Chaîne de compilation formellement certifiée

David Monniaux

4 décembre 2019

Résumé

Les compilateurs sont des logiciels complexes, dont les bugs peuvent introduire des bugs dans le code compilé. Il existe déjà des compilateurs formellement prouvés correct. Le défi consiste à les améliorer (ajout d'optimisations) et à étendre le champ d'une part vers des langages de plus haut niveau, d'autre part vers la synthèse de mécanismes de sécurité ou vers les architectures parallèles.

1 État de l'art

De nos jours, on rédige rarement des logiciels en langage machine ou d'assemblage, et la plus grande partie des logiciels sont rédigés dans des langages de plus au moins haut niveau, qui sont ensuite compilés ou interprétés. Les compilateurs sont eux-mêmes des logiciels complexes, qui peuvent donc contenir des bugs. Certains de ces bugs se traduisent par un arrêt intempestif de la compilation, mais, dans certains cas, la compilation n'échoue pas et du code incorrect est émis. Comme des changements minimes du code source, d'options de compilation etc. peuvent induire des comportements différents du compilateur, ces bugs peuvent très bien ne pas se manifester dans des versions de test ou de déboguage, mais se manifester dans la version de production. Ces bugs de compilation sont également très difficiles à distinguer de bugs du code source induisant des comportements indéfinis au sens du standard du langage et produisant des comportements aberrants dans le code objet. Ils sont donc très difficiles à détecter et à isoler.

Pour ces raisons, les étapes de compilation sont difficiles à gérer dans l'optique de la qualification du code objet pour des applications critiques (conduite d'aéronefs, de véhicules, de processus nucléaires ou chimiques...). On peut utiliser des arguments tels que «ce compilateur a été utilisé pendant N années dans des applications critiques sans que cela n'ait posé de problème», ou encore procéder à des comparaisons et relectures entre code source et code objet. De telles comparaisons ne sont souvent possibles qu'en désactivant les optimisations, ce qui est restrictif voire intolérable pour de nombreuses applications. [1]

Le projet CompCert [2] a apporté une première réponse à ce problème. Il s'agit d'un compilateur pour le langage C vers différentes architectures (x86, PowerPC, ARM, Risc-V), avec la particularité unique que tant le langage source que le langage cible, mais aussi tous les langages intermédiaires, bénéficient d'une sémantique for-

1. Dans des applications critiques, il n'est souvent pas possible de mettre un processeur plus rapide : ces processeurs peuvent être moins résistants aux conditions de température ou de rayonnement rencontrés, dissiper trop de chaleur, ou encore être trop complexes pour que l'on puisse être confiants dans leur bon fonctionnement.

2. `http://compcert.inria.fr/`

melle, [3] et que chaque phase de compilation vient avec une preuve qu'elles préserve la sémantique du programme. Ces preuves sont vérifiées à l'aide de l'assistant Coq. Bien que CompCert soit une *success story* de la vérification formelle, avec transfert industriel, ce n'est qu'un début vers une chaîne de compilation formellement certifiée.

2 Optimisation avancée

CompCert n'optimise que modérément le code qu'il génère, qui a donc en général des performances inférieures à celles du code produit par des compilateurs tels que gcc, llvm ou icc dans leurs modes supérieurs d'optimisation. Pour certaines applications critiques, notamment en avionique, cette optimisation modérée est déjà un plus, car les compilateurs classiques sont utilisés en désactivant leurs optimisations. Toutefois, dans d'autres applications, par exemple dans le ferroviaire, des performances trop inférieures à celles des compilateurs de référence sont rédhibitoires.

Un premier défi est donc de rajouter dans CompCert des optimisations plus ambitieuses que celle déjà implantées, par exemple :

- déplacement de code invariant en dehors des boucles
- décomposition de variables structures
- réordonnancement d'instructions, spéculation logicielle, notamment pour des cibles à exécution dans l'ordre (comme souvent pour les processeurs destinés à l'embarqué critique, notamment si le temps d'exécution doit être très prédictible)
- réorganisation de boucles, tuilage, pipeline logiciel...
- réorganisation de nids de boucles.

Si ces optimisations sont classiques, les mettre en œuvre en ayant une preuve formelle de correction de leur fonctionnement tout en préservant une complexité algorithmique acceptable est une gageure. [4] Il faut une réflexion très fine sur les éventuelles représentations intermédiaires supplémentaires à introduire, les invariants à maintenir, les propriétés à prouver, etc. afin d'arriver à mener à bien les preuves formelles, voire, idéalement, à les rendre robustes à d'éventuels changements dans l'optimisation.

Des travaux ont été menés à l'IRISA [1] notamment sur les représentations intermédiaires SSA, et à VERIMAG [4] sur le réordonnancement d'instructions en phase finale de compilation, notamment pour processeurs VLIW. Il reste encore beaucoup à faire.

3 Langages de haut niveau

Les systèmes de contrôle critiques sont souvent spécifiés dans des langages de haut niveau adaptés, par exemple Scade. Ces langages sont ensuite compilés vers un langage de plus bas niveau, souvent C. Les compilateurs habituels pour ces langages ne sont pas formellement certifiés et posent donc des problèmes similaires à ceux des compilateurs C : que se passe-t-il en cas de bug, comment justifier de leur absence. Des travaux ont été menés au LIENS [2] sur le compilateur Vélus depuis un sous-ensemble de Scade vers une des représentations intermédiaires de CompCert, mais il ne s'agit que d'une première étape.

