Calcul de temps d'exécution au pire cas pour code mobile

Nadia Bel Hadj Aissa, Gilles Grimaud
IRCICA/LIFL, UMR CNRS 8022
INRIA Futurs, projet POPS
Bâtiment M3, Cité Scientifique, 59650 - Villeneuve d'ascq Cedex
{Nadia.Bel-Hadj-Aissa,Gilles.Grimaud}@lifl.fr

Abstract

*Nous nous proposons de définir une approche distribuée de calcul de temps d'exécution au pire cas en considérant les problèmes liés à la mobilité du code. En effet, l'utilisation de supports d'exécution distincts (i.e*i.e. *producteurs et consommateurs de code), qui ne se font pas mutuellement confiance et qui n'ont pas des ressources équitables, soulève de nouveaux problèmes quand le respect des échéances doit être assuré. La distribution efficace du processus de calcul de WCET devra permettre, en premier lieu, d'alléger la charge attribuée au consommateur qui dispose d'une architecture minimaliste. En deuxième lieu, le processus distribué devra garantir que la sécurité de l'hôte n'est pas menacée. Nous nous addressons plus précisément à des problèmes de sécurité souvent négligés tels que la disponibilité des ressources et la garantie de temps de réponse.*

1 Introduction

Nous nous proposons de définir une approche distribuée de calcul de temps d'exécution au pire cas en considérant les problèmes liés à la mobilité du code. En effet, l'utilisation de supports d'exécution distincts (i.e. *producteurs et consommateurs de code*) qui ne se font pas mutuellement confiance soulève de nouveaux problèmes quand le respect des échéances doit être assuré. Dans nos hypothèses de départ, le producteur, qui compile le code source dans une forme intermédiaire, dispose de ressources matérielles (mémoire, CPU,...) considérées infinies. D'autre part, le consommateur, qui reçoit le code intermédiaire et l'exécute, est un Petit Objet Portable et Sécurisé (POPS) tel qu'une carte à puce,...Deux questions majeures se posent généralement quand on s'adresse à ce type de dispostif dans le cadre d'un scénario de mobilté du code : Est-ce que le peu de ressources dont dispose le consommateur est suffisant pour exécuter le code et vérifier sa sûreté ? Comment garantir que le code envoyé par le producteur peut s'exécuter sur l'hôte sans compromettre sa sécurité ?

Nous essayerons, dans ce qui suit, de présenter une ébauche de notre réponse à ses questions en considérant des problèmes de sécurité essentiels qui restent en suspens dans les travaux actuels, et cela tant en termes de disponibilité des ressources qu'en termes de garantie de temps de réponse dans des systèmes temps réels désormais ouverts.

2 Pourquoi distribuer le calcul de WCET ?

Plusieurs approches de calcul de WCET existent aujourd'hui [1, 2, 3]. Toutefois, les travaux qui considèrent les problèmes liés à la mobilité du code ne sont pas nombreux [4]. En effet, le processus de calcul du pire temps d'exécution sur du code mobile diffère du processus atomique classique qui s'exécute dans une plateforme dont les propriétés (architecture matérielle, environnement d'exécution, stratégie de compilation ou d'interprétation...) sont connues à l'avance.

Quand on se place dans le contexte de code mobile, le code est d'abord compilé sur un producteur puis mis à la disposition de consommateurs potentiels qui devront le déployer sur leur propre environnement d'exécution. Au moment de la compilation, le producteur ne peut émettre aucune supposition préalable sur le consommateur de code. La forme intermédiaire du code source qu'il fournit est complètement indépendante de la plate-forme sur laquelle il va s'exécuter. Ce schéma de fonctionnement distribué a un impact sur l'algorithme de calcul de WCET. Ce dernier, intimement lié à l'architecture matérielle du consommateur (i.e. *nombre de cycles du processeur consommés par le programme*), doit donc être exécuté au moment du déploiement du code sur le consommateur plutôt qu'au moment de la compilation sur le producteur.

Toutefois, les dispositifs visés étant fortement contraints en terme de puissance de calcul, d'espace mémoire ..., on peut difficilement envisager de laisser le calcul du WCET à la charge du consommateur. L'analyse statique de code, la recherche du plus long chemin dans un arbre, par exemple, sont des composantes essentielles dans un tel algorithme et sollicitent généralement beaucoup de ressources. De ce fait, la distribution efficace du processus de calcul de WCET devrait profiter des ressources considérés infinies du producteur de code pour

minimiser la charge de calcul laissée au consommateur dans la mesure du possible.

