1 Fiche résumée

ACRONYME DU PROJET : Hisseo
TITRE DU PROJET : Analyse statique et dynamique de programmes avec calculs en virgule flottante
Nature du projet : émergence
Responsable du projet :

Nom : CUOQ Pascal
Fonction : Ingénieur-Chercheur
Institution / Laboratoire : CEA LIST, LSL (Lab. de Sûreté du Logiciel)
Adresse postale : CEA Saclay, 91191 Gif sur Yvette CEDEX
Téléphone : 01 69 08 94 09
Fax : 01 69 08 83 95
Adresse email du responsable du projet : Pascal.Cuoq@cea.fr
Nom du directeur du laboratoire : Denis Platter
Adresse email du directeur du laboratoire : Denis.Platter@cea.fr

Partenaires Digiteo du projet

Laboratoire partenaire n°1 : CEA LIST, Laboratoire de Sûreté du Logiciel
Laboratoire partenaire n°2 : INRIA Saclay - Île-de-France, équipe-projet ProVal
Laboratoire partenaire n°3 : INRIA Paris - Rocquencourt, équipe-projet Gallium

Objectifs du projet (résumé) : scientifiques, techniques, économiques, sociétaux

Hisseo est une proposition de projet Digiteo 2008 pour le DIM "Logiciels et systèmes complexes".

Hisseo vise à réunir des chercheurs dont les spécialités sont soit la compilation (Gallium, INRIA Paris - Rocquencourt.), soit l'analyse statique de programmes (ProVal et CEA LIST). Son objectif est de combler un vide existant au niveau de l'analyse de codes utilisant des calculs en virgule flottante. Les applications visées sont de type programmes embarqués, requérant un très haut niveau de confiance, dans les domaines de l'avionique, le nucléaire, l'automobile, etc.

La sémantique précise définie par la norme IEEE754 n'est pas respectée en détail par les compilateurs. Pour cette raison il est difficile d'avoir une confiance absolue dans les résultats d'analyses faites au niveau du code source. Les directions de recherche proposées s'appuient soit sur la définition d'une sémantique formelle de la compilation des calculs flottants, soit sur l'analyse de l'assembleur généré par la compilation, dans lequel les choix pouvant s'écarter de la norme sont explicités.

Étapes clés du projet :

- Définition d'une sémantique formelle de référence pour les programmes flottants de précision fixée,
- Vérification de propriétés à l'exécution pour des programmes flottants,
- Analyse statique du code assembleur de programmes flottants.

Fournitures :

- Compilateur prouvé formellement correct pour programmes flottants,
- Prototype de vérificateur de propriétés pour du code binaire produit par un compilateur fixé "grand public".
- Prototype d'analyseur pour l'assembleur reposant sur la plate-forme Why[7].

2 Description détaillée du projet

2.1 Intitulé

ACRONYME : Hisseo
TITRE DU PROJET : Analyse statique et dynamique de programmes avec calculs en virgule flottante
Nature du projet : émergence

2.2 Organisation du partenariat

Thématique du projet : Maîtrise du logiciel, et de ses frontières avec les technologies matérielles, y compris calcul intensif.
École doctorale de rattachement : École Doctorale d'Informatique de l'Université Paris-Sud
Nom du responsable de l'ED : Christine Paulin
Numéro de téléphone du responsable de l'ED : 01 72 92 59 05
Adresse électronique du responsable de l'ED : paulin@lri.fr

Responsable du projet : Pascal Cuoq (CV en annexe)
Partenaires Digiteo du projet :

Laboratoire partenaire n°1 : CEA LIST, Laboratoire de Sûreté du Logiciel
Laboratoire partenaire n°2 : INRIA Saclay - Île-de-France, équipe-projet ProVal
Laboratoire partenaire n°3 : INRIA Paris - Rocquencourt, équipe-projet Gallium
(CVs en annexe)

2.3 Description du projet

Les logiciels embarqués sont de plus en plus présents dans notre environnement, et remplissent souvent des fonctions critiques : énergie, automobile, ferroviaire, aéronautique. Pour ces logiciels, il est crucial d'offrir des outils permettant de garantir leur bon fonctionnement. L'une des compétences présente au sein de Digiteo est la conception de ces outils, axe par lequel Digiteo s'inscrit dans le groupe de travail "Sécurité et Défense" du pôle SYSTEM@TIC.

