Sujet de stage d'ingénieur en informatique HPC à Université Côte d'Azur : Génération de code source efficace pour le contrôle des systèmes synchrones critiques

2022

Responsable du stage : Pr Sid Touati.

Contacts :

— Sid.Touati@inria.fr

Laboratoires : I3S/INRIA

Équipe d'accueil : COMRED/Kairos

1 Motivations

Les méthodes formelles de modélisation des systèmes critiques proposent des langages précis de description de comportement, dont la sémantique est plus riche que celle des langages de programmation impérative. Grâce à ces méthodes formelles, il est possible de garantir ou vérifier certaines propriétés exigées pour le bon fonctionnement d'une application. Ces propriétés peuvent être de différentes natures : performances (temps d'exécution, consommation mémoire), de correction (une fonction calcule-t-elle réellement ce qu'elle est sensée calculer ?), de comportement (l'application réagit-elle à son environnement au bon rythme ?). Il existe toute une communauté de recherche académique et industrielle autour des méthodes formelles de description d'application critiques (aviation, transport, armement, industrie spatiale), qui

travaille depuis plusieurs années autour des langages et des outils de d'analyse et de manipulation des modèles.

Cependant, ces langages formels ont encore du mal à être acceptés par une majorité de concepteurs ou de programmeurs. Deux explications peuvent être apportées :

1. Mentalité algorithmique des programmeurs : les langages formels proposent de modéliser des applications avec une vision assez éloignée de l'algorithmique, l'application n'est pas décrite avec son comportement dans un langage de programmation classique. Cela exige de raisonner avec une vue d'esprit différente de celle de l'informaticien, dans un niveau d'abstraction plus élevé. Il existe plusieurs sortes de modélisations haut niveau : graphiques (UML), fonctionnelles (équations mathématiques), synchrones (description minutieuse et formelle des horloges qui rythment le système), etc.
2. Absence d'outils fiables de génération de code efficace : avec les langages formels, il est possible d'analyser le système, de le manipuler, mais il n'est pas encore possible de générer du code efficace, comme le feraient des compilateurs. Par code efficace, nous entendons un programme source puis un code binaire qui s'exécute avec des performances satisfaisantes sur des processeurs physiques. Actuellement, il existe des outils de simulation de modèles formels hélas peu performants.

Afin de motiver les concepteurs d'utiliser des méthodes formelles, nous orientons nos efforts de recherche durant les prochaines années à résoudre le 2e point ci-dessus. Nous souhaitons réfléchir et implémenter des méthodes automatiques de génération de code efficace à partir de langage de spécification haut-niveau qui combine flots de données et de contrôle, description causale et temporelle, propriétés attendues et règles de réécriture opérationnelles. Les langages synchrones sont des candidats idéaux pour la description des règles opérationnelles, le langage CCSL (Clock Constraint Specification Language) est un candidat pour la description des propriétés causales et temporelles. Le code généré se fera dans un langage impératif connu comme le C et le C++, qui sera ensuite destiné à une optimisation avancée via un compilateur. Le code généré devra être *compiler friendly*, à savoir être assez clair pour le compilateur et bien disposé à des techniques d'optimisation de code bas niveau. Le code optimisé est prévu pour être exécuté sur un système em-

barqué ou cyber-physique avec contraintes fortes sur les performances (temps d'exécution ou consommation mémoire).

2 État de l'art

Ce sujet de thèse propose de lier les connaissances de deux grands domaines de recherche en informatique : les méthodes formelles de description de systèmes critiques avec l'optimisation de code (compilation optimisante). Les deux domaines ont évolué séparément durant les décennies passées. Nous arrivons à une époque où les briques des différentes communautés se rassemblent pour construire un édifice commun. Nous présentons ci-dessous un bref aperçu de l'état de l'art de ces deux domaines.

2.1 Description formelle avec CCSL

Le langage CCSL (*Clock Constraint Specification Language* [**CCSL2008**]) est un langage de spécification, formel [**CCSL2009**] et déclaratif, basé sur la notion d'horloge logique qui abstrait la causalité et la temporalité des systèmes. Il permet de décrire les propriétés temporelles et causales que doivent satisfaire les systèmes temps-réels critiques. Une spécification CCSL définit un ensemble de comportements acceptables, il s'agit ensuite de vérifier qu'un système réel et opérationnel satisfait ces propriétés. Pour cela, une spécification CCSL est généralement compilée en automate [**SCP2005**] reconnaissant qui combiné au code du systéme propose un diagnostique. Cependant, il faut trouver un compromis entre la taille de l'automate (souvent avec un nombre infini d'états) et l'efficacité de la vérification. La nature déclarative du langage pousse à chercher des techniques de compilation modulaires (un automate par contrainte) ce qui est souvent non compatible avec l'hypothèse synchrone sur laquelle repose CCSL. Il s'agit de regarder si les techniques de compilation et d'optimisation de code de l'état de l'art permettent d'améliorer la compilation d'une spécification CCSL de façon efficace tout en combinant au code fonctionnel et opérationnel qui décrit le système étudié.