Une première possiblité consiste à exporter un profil décrivant l'architecture matérielle du consommateur. Le producteur disposant de ces informations serait alors capable de calculer le pire temps d'exécution de son programme sur la cible désormais connue. Exporter des informations décrivant des éléments matériels n'est pas souhaité dans le contexte de la mobilité de code pour les nombreux problèmes de sécurité soulevés (e.g. *attaques temporelles dans le cas des cartes à puce*). En effet, ces informations mises à disposition d'utilisateurs malveillants peuvent servir à déstabiliser le système et entraîner la fuite d'informations confidentielles.

Une deuxième alternative consiste à attribuer au producteur les calculs fastidieux ne nécéssitant aucune connaissance préalable de la plateforme d'exécution (e.g. *la détermination des bornes des boucles*). De l'autre coté, la finalisation du calcul par la détermination du WCET global resterait à la charge du consommateur. Dans cette optique, la distribution du processus de calcul se traduit par un échange d'information entre deux sites d'exécution différents qui ne se font pas mutuellement confiance et pose aussi des problèmes de sécurité (en terme de disponibilité). En l'occurence, un producteur de code malintentionné peut donner intentionnellement une valeur de WCET inférieure à la valeur réelle, induire l'ordonnanceur en erreur et engendrer un déni de service (e.g. *un déni d'accès au processeur*) alors que le système avait établi qu'il était en mesure de respecter les échéances de chaque tâche. Par ailleurs, dans le cas d'un système temps réel dur, un producteur de code peut minimiser le WCET (e.g. *en donnant des bornes de boucles inéxactes*) afin d'empêcher le consommateur de respecter les échéances.

On se propose dans ce qui suit de définir un schéma de calcul de WCET distribué qui définit deux phases se déroulant sur deux supports d'exécution distincts et nous privilégions la seconde alternative expliquée ci-dessus amenant le producteur et le consommateur de code à s'inscrire dans une logique de collaboration. En outre, pour sécuriser les échanges, nous optons pour une approche de type PCC (Proof-Carrying Code)[5].

3 Description de notre approche

Pour prédire le WCET d'un code donné, une des propriétés les plus intéressantes à étudier est la détermination des bornes de boucles. Pour calculer le WCET par analyse statique, la connaissance des bornes des boucles est nécessaire. Sans ces informations, le WCET global ne peut pas être calculé. Dans le contexte dans lequel on se place, on veut aussi garantir que les bornes des boucles sont sûres et qu'elles n'induiraient pas un mauvais fonctionnement du système.

3.1 Description du traitement du coté du producteur de code

3.1.1 Construire un graphe de flot de contrôle pondéré

Par pondérer le graphe de flot de contrôle, on entend déterminer les poids des différents arcs qui le constituent, comme illustré par la figure 1. Un arc orienté allant du bloc de base BB1 à BB2 ayant un poids n signifie que l'arc orienté BB1 vers BB2 sera emprunté au maximum n fois lors de l'exécution du programme.

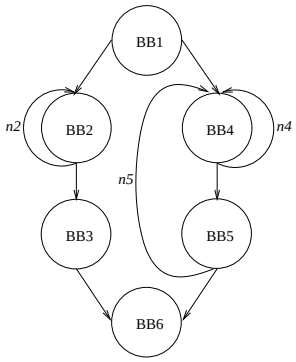

FIG. **1. Exemple de graphe de flot de controle pondere**

Dans une approche similaire à l'inférence de type, on considère qu'une variable x du programme P peut être non initialisée $\top$, un ensemble contenant une valeur constante $\{\alpha\}$, un intervalle ouvert dont la borne inférieure et le pas sont connus $[\alpha \,\overset{\sigma}{.}\, [$, un itérateur potentiel $[\alpha \,\overset{\sigma}{.}\, \psi]^{?}$ ou un itérateur confirmé $[\alpha \,\overset{\sigma}{.}\, \psi]^{!}$. Comme dans un moteur d'inférence classique, nous avons aussi défini des règles de transition qui dépendent des instructions du programme. Par exemple, si une variable x était à l'état $\top$ et que le moteur d'inférence rencontrait une instruction qui initialise x à une constante, le type de x devient désormais $\{\alpha\}$. Aux instructions de branchement, des opérations d'unification permettent de fusionner les informations recueillies par les différents chemins d'exécution du programme.

3.1.2 Construire l'arbre annoté

Une fois les boucles identifiées, les poids reportées sur les arcs orientés du graphe de flôt de contrôle doivent nous servir à déterminer le nombre de fois chaque bloc de base s'exécute. Le poids de l'arc de retour allant de BB2 à BB1 est propagé sur les noeuds BB2 et BB1 ainsi que sur tous les noeuds qui se trouvent sur un chemin menant de BB2 jusqu'à BB1.