Les méthodes formelles peuvent être utilisées pour augmenter la confiance envers un logiciel critique. Les approches utilisées se divisent en approches statiques et dynamiques : les approches dynamiques s'appuient sur l'exécution du code étudié (on trouve par exemple dans ces approches la génération automatique de tests à partir de description formelles). Les approches statiques, elles, ne nécessitent pas d'exécuter le code analysé. Elle attaquent en général le problème au niveau du code source (non compilé), et utilisent des techniques plus ou moins automatiques – logique de Hoare([8]), interprétation abstraite([6]) – pour énoncer et garantir des propriétés exprimées en termes de points d'exécution dans le programme source et de variables du programme source.

Le cadre Frama-C([1]), développé dans le contexte du projet "CAT" (ANR RNTL) par le CEA LIST et l'INRIA (équipe-projet ProVal), est un environnement pour développer et faire coopérer des analyseurs statiques pour des programmes écrits en langage C. La clef de voûte de cet environnement est le langage logique dans lequel sont exprimées les propriétés que les analyseurs échangent avec l'utilisateur et qu'ils échangent entre eux. Ce langage est nommé ACSL (ANSI C Specification Language [3]), et il exprime, de façon classique dans le domaine de l'analyse statique, des propriétés des variables du code source par rapport à des points dans le code source.

Il est un domaine pour lequel l'analyse au niveau du code source n'est pas entièrement satisfaisante : il s'agit des calculs en virgule flottante [14]. Bien que l'adoption de la norme IEEE 754 ait été une avancée remarquable par rapport à la situation antérieure, la situation en 2008 est encore telle que les choix de compilation peuvent faire diverger les résultats obtenus dans une exécution réelle de ceux qui seraient attendus avec une interprétation stricte de la norme [12]. Une analyse statique faite par rapport à

l'interprétation stricte de la norme peut donc être incorrecte par rapport au programme compilé — ou le programme compilé incorrect par rapport à la norme, selon le point de vue. Par exemple,

```
x=a+b;
y=(x-a)-b;
```

peut facilement s'optimiser en `y=0`. Malheureusement, ce programme flottant bien connu [9] calcule un `y` qui est l'erreur lors du calcul de `x` (si `|a| >= |b|`), c'est-à-dire le nombre flottant tel que l'on ait l'égalité *mathématique* `x+y=a+b`.

Il reste que les causes de certaines de ces divergences sont ancrées dans la conception des processeurs utilisés à l'heure actuelle, et qu'il n'est pas raisonnable d'attendre que ces divergences disparaissent. Avec les jeux d'instructions actuels, la vitesse d'exécution d'un programme qui serait généré pour respecter strictement la norme peut être trop pénalisée pour que l'exercice intéresse les concepteurs de compilateurs. Il est aussi possible de concevoir des analyseurs statiques conservatifs, qui envisagent toutes les possibilités résultant des différents choix faits par le compilateur pendant la génération du code. Mais l'analyse est alors rendue plus complexe (on peut voir cette approche comme l'analyse d'un programme non-déterministe explicitant tous les comportements possibles pour le programme original. Sans hypothèses sur le compilateur, la combinatoire des comportements possibles risque de rendre l'approche infructueuse).

Le sujet du traitement des flottants étant particulièrement difficile, et orthogonal du point de vue technique aux autres problèmes difficiles définis dans le projet CAT, il a été omis des objectifs de celui-ci. Toutefois, les logiciels embarqués, pour lesquels les techniques d'analyse statique ont un intérêt particulier, s'appuient souvent sur des calculs en virgule flottante. Le projet Hisseo vise donc à explorer les techniques qui pourraient être utilisées pour analyser de manière sûre les codes en virgule flottante. Ces techniques sont celles sur lesquelles pourraient s'appuyer dans l'avenir les analyseurs statiques développés dans le cadre du projet CAT pour étendre leurs diagnostics à ce type de codes.

Les tâches sont :

1. La définition formelle de sémantiques pour les calculs flottants et leur exploitation dans la vérification formelle d'un compilateur. De telles sémantiques formelles agissent comme un contrat entre d'un côté le compilateur (quelles sont les transformations de calculs flottants qu'il peut se permettre d'effectuer ?) et de l'autre les analyseurs statiques (quelles garanties peuvent-ils attendre de la part du compilateur ?). Cette partie du travail s'appuiera sur le compilateur Compcert, qui est entièrement spécifié et vérifié en Coq [10].