3. En réalité, le premier langage formellement spécifié n'est pas exactement C et le dernier pas exactement l'assembleur, de sorte qu'il existe de petites phases dont la correction n'est pas prouvée. Ceci est sans commune mesure avec l'absence globale de preuve dans les compilateurs classique.

4. Ainsi, certains travaux existants sur le réordonnancement certifié avaient une complexité exponentielle [5].

L'exécution de modèles issus de l'apprentissage profond passe souvent par l'usage de descriptions de haut niveau (TensorFlow...) ensuite implantées pour exécution efficace en machine. Là encore, le processus de compilation, mettant éventuellement en œuvre des réorganisations complexes de boucles (e.g. décomposition par blocs), n'est pas certifié. Dans certains cas, un code produit incorrect peut non seulement mettre en œuvre incorrectement le modèle issu de l'apprentissage, mais aussi corrompre les données d'éventuels logiciels plus classiques dans le même espace mémoire. Là encore, il est souhaitable que les étapes de compilation de haut niveau soient formellement certifiées.

4 Intégration de mécanismes de sûreté et de sécurité à l'exécution

Un processeur peut parfois calculer incorrectement en environnement difficile (température, rayonnements...) ou en cas d'attaques matérielles (dérangement intentionnel de l'alimentation électrique, impulsions électromagnétiques ou lumineuses...). Il importe donc de s'en rendre compte voire de compenser ces dysfonctionnements. Si des mécanismes matériels (mémoire à code de correction d'erreurs) peuvent aider, ils ne suffisent pas dans bon nombre de cas (les unités d'exécution ainsi que certains niveaux de caches ne sont typiquement pas dotés de tels codes). L'ajout naïf de calculs ou de tests redondants dans le code source peut être inefficace (un compilateur peut détecter que l'on calcule deux fois la même chose et éliminer la redondance !).

Il serait souhaitable que ces ajouts soient automatiques, au moment de la compilation. Des travaux sont en cours à Verimag sur ce sujet.

Dans le cadre d'une chaîne de compilation certifiée, il est bien entendu nécessaire de démontrer que ces ajouts préservent le fonctionnement normal du programme. Toutefois, on peut être plus ambitieux et vouloir démontrer qu'ils préservent le programme de certaines attaques. Ceci suppose d'avoir un *modèle d'attaquant* ainsi qu'une sémantique non standard de l'exécution intégrant les actions de l'attaquant.

5 Concurrence

Actuellement, CompCert n'a pas de notion d'accès mémoire concurrents. [5] Il serait donc impossible de démontrer, dans sa sémantique, la correction d'un programme faisant des accès concurrents à une même variable depuis plusieurs *threads*. A fortiori, il serait impossible de démontrer la correction d'une phase de compilation transformant une boucle séquentielle dont tous les calculs seraient indépendants en une boucle parallèle, comme peut le faire OpenMP. Pourtant, ce type de transformation est très utile pour la parallélisation automatique, que ce soit pour le calcul scientifique ou pour l'exécution de modèles issus de l'apprentissage profond.

Une difficulté est que l'action des accès sur une mémoire partagée réelle n'est pas un entrelacement des accès des différents cœurs de calcul, comme c'est le cas dans les modèles simples de la concurrence. Les architectures réelles mettent en œuvre des modèles de cohérence faible, qui plus est différents suivants les architectures (x86 vs ARM...) et plutôt mal documentés. Des travaux ont déjà été menés pour l'intégration d'une telle sémantique faible de la mémoire dans CompCert [3], mais n'ont pas été poursuivis.

5. Si ce n'est via des variables «volatiles», que la sémantique considère en quelque sorte comme des ports d'entrée/sortie vers l'environnement. Il peut également exister des *builtins* de lectures ou écritures atomiques en mémoire.

6 Calcul distribué

Le développement d'applications parallèles en mémoire non partagée passe communément par l'usage de bibliothèques d'envoi de messages (par exemple MPI). Cela est malaisé et il est facile d'introduire des *bugs*, parfois correspondant à des entrelacements rares d'exécutions, donc difficiles à reproduire.

Si l'on compile depuis un langage de haut niveau, par exemple un langage synchrone, il peut être possible de délimiter des tâches devant s'exécuter sur des processeurs différents et s'échangeant les données, et synthétiser le code de communication. Un tel code de communication correct par construction serait plus fiable qu'un code émis à la main.

7 Défi

Parmi les items précédents, certains sont déjà implantés dans des compilateurs *mainstream*. La difficulté est de le faire d'une façon certifiée.

Les preuves papier des transformation de compilation sont souvent sur des langages jouet, avec des étapes omises ; parfois ces transformations et/ou ces preuves sont incorrectes. [6] Elles ne sont quasiment jamais formulées au niveau de détail et de rigueur nécessaires à une validation formelle, qui nécessite des sémantiques très précises, des relations de simulation, etc. Il y a là un grand travail de recherche de formalisation et de clarification.

La synthèse vérifiée de code distribué à partir d'une description de haut niveau est une perspective de plus long terme, très prometteuse, permettant d'exploiter les processeurs *many-core*.

Références

[1] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an ssa-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1) :4 :1–4 :35, 2014.

[2] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 586–601. ACM, 2017.

[3] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso : A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3) :22 :1–22 :50, 2013.

[4] Cyril Six, Sylvain Boulmé, and David Monniaux. Certified compiler backends for VLIW processors highly modular postpass-scheduling in the compcert certified compiler, July 2019. working paper or preprint.

[5] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators : a case study on instruction scheduling optimizations. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 17–27. ACM, 2008.

6. Par exemple, l'algorithme original de la sortie de la représentation SSA était incorrect.