Dans le cas de boucles imbriqués comme dans la figure 2, la boucle correspondant à l'arc de retour $BB4 \rightarrow BB5$ englobe la boucle correspondant à l'arc de retour $BB4 \rightarrow BB4$. Dans ce cas, le nombre d'exécution de BB4 est le

résultat de la multiplication des poids des arcs de retour de la boucle englobée et englobante.

Une fois le nombre d'exécution de chaque bloc de base calculé, tous les arcs de retour sont supprimés. On construit un arbre à partir du graphe obtenu en ajoutant un nouvelle branche d'exécution à chaque fois qu'une alternative est rencontrée. La principale intuition derrière cette transformation est que la recherche du plus long chemin sur un arbre est moins coûteuse que sur un graphe.

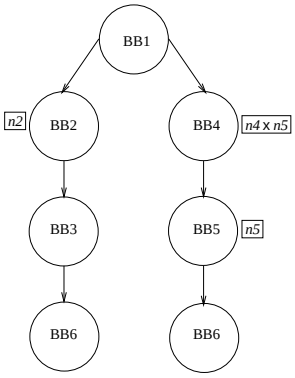

FIG. **2. Transformation du graphe de flot de controle pondere en arbre annote**

3.2 Description du traitement du coté du consommateur de code

3.2.1 Vérification des itérateurs

Le producteur fournit un binaire contenant le code intermédiaire ainsi que l'arbre annoté et les annotations de boucles. A partir de ces éléments, le consommateur devrait être capable de vérifier les variables d'itérations annoncées par l'extérieur. Le consommateur doit s'assurer que la séquence d'instructions permettant de fabriquer un itérateur a bien lieu et conserve les même propriétés. La vérification se fait en une seule passe car les annotations de boucles sont reportées à l'entrée de chaque bloc de base et peuvent être calculés linéairement pour les instructions contenues dans les blocs de base.

3.2.2 Instanciation sur la plateforme cible

Lorsque l'étape de vérification est validée, le code peut être transformé en code natif et les performances du système sont désormais connues (i.e. *nombre de cycles du processeur consommés par instruction*). Les WCET des blocs de base peuvent alors être calculés et un algorithme de recherche du chemin le plus coûteux sur l'arbre annoté permettra de déterminer le WCET global.

4 Conclusions et perspectives

Nous avons présenté les spécificités du code mobile et les conséquences qu'elles peuvent avoir sur l'algorithme de calcul du WCET. Les algorithmes classiques n'étant pas applicables sans adaptation préalable, nous avons montré à travers l'approche que nous avons proposé que ce calcul peut être distribué et finalisé sur le consommateur de code. Le processus de calcul distribué fait intervenir le producteur de code pour construire un graphe de flot de contrôle pondéré en détectant les bornes des boucles automatiquement ou en se basant sur les annotations fournies par le programmeur. Ce graphe est transformé ensuite en un arbre annoté afin d'alléger la charge de recherche du plus long chemin qui sera effectuée sur l'hôte. Sur le consommateur, on procède simplement à une vérification linéaire des bornes annoncées et à l'instanciation du calcul sur la plate-forme cible.

Il faut toutefois souligner que dans le cas général, les bornes des boucles ne peuvent pas être déduites automatiquement à partir d'une analyse du code. Le problème de la détermination des bornes des boucles est en effet une déclinaison du problème de terminaison de programme qui est lui-même NP-complet. Le programmeur est donc sollicité pour introduire des annotations dans le code afin de compléter les informations récoltées automatiquement. Cette tâche effectué manuellement peut introduire des erreurs et devenir difficile à gérer quand il s'agit de programmes complexes. Des programmeurs distraits ou malveillants peuvent, par ailleurs, donner des annotations erronées qui fausseraient la prédiction du WCET. L'insertion d'itérateurs fictifs permettraient dans ce cas de vérifier les bornes des boucles données par annotations au même titre que les itérateurs déterminés automatiquement.

Références

[1] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Techniques et Sciences Informatiques (TSI)*, 22(5) :651–677, 2003.

[2] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *Workshop on Languages, Compilers & Tools for Real-Time Systems*, 1995.

[3] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1989.

[4] Aloysius K. M. and Weijiang Y. Enforcing resource bound safety for mobile snmp agents. In *Proceedings of 18th Annual Computer Security Applications Conference (ACSAC)*, pages 69–77, Las Vegas, Nevada, USA, 2002.

[5] G. C. Necula. Proof-Carrying Code. In *Proc. of the 24th Symp. Principles of Programming Languages*, 1997.