Une première étape consiste à formaliser en Coq une sémantique déterministe «au bit près» des calculs flottants, suivant les normes ANSI/ISO-C et IEEE 754. La sémantique formelle du sous-ensemble de C utilisé dans Compcert s'écarte de ces normes en plusieurs points et devra être revue. Les preuves de correction (préservation de la sémantique) du compilateur seront adaptées en conséquence.

Du point de vue de l'analyse statique, cette approche (compilateur absolument fidèle aux normes) est optimale en termes de précision des résultats et d'efficacité de l'analyse. Cependant, pour certains processeurs cible (Intel IA32), elle peut être difficile à mettre en œuvre et diminuer les performances du code compilé.

La seconde étape consiste donc à définir une ou plusieurs sémantiques *non-déterministes* pour les calculs flottants, laissant plus de flexibilité au compilateur pour choisir des traductions efficaces de ces calculs. Une telle sémantique formaliserait les déviations par rapport aux normes communément admises dans les compilateurs C, comme par exemple de calculer des résultats intermédiaires avec une précision plus élevée que prévue par la norme, ou d'effectuer des doubles arrondis[1] là où la norme n'en permet

[1] Typiquement, sur les processeurs IA32, les instructions historiques de calcul flottant calculent avec 80 bits de précision. Lors du stockage en mémoire d'une variable 80 bits, elle doit être arrondie à 64 bits et la valeur subit donc un second arrondi. Ce double arrondi peut dans des cas limites donner un résultat différent du calcul spécifié par la norme IEEE 754 directement arrondi à 64 bits.

qu'un.

La prise en compte de ce non-déterminisme dans l'analyse statique et dans la preuve de compilateurs soulève des problèmes nouveaux. Les analyseurs statiques doivent payer les prix d'une plus grande complexité et d'une moindre précision. Quant à la preuve de correction du compilateur, il ne s'agit plus de montrer une préservation «au bit près» des résultats des calculs, mais plutôt de montrer un résultat de raffinement : le compilateur choisit toujours des implémentations des calculs flottants qui sont permises par la spécification non-déterministe.

2. La vérification de propriétés écrites au niveau du code source à l'exécution d'un binaire compilé sans instrumentation. Il est nécessaire de pouvoir travailler sur le code binaire non instrumenté – le code qui sera réellement exécuté après vérification – pour rendre cette approche intéressante, car toute instrumentation modifie le processus de compilation (allocation des registres, ordonnancement et optimisations), et le code vérifié n'est alors plus celui qui est exécuté.

Une autre difficulté à prendre en compte est le fait que la logique ACSL a été conçue pour l'analyse statique et que toutes les formules écrites en ACSL ne sont pas nécessairement exécutables. Il pourra rester nécessaire d'utiliser des prouveurs automatiques, comme c'est déjà le cas dans l'utilisation normale d'ACSL, pour la partie "logique" d'une formule à vérifier, l'exécution fournissant seulement des valuations pour les variables flottantes.

Ces deux contraintes pointent vers l'utilisation d'un simulateur de processeur pour l'exécution, mais les facilités de debuggage proposées par les processeurs peuvent aussi être mises à contribution pour faire l'exécution sur un processeur réel.

3. L'analyse statique de code assembleur. Même en l'absence d'une sémantique formelle de la traduction des opérations flottantes pour le compilateur utilisé, une fois un code source donné transformé en assembleur, les choix de compilation qui peuvent dévier de l'interprétation stricte de la norme IEEE 754 ont presque tous été faits. La seule indéterminée restante est que le jeu d'instruction d'une famille de processeurs peut contenir des instructions de calcul flottant dont le résultat varie d'une génération de processeur à l'autre. On suppose alors le(s) modèle(s) précis de processeur(s) cible utilisé(s) à runtime connu(s) et que des spécifications correctes dans tous les cas de figure sont disponibles pour ces instructions.

Cette fois encore, il sera important de faire le lien entre les propriétés du code assembleur et celles du code source original. Indépendamment de ce point, une possibilité pragmatique pour minimiser la quantité de travail non-portable sera la traduction du code assembleur en C, préservant l'ordonnancement et l'allocation des registres. Il ne s'agira pas de décompilation [5] mais d'une traduction plus directe, et le source C obtenu, bien qu'écrit dans le même langage, sera complètement différent du code original. Le lien entre les propriétés de l'un et de l'autre restera entièrement à faire.

