesures, le tampon de cibles de prédiction de branchement, ainsi que le cache d'instructions ont été vidés avant chaque appel de directives.
Le tableau 1 présente nos résultats. Les colonnes sont étiquetées *"est."* pour les résultats obtenus par analyse statique de TEPC et *"exec."* pour les résultats obtenus en exécutant les directives.

I	II			III			IV		
	Défauts cache instruction			Mauvaises préd. branchement			TEPC (nombre de cycles)		
Directive RTEMS	*est.*	*exec.*	ratio	*est.*	*exec.*	ratio	*est.*	*exec.*	ratio
rtems_clock_get	10	10	1.00	8	7	1.14	1420	1302	1.09
rtems_semaphore_create	106	88	1.20	204	112	1.82	27050	11878	2.28
rtems_semaphore_obtain	380	180	2.11	983	363	2.71	288484	97417	2.96
rtems_semaphore_release	227	137	1.66	271	131	2.07	40414	17025	2.37
rtems_task_create	393	231	1.70	852	283	3.01	435550	309618	1.41
rtems_task_delete	315	213	1.48	630	262	2.40	72679	47275	1.54
rtems_task_ident	23	35	1.21	229	31	7.39	150093	98439	1.52
rtems_task_resume	93	90	1.03	138	83	1.66	19045	11521	1.65
rtems_task_set_priority	164	140	1.17	182	139	1.31	24728	20179	1.23
rtems_task_start	473	214	2.21	730	149	4.90	138940	36532	3.80
rtems_task_suspend	111	100	1.11	149	110	1.35	20106	16237	1.24
rtems_task_wake_after	156	116	1.34	250	122	2.05	19823	16923	1.17
Moyenne			1.44			2.65			1.86

TAB. 1 – *Confrontation entre résultats de l'analyse et résultats expérimentaux*

La colonne II donne le nombre de lectures de blocs d'instructions ayant provoqué un défaut dans le cache ; la colonne III présente le nombre de branchements mal prédits par le mécanisme de prédiction de branchement. La colonne IV confronte les TEPCs mesurés et estimés, exprimés en nombres de cycles sur un processeur Pentium 90Mhz. Ces résultats montrent que l'analyseur HEPTANE fournit des résultats sûrs, et que la modélisation de micro-architecture qu'il inclut réduit le pessimisme de l'analyse.
Les deux sources principales du pessimisme de l'analyse ont été identifiées. La première est la multiplicité des appels à la même fonction dans une même directive, alors que cette fonction est exécutée au plus une fois par directive. Cette fonction est *Thread_Dispatch*, qui effectue le changement de contexte. Son TEPC est important et elle peut apparaître jusqu'à 3 fois dans l'arbre d'appel de certaines directives. La deuxième source de pessimisme provient de quelques fonctions qui peuvent être appelées aussi bien par l'utilisateur que par les directives de RTEMS. Dans le dernier cas, il s'avère que la majorité des paramètres de ces fonctions est constante, et dans ce cas alors qu'une partie du code de ces fonctions n'est pas utilisée. D'où une surestimation du TEPC des fonctions dûe à la prise en compte de chemins d'exécution impossibles lors du calcul du TEPC. Nous étudions actuellement la possibilité d'éliminer ces deux sources de pessimisme en appliquant une des techniques d'évaluation partielle sur le code source de RTEMS. L'évaluation partielle est une transformation du code source qui devrait nous permettre de supprimer ce type de chemin d'exécution impossible.
Cependant, certains chemins d'exécution impossibles restent difficilement détectables statiquement, comme par exemple des branchements conditionnels mutuellement exclusifs.

Sur cet exemple, le pire chemin d'exécution est A;B;C;D. Mais les branches B et D s'excluent mutuellement. La détection de ce type de situation requiert une analyse flot de données.

```
if(i==0)                      | A
    {some code; j=1;}         | B
if(j==0)                      | C
    {some code}               | D
```

5. Conclusions

Nous avons utilisé l'analyse statique de TEPC pour obtenir le TEPC des appels système d'un système d'exploitation temps-réel : le noyau RTEMS.
Nous avons observé que RTEMS est constitué d'un grand nombre de fichiers répartis dans de nombreux répertoires et que le degré de réutilisation des fonctions est élevé. Ceci favorise la construction d'outils d'analyse de WCET permettant l'analyse de WCET *partielle* (*c.-à-d.* l'analyse de fragments de programme plutôt que des programmes entiers).
Nous avons constaté que les restrictions imposées par HEPTANE au langage source sont raisonnables : la plupart des appels dynamiques ont pu être remplacés par des appels statiques, et les "goto" ont été facilement remplacés par des boucles.
Concernant l'identification du pire chemin d'exécution, l'obtention du nombre maximum d'itérations des boucles n'a été évidente que pour 25% d'entre elles. L'obtention du nombre maximum d'itérations des boucles restantes a nécessité une étude approfondie du code source de RTEMS. L'estimation concernant la boucle de la fonction d'allocation dynamique de mémoire est plutôt pessimiste. Il est clair que ce mécanisme d'allocation dynamique n'est pas adapté aux systèmes temps-réel strict. Le nombre maximum d'itérations de la plupart des autres boucles (par exemple les boucles de l'ordonnanceur) dépend des conditions de fonctionnement, telles que la loi d'arrivée des interruptions ou le nombre réel de tâches activées à un niveau de priorité donné. Il est possible de ne pas prendre en compte certains cas très pessimistes quand les informations fournies par l'utilisateur le permettent.
En ce qui concerne notre travail actuel, nous travaillons à l'utilisation de *l'évaluation partielle* avant analyse de TEPC (*c.-à-d.* la spécialisation d'un programme en exploitant la connaissance partielle de ses entrées) pour améliorer la précision des résultats de l'analyse de TEPC. Cette technique devrait permettre d'écarter quelques chemins d'exécution impossibles lors de l'analyse. Nous utilisons actuellement Tempo [5], un évaluateur partiel pour C sur le code de RTEMS. Nous projetons également d'étudier quels algorithmes présents dans les systèmes d'exploitation sont les plus adaptés aux systèmes temps-réel strict (gestion de files d'exécution, allocation dynamique de mémoire).