2.2 Compilation de code efficace

Historiquement, l'optimisation de code fait partie de la compilation avancée [**ToD2014**]. Un compilateur ne se contente pas uniquement de transformer un code d'un langage haut niveau vers l'assembleur, mais effectue de très nombreuses analyses et optimisations pour que le code généré soit bien adapté au processeur cible. De nos jours, un bon compilateur industriel (comme icc) ou libre (comme gcc ou Clang/LLVM) contient des centaines d'option de compilation. Malheureusement, seuls les experts savent se servir d'une telle quantité d'options. Le programmeur se contente d'utiliser des options génériques comme -O1, -O2, -O2, -Ofast, etc. La raison est que le problème de trouver la "meilleure" combinaison d'option de compilation pour un programme donné est un problème indécidable, non solvable par un algorithme. L'intervention d'un homme inspiré reste nécessaire pour optimiser au mieux un code.

Un code source est *compiler friendly* s'il est écrit d'une façon claire pour le compilateur, lui permettant de débloquer plusieurs méthodes d'optimisation, que nous regroupons ci-dessous.

1. Exploitation du parallélisme d'instruction : le compilateur analyse les dépendances de données entre les instructions d'un programme, et ordonnance les instructions selon deux critères : optimiser l'exploitation des unités fonctionnelles du processeur (maximiser le parallélisme), optimiser l'utilisation des registres (faire en sorte de maintenir les données en registre et non pas en mémoire).
2. Exploitation du parallélisme de données : le compilateur détecte les traitements parallèles sur les tableaux, et si ce traitement est identique sur chaque case de tableau, il génère des instructions vectorielles.
3. Exploitation des différents niveaux de caches matériels : le compilateur peut restructurer entièrement les nids de boucles d'un programme pour exploiter au mieux les différents niveaux de caches de données. Le but est de maintenir, dans les différents caches, les données traitées par un nid de boucle tant qu'elles sont accédées. Concernant les caches d'instruction, le compilateur peut limiter la taille des corps de boucle (pour éviter qu'une taille de boucle dépasse la taille d'un cache d'instructions) ou déplacer les codes des fonctions (pour éviter que deux fonctions soient en conflit sur le cache d'instruction), etc.
4. Exploitation des prédicteurs de branchement : les branchements sont

parfois difficilement analysables par le compilateur ; Il est difficile de savoir statiquement quel serait la cible d'un branchement conditionnel tant que les données en entrée du programme sont inconnues. Les branchements sont donc traités dynamiquement par le processeur, qui essaye d'anticiper leur destination avant de calculer le résultat du test. Cela est possible car la majorité des branchements dans un programme dit régulier ont un comportement régulier. Le processeur peut ainsi savoir qu'un tel branchement ne serait *a priori* jamais pris, ou toujours pris, en examinant son historique. La tache du compilateur serait ici de générer un code qui aide le processeur à prédire le comportement des branchements.

5. Exploitation du parallélisme de *threads* : Le code source peut contenir des directives de parallélisme (comme OpenMP). Le code généré peut exploiter une granularité de parallélisme au niveau des processus légers, afin d'exploiter les multiples coeurs des processeurs avec une problématique de placement de threads.

3 Sujet de stage

Ce stage se déroulera en collaboration avec un doctorant, qui a pour objectif de générer du code efficace (C ou C++) à partir d'une description CCSL (*Clock Constraint Specification Language*) et des règles opérationnelles de progression du système. CCSL permet de décrire non pas une application entièrement, mais uniquement le comportement de ses horloges. Une horloge permet de déclencher ou pas l'exécution d'une fonction ou d'une tache à un instant précis. CCSL n'est pas un langage de programmation qui permet d'exprimer l'algorithmique des fonctions, il exprime uniquement l'instant de leur exécution. Le code généré doit pouvoir efficacement permettre de calculer les valeurs de ces horloges logiques et de les combiner au code du système. Il y aura les travaux suivants à aborder :

1. Faire des expérimentations massives sur l'outil ECOGEN de génération de code ;
2. Faire une synthèse empirique des résultats ;
3. Déduire de nouveaux schémas de génération de code.

4 Références

ToD2014 Sid Touati and Benoît Dupont-de-Dinechin. *Advanced Backend Code Optimization*. ISTE, Wiley, 2014. ISBN-13 : 978-1848215382. Informations additionnelles

CCSL2009 Charles André. *Syntax and Semantics of the Clock Constraint Specification Language (CCSL)*. RR-6925, INRIA (2009). https://hal.inria.fr/inria-00384077/

CCSL2008 Frédéric Mallet, Charles André and Robert de Simone. *CCSL : specifying clock constraints with UML/Marte*. Innovations in Systems and Software Engineering 4(3) : 309-314, Springer, 2008. https://hal.inria.fr/inria-00371371/

SCP2005 Frédéric Mallet and Robert de Simone. *Correctness issues on MARTE/CCSL constraints*. Science of Computer Programming, 106 : 78-92, Elsevier, 2015. https://hal.inria.fr/hal-01257978/