Pour les points 2 et 3, la correspondance de propriétés entre le code source et le code compilé a déjà été l'objet de recherches. Par exemple, dans le cadre de propriétés issues l'interprétation abstraite et d'assembleur de processeur généraliste[13], ou dans le cas de la vérification directe de programmes pour des processeurs de type "DSP"[11]. D'autres travaux se placent dans le cadre d'obtention automatique de certificats de nature "Proof-Carrying Code", [2]. Il sera possible de s'appuyer sur ces recherches.

Références

[1] Frama-c : Framework for modular analysis of c.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, LNCS, Seoul, Korea, August 2006. Springer-Verlag.

[3] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl : Ansi c specification language.

[4] Sylvie Boldo and Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[5] C. Cifuentes. Reverse compilation techniques, 1994.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[7] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[9] Donald E. Knuth. *The art of computer programming*, volume 2. Addison-Wesley, 2nd edition, 1981.

[10] Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 42–54. ACM, 2006.

[11] Matthieu Martel. Validation of assembler programs for dsps : a static analyzer. In *PASTE '04 : Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 8–13, New York, NY, USA, 2004. ACM.

[12] David Monniaux. The pitfalls of verifying floating-point computations. *ACM TOPLAS*, 2007. to appear.

[13] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*, pages 1–13. ACM, 2004.

[14] David Stevenson et al. A proposed standard for binary floating point arithmetic. *IEEE Computer*, 14(3) :51–62, 1981.

2.4 Échéancier

La répartition des tâches selon les partenaires est la suivante :

Tâche	CEA LIST	INRIA Saclay - Île-de-France	INRIA Paris - Rocquencourt
T1	✓	✓	✓
T2	✓	✓	
T3	✓	✓	

L'échéancier des tâches est le suivant :

Tâche	Année 1	Année 2	Année 3
T1	✓		
T2		✓	✓
T3	✓	✓	✓

La collaboration donnera lieu à la proposition d'un sujet de thèse encadré par Sylvie Boldo et Claude Marché et à un post-doctorat encadré par Pascal Cuoq (CVs et sujet de la thèse en annexe).

2.5 Critères proposés pour juger du succès du projet

Publications sur les thèmes du projet Hisseo,
Preuve de concept sous la forme des livrables suivants : prototype de vérificateur de propriétés à l'exécution du binaire sans instrumentation, prototype d'analyseur statique pour l'assembleur,
Spécification formelle d'une sémantique de la compilation des calculs flottants,
Collaborations plus avancées entre les partenaires du projet.

2.6 Contrats précédents ou en cours proches du sujet traité

Projet ANR RNTL CAT.

Le projet CAT (CEA LIST, INRIA Saclay - Île-de-France, ...) vise à produire un cadre pour la coopération de techniques d'analyses pour les codes C critiques.

Projet ANR Blanc CerPAN

L'objectif du projet CerPAN (INRIA Saclay - Île-de-France, INRIA Paris - Rocquencourt projet ESTIME, ...) est de développer et mettre en application des méthodes permettant de démontrer formellement la correction de programmes issus du domaine de l'analyse numérique.

Projet ANR ARA SSIA Compcert

Le projet Compcert (INRIA Paris - Rocquencourt projet GALLIUM, ...) vise à produire un compilateur C réaliste (utilisable pour la compilation de code embarqué critique) entièrement formellement vérifié.

2.7 Référents

David Monniaux
Vérimag - Centre Equation
2, avenue de Vignate
F-38610 Gières Cedex
FRANCE
David.Monniaux@imag.fr
tél : 04 56 52 03 68

Marc Pantel
Laboratoire Informatique et Mathématiques Appliquées
Site ENSEEIHT de l'IRIT-UMR CNRS 5505
Bâtiment I, Bureau I302
2, rue Charles Camichel - BP 7122
F-31071 TOULOUSE CEDEX 7
Marc.Pantel@enseeiht.fr
tél : 05 61 58 83 46

3 Tableaux récapitulatifs

3.1 Récapitulatif des dépenses de fonctionnement

Acronyme du projet : Hisseo

Partenaires :