Remerciements

Le travail décrit dans cet article a été partiellement financé par la Direction Générale de l'Armement (DGA/DSP), contrat #98.34.375. Nous remercions Philippe Selo, qui a initié l'analyse de RTEMS en été 1999. Nous remercions également Éric Valette, de Canon recherche France, Pascal Chevochot et David Decotigny de l'IRISA, ainsi que les relecteurs de l'article pour leurs remarques constructives sur des versions précédentes de l'article.

Bibliographie

1. Anceaume (E.), Cabillic (G.), Chevochot (P.) et Puaut (I.). – A flexible run-time support for distributed dependable hard real-time applications. *In : Proceedings of the 2nd International Symposium on Object-oriented Real-time distributed Computing*, pp. 310–319. – St Malo, France, mai 1999.
2. Audsley (N.C.) et Burke (M.). – Distributed fault-tolerant avionic systems - a real-time perspective. *In : Proceedings of IEEE Aerospace*, pp. 8.104.1 – 8.104.18. – Aspen, U.S.A., 1998.
3. Colin (A.) et Puaut (I.). – Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, vol. 18, n° 2-3, mai 2000, pp. 249–274.
4. Colin (A.) et Puaut (I.). – A modular and retargetable framework for tree-based wcet analysis. *In : Proc of the 13th Euromicro Conference on Real-Time Systems*. – Delft, The Netherlands, juin 2001.
5. Consel (C.), Hornof (L.), Lawall (J.), Marlet (R.), Muller (G.), Noyé (J.), Thibault (S.) et Volanschi (N.). – Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, vol. 30, n° 3, septembre 1998.
6. Engblom (J.). – Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. *In : Proceedings of the 1999 Real-Time technology and Applications Symposium*, pp. 46–55. – Vancouver, CA, juin 1999.

7. Guillemont (M.). – *Chorus/classix r3 technical overview.* – Rapport technique n° CSI-TR-96-119.13, Chorus Systems, mai 1996.
8. Healy (C. A.), Whalley (D. B.) et Harmon (M. G.). – Integrating the timing analysis of pipelining and instruction caching. *In : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pp. 288–297.
9. Inc. (Accelerated Technology). – Nucleus PLUS embedded real-time kernel. – Accelerated technology, *http : //atnucleus.com*, 1999.
10. Jeffay (K.) et Stone (D.L.). – Accounting for interrupt handling costs in dynamic priority task systems. *In : Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS93)*, pp. 212–221. – Raleigh-Durham, North Carolina, décembre 1993.
11. Li (Y.-T. S.), Malik (S.) et Wolfe (A.). – Cache modeling for real-time software: Beyond direct mapped instruction cache. *In : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*. IEEE, pp. 254–263. – IEEE Computer Society Press.
12. Liu (C.L.) et Layland (J.). – Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, vol. 20, n° 1, janvier 1973, pp. 46–61.
13. Mueller (F.). – *Static Cache Simulation and its Application.* – Thèse de PhD, Departement of Computer Sciences, Florida State University, juillet 1994.
14. On-Line Applications Research Corporation, Huntsville, AL, USA. – *RTEMS Applications C User's Guide*, octobre 1998, 4.0 édition. (http://www.oarcorp.com/RTEMS/rtems.html).
15. Puschner (P.) et Burns (A.). – A review of worst-case execution-time analysis. *Real-Time Systems*, vol. 18, n° 2-3, mai 2000, pp. 115–128. – Guest Editorial.
16. Puschner (P.) et Koza (C.). – Calculating the maximum execution time of real-time programs. *Real-Time Systems*, vol. 1, n° 2, septembre 1989, pp. 159–176.
17. Sandström (K.), Eriksson (C.) et Fohler (G.). – Handling interrupts with static scheduling in an automotive vehicle control system. *In : Proceedings of the 1998 Real-Time technology and Applications Symposium.* – Hiroshima, Japan, octobre 1998.
18. Zhang (N.), Burns (A.) et Nicholson (M.). – Pipelined processors and worst case execution times. *Real-Time Systems*, vol. 5, n° 4, octobre 1993, pp. 319–343.