CEA LIST

INRIA Saclay - Île-de-France

INRIA Paris - Rocquencourt projet GALLIUM

Responsable du projet : Pascal Cuoq

Financement demandé au titre de l'appel à projets					Co-financement				
Allocations doctorales (EUR)	Allocations post-doctorales (EUR)	Personnel technique (EUR)	Autres frais de fonctionnement (EUR HT)	TOTAL	Personnel technique (EUR)	Autres frais de fonctionnement (EUR HT)	Autres co-financements (allocations, gestion. . .)	TOTAL	Partenaire fina
97200	82661	0	x	x	0	0	0	0	

3.1.1 Présentation détaillée de personnel technique en CDD

Descriptif du poste :	Total	Financement sollicité au titre de l'appel à projets	Co-financement
Total	0	0	

3.1.2 Présentation détaillée des "autres dépenses de fonctionnement"

Autres dépenses de fonctionnement	Montant total TTC	Montant total HT	Montant du financement sollicité au titre de l'appel à projets	Montant du co-financement	Partenaire financier
Frais de formation des allocataires	1500				
Manifestations scientifiques	0				
Frais de déplacement	5000				
Animation du réseau, communication					
Accueil de chercheurs étrangers (transports et per diem)					
Consommables plates-formes					
Total					

3.2 Récapitulatif des dépenses d'équipement

Descriptif des équipements	Montant total TTC	Montant total HT	Financement sollicité au titre de l'appel à projets *Max 66% du total HT*	Montant du co-financement	Partenaire financier
station de travail + portable	4000	3344	2207	1137	INRIA Paris Rocquencourt
station de travail + portable	4000	3344	2207	1137	CEA LIST
2 stations de travail + portable	6000	5017	3311	1706	INRIA Saclay - Île-de-France

A Pascal Cuoq

Pascal Cuoq est né en 1975 au Le Puy en Velay. Il s'est spécialisé en informatique pendant qu'il était étudiant normalien à l'École Normale Supérieure de Lyon (1995-1998). En 2002 il a obtenu son doctorat de l'Université Pierre et Marie Curie (Paris 6) dans le domaine des langages fonctionnels de flots synchrones. Il a passé une année à KAIST (Daejeon) et l'Université de Séoul, en tant que membre du groupe de recherche ROPAS, qui se focalise sur l'analyse statique du logiciel. Il est depuis employé comme ingénieur de recherche au sein du Laboratoire de Sûreté du Logiciel au CEA LIST.

B Sylvie Boldo

Sylvie Boldo a 29 ans et est chargée de recherche depuis 2005 à l'INRIA Saclay – Île de France dans l'équipe-projet ProVal.

Cursus

1996–1998 Classes préparatoires au Lycée Louis-le-Grand
1998–2002 Élève à l'École Normale Supérieure de Lyon
2002–2005 Allocataire Couplée à l'École Normale Supérieure de Lyon sous la direction de Marc Daumas sur "Preuves formelles en arithmétiques à virgule flottante"
2005 Visite au NIA (National Institute of Aerospace) à Hampton, USA.

5 publications :

- Sylvie Boldo, Marc Daumas, and Ren-Cang Li. Formally Verified Argument Reduction with a Fused-Multiply-Add. *IEEE Transactions on Computers*, 2008. Minor revision.
- Sylvie Boldo and Guillaume Melquiond. Emulation of FMA and correctly-rounded sums : proved algorithms using rounding to odd. *IEEE Transactions on Computers*, April 2008.
- Sylvie Boldo and Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In Peter Kornerup and Jean-Michel Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 187-194, Montpellier, France, June 2007.
- Sylvie Boldo. Pitfalls of a Full Floating-Point Proof : Example on the Formal Proof of the Veltkamp/Dekker Algorithms. In *Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR)*, pages 52-66, Seattle, USA, August 2006.
- Sylvie Boldo and Jean-Michel Muller. Some Functions Computable with a Fused-mac. In Paolo Montuschi and Eric Schwarz, editors, *Proceedings of the 17th Symposium on Computer Arithmetic*, pages 52-58, Cape Cod, USA, 2005.

C Xavier Leroy

Cursus

1987–1991 École Normale Supérieure, Paris
1989–1992 Doctorat d'informatique fondamentale, Université Paris 7
1993–1994 Post-doc, Stanford University
1994–1998 Chargé de recherche, INRIA Rocquencourt
1999–2004 Ingénieur R&D puis consultant pour la start-up Trusted Logic
Depuis 2000 Directeur de recherche, INRIA Rocquencourt

Sélection de publications pertinentes pour le projet Voir `http://gallium.inria.fr/~xleroy` pour une liste complète de publications.

1. Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42-54. ACM Press, 2006.
2. Sandrine Blazy, Zaynah Dargaye, et Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, LNCS 4085, pages 460-475. Springer, 2006.
3. Jean-Baptiste Tristan et Xavier Leroy. Formal verification of translation validators : A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17-27. ACM Press, 2008.
4. Xavier Leroy et Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 2008. À paraître.
5. Xavier Leroy et Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 2007. À paraître.
6. Xavier Leroy. Java bytecode verification : algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4) :235-269, 2003.

Développements logiciels

- Objective Caml, langage de programmation fonctionnelle, `http://caml.inria.fr/`
- CompCert C, compilateur formellement vérifié, `http://compcert.inria.fr/`
- Vérificateur de bytecode Java Card embarqué sur carte à puce.

Responsabilités collectives

- Responsable scientifique de l'équipe-projet INRIA Gallium
- Direction de thèses : En cours : Z. Dargaye, T. Ramananandro, J.-B. Tristan. Soutenues : F. Pessaux, B. Grégoire, T. Hirschowitz.
- Co-rédacteur en chef de la revue *Journal of Functional Programming*. Membre des comités de rédaction des revues *Journal of Automated Reasoning* et *Journal of Formalized Reasoning*.
- Président du comité de sélection des congrès POPL 2004 et ICFP 2001. Participation à 30 comités de sélection de congrès ou workshops.
- Coordinateur du projet ANR ARA SSIA Compcert (2005–2008).

Prix et distinctions Lauréat 2007 du prix Michel Monpetit de l'Académie des Sciences.

D Sujet de thèse

Titre :	Éviter les guet-apens des compilateurs : preuves formelles de programmes avec nombres flottants
Encadrants :	Sylvie Boldo (CR INRIA)
	Claude Marché (DR INRIA, HDR)

Les logiciels embarqués sont de plus en plus présents dans notre environnement, et remplissent souvent des fonctions critiques : énergie, automobile, ferroviaire, aéronautique. Pour ces logiciels, il est crucial d'offrir des outils permettant de garantir leur bon fonctionnement.

Les méthodes formelles peuvent être utilisées pour augmenter la confiance envers un logiciel critique. On s'intéresse ici à une approche statique, qui attaque le problème au niveau du code source (non compilé), et utilise des techniques plus ou moins automatiques (logique de Hoare [8]), interprétation abstraite) pour énoncer et garantir des propriétés exprimées en termes de points d'exécution dans le programme source et de variables du programme source. Cette approche est celle de la plateforme Why [7] développée dans l'équipe-projet ProVal.

Il est un domaine pour lequel l'analyse au niveau du code source n'est pas entièrement satisfaisante : il s'agit des calculs en virgule flottante [14]. Bien que l'adoption de la norme IEEE 754 ait été une avancée remarquable par rapport à la situation antérieure, la situation en 2008 est encore telle que les choix de compilation peuvent faire diverger les résultats obtenus dans une exécution réelle de ceux qui seraient attendus avec une interprétation stricte de la norme [12]. Une analyse statique faite par rapport à l'interprétation stricte de la norme [4] peut donc être incorrecte par rapport au programme compilé — ou le programme compilé incorrect par rapport à la norme, selon le point de vue. Par exemple,

```
x=a+b;
y=(x-a)-b;
```

peut facilement s'optimiser en `y=0`. Malheureusement, ce programme flottant bien connu [9] calcule un `y` qui est l'erreur lors du calcul de `x` (si `|a| >= |b|`), c'est-à-dire le nombre flottant tel que l'on ait l'égalité *mathématique* `x+y=a+b`.

A ce problème d'optimisations abusives s'ajoutent des divergences matérielles selon les processeurs, détectables sur le code assembleur :

- Les processeurs de type PowerPC (présents sur certains modèles de Macintosh et comme processeurs embarqués dans l'automobile et l'avionique) présentent une instruction FMA qui calcule `a*b+c` avec un seul arrondi et ne donne donc pas le même résultat qu'une multiplication suivie d'une addition.
- Les processeurs Intel (x86) présentent des processeurs flottants sur 80 bits (au lieu de 64). Les calculs peuvent donc être effectués avec plus de précision. Malheureusement, il est impossible de prévoir avant compilation quels calculs seront plus précis. L'attribution des registres 80 bits à telle ou telle valeur peut donc changer la valeur finale d'un calcul puisque les variables intermédiaires peuvent être arrondies à 64 bits si le processeur a besoin de ses registres flottants.

Cela crée des divergences de résultats selon le processeur choisi, mais aussi selon le compilateur et son attribution des registres. Les causes de certaines de ces divergences sont ancrées dans la conception des processeurs utilisés à l'heure actuelle, et il n'est pas raisonnable d'attendre que ces divergences disparaissent. Avec les jeux d'instructions actuels, la vitesse d'exécution d'un programme qui serait généré pour respecter strictement la norme peut être trop pénalisée.

Cette thèse consistera à attaquer ce problème par une analyse statique de code assembleur. On aura ainsi toutes les informations sur la précision de chaque calcul et sur les optimisations effectuées de façon à prouver le code effectivement exécuté. Une des difficultés est que les annotations écrites au niveau du code C doivent être traduites au niveau de l'assembleur. Or ces annotations parlent de points de programme et de variables du programme source qui peuvent avoir disparues, être dédoublées… Une fois cette traduction effectuée, il faudra ensuite pouvoir générer les obligations de preuve conséquentes au code assembleur et à ces annotations en se basant sur les travaux pour le C et Java de la plateforme Why [7].

E Sujet de postdoc

Titre : Vérification de propriétés du code source à l'exécution du binaire non intrumenté
Encadrant : Pascal Cuoq (CEA LIST)

Pour vérifier des propriétés de bon fonctionnement relatives à un programme dont on dispose du code source, les méthodes les plus couramment utilisées à l'heure actuelle se divisent en deux catégories fondamentalement différentes. On peut étudier le code source en utilisant des techniques d'analyse statique. Ou bien, si les propriétés à vérifier sont exécutables, on peut les insérer dans le programme, puis compiler et exécuter le code source ainsi instrumenté. Les deux approches ont leurs avantages et inconvénients respectifs, mais il est un problème particulier auquel aucune ne répond de manière parfaitement satisfaisante même en supposant que ses inconvénients particuliers ne soient pas un facteur limitant. Il

s'agit de la vérification de propriétés de programmes comportant des calculs en virgule flottante. En effet, la pluspart des compilateurs ne permettent pas de déterminer avec certitude le résultat exact d'un calcul mettant en oeuvre des nombres flottants à partir uniquement du code source, ce qui gêne l'analyse statique – qui ne s'appuie justement que sur celui-ci. Les mêmes raisons font que l'instrumentation du code source peut modifier le résultat de calculs qui ne devraient pas être affectés par l'instrumentation. En effet, l'instrumentation peut par exemple, sur un processeur x86 sans SSE, forcer le compilateur à stocker temporairement un registre flottant 80 bits dans un emplacement mémoire 64 bits, et sur un processeur PowerPC, activer ou désactiver l'utilisation par le compilateur de l'opération `fmadd`. Cette instruction est utilisée par les compilateurs comme si elle était équivalente à une multiplication suivie d'une addition alors que ce n'est pas tout à fait le cas. Dans ces deux cas, le résultat donné par le binaire instrumenté peut s'en trouver différent de celui du binaire non instrumenté, alors que l'instrumentation semble sans effet au niveau source. Dans ces conditions, on ne peut pas exclure que l'analyse du binaire intrumenté ne révèle pas de bug alors qu'il y en a un dans le binaire non instrumenté finalement utilisé.

L'objectif de ce post-doctorat est de définir et d'implémenter une méthode de vérification à l'exécution de propriétés du code source, sans que celui-ci ait fait l'objet d'instrumentation avant sa compilation. Le code binaire exécuté pendant l'analyse peut ainsi être exactement identique au code final.

Il s'agira de trouver des solutions pouvant être mises en oeuvre à l'heure actuelle, avec un ou plusieurs compilateurs courants, quite à imposer que la compilation soit faite avec peu ou pas d'optimisations. Ce travail pourra mettre en lumière les modifications qu'il faudrait apporter à un compilateur pour qu'il fournisse toutes les informations nécessaires pour permette d'étendre ces solutions à la compilation avec optimisations.

Le problème présenté est à rapprocher de la vérification de propriétés de temps d'exécution, ces propriétés ayant en commun avec le problème présenté de n'avoir de sens qu'au niveau du code binaire et d'être affectées par la compilation. Une voie qui pourra être explorée est l'utilisation des mêmes simulateurs de processeurs que ceux qui sont utilisés dans ce cadre.