Titre de Thèse: Réécriture Spéculative de Programmes Récursifs en Boucles Candidates à une Parallélisation et une Optimisation Efficaces Utilisant un Mécanisme Inspecteur-Exécuteur

Doctorante: Salwa Kobeissi **Directeur de Thèse:** Philippe Clauss

Résumé

Introduction et Contexte :

Du développement du programme à l'exécution finale sur une plate-forme matérielle, un logiciel passe par plusieurs phases de transformation, ce qui rend les codes exécutables finaux bien différents du code source initial. L'objectif général de ces transformations est de générer un code exécutable sémantiquement équivalent au code source d'entrée, mais dont le comportement à l'exécution est satisfaisant en ce qui concerne le temps d'exécution et la taille du code. Les compilateurs appliquent de nombreuses passes d'optimisation sur le code source d'entrée qui modifient souvent, voire suppriment, des instructions, des structures de contrôle ou des structures de données. De telles transformations de code peuvent s'appliquer du niveau instruction jusqu'à la structure globale du code. Certains compilateurs et optimiseurs avancés peuvent paralléliser automatiquement des codes séquentiels en détectant des régions indépendantes.

Dans le domaine de l'optimisation des codes impératifs, les boucles sont des cibles importantes d'optimisation parce qu'elles implémentent généralement les calculs intensifs coûteux en temps d'exécution. Un exemple d'optimiseur-paralléliseur automatique de boucles de type "for" est le compilateur source-à-source Pluto, qui est basé sur le modèle polyédrique. Le modèle polyédrique est un cadriciel mathématique qui apporte une représentation substantielle pour les nids de boucles affines accédant à des tableaux multidimensionnels via des références affines. Ce cadriciel fournit des méthodes d'analyse précises et des transformations d'optimisation et de parallélisation agressives et automatiques de boucles (Pavage, échange de boucles, torsion, ...).

Pourtant, d'autres structures de contrôle qui implémentent aussi des calculs intensifs ne profitent pas des optimisations polyédriques, comme les boucles de type "while", et les fonctions récursives. Il s'avère parfois que les boucles, qui n'ont pas de structure affine à la compilation et ne correspondent pas au modèle polyédrique, peuvent en fait exiber un comportement polyédrique à l'exécution. Par conséquent, il y a encore des opportunités d'optimisation avancées cachées à la compilation, qui peuvent être découvertes à l'exécution.

La technique de spéculation "au niveau thread" consiste à exécuter en parallèle de manière spéculative et optimiste des régions du code avant de connaître les dépendances entre ces régions. Les accès à la mémoire sont surveillés et une vérification d'accès conflictuels est effectuée. Dans le cas où la spéculation es invalidée, à cause d'une violation de dépendance, un mécanisme de récupération est exécuté. Cette technique a été exclusivement dédiée à l'optimisation des structures de boucle. Cependant, son succès et le gain de performance ne sont pas garantis, soit en raison d'un désequilibre de charge entre les threads, ou de spéculations invalides, ou d'optimisations de boucle trop limitées par rapport, notamment, à celles des optimiseurs polyédriques.

Le mécanisme d'inspecteur-exécuteur est une technique bien connue dans le domaine de l'optimisation dynamique et spéculative de boucles. Cette technique implique deux processus principaux : l'inspecteur, qui exécute une version allégée de la boucle, permettant de vérifier au préalable les accès mémoire, et d'effectuer des calculs légers pour guider les transformations de boucles (dépendances de données); et l'exécuteur, qui exécute une version optimisée et parallélisée de la boucle, en utilisant les informations collectées par l'inspecteur.

À cet égard, Apollo, un optimiseur polyédrique spéculatif développé dans l'équipe ICPS du laboratoire ICube, combine les deux approches présentées ci-dessus. Il a été mis en œuvre pour capturer les comportements polyédriques dynamiques de boucles statiquement non affines, à l'aide d'un profilage dynamique, et pour exploiter les puissants outils polyédriques au cours de l'exécution.

Mais qu'en est-il des structures autres que les boucles, comme les fonctions récursives ? Ces fonctions participent à un cycle d'appels pouvant représenter un temps d'exécution important, mais ne peuvent pas bénéficier de techniques d'optimisation et de parallélisation automatiques tout aussi puissantes que pour les boucles.

Etat de l'Art :

Dans les programmes en général, les fonctions récursives, à l'instar des boucles, sont des structures coûteuses en temps, souvent responsables de la majeure partie du temps d'exécution. En général, les fonctions récursives implémentent des algorithmes complexes, pour le calcul haute performance, ou pour l'analyse et le traitement d'énormes structures de données. Une approche récursive est généralement adoptée lors de la résolution de tout problème dont la solution repose sur des instances plus petites et plus simples, via une approche de type "diviser pour régner". La récursivité facilite également l'expression générique de calculs avec des paramètres tels que des profondeurs de recherche ou des tailles des structures de données d'entrée.
Dans la littérature traitant de l'optimisation des fonctions récursives, celles-ci sont soit considérées directement, soit transformées tout d'abord en boucles.

Problématique et Motivation :

Bien qu'il existe de nombreuses études sur l'optimisation des structures récursives et de nombreuses propositions d'optimiseurs automatiques, ils sont tous utilisés au moment de la compilation, sur la base d'une analyse statique. Mais leurs optimisations sont relativement limitées par rapport aux possibilités d'optimisation qui existent pour les structures de boucle. Ces techniques n'incluent pas l'opportunité des codes récursifs pouvant être réécrits sous forme de boucles affines, afin d'être candidats à de puissantes transformations polyédriques d'optimisation et de parallélisation.

C'est pourquoi, l'objectif de cette thèse est d'aller au-delà des limites classiques des techniques de dérécursivation et de découvrir les opportunités d'optimisation que le modèle polyédrique peut offrir aux fonctions récursives dont le comportement à l'exécution est itératif et affine. Notre travail est dédié aux fonctions récursives ayant un comportement particulier, plus précisément un comportement "polyédrique". Non seulement nous nous appuyons sur une analyse statique, mais également sur une analyse dynamique pour guider notre transformation de récursivité en boucles affines. Ainsi, nous présentons la contribution principale de notre thèse qui est le système d'optimisation de programmes récursifs *Rec2Poly*.

REC2POLY :

Le système Rec2Poly est un optimiseur dynamique et spéculatif des fonctions récursives basé sur le compilateur LLVM/Clang. Il découvre, grâce à une technique de profilage hors ligne, les comportements d'exécution affines de codes récursifs qui manipulent des structures de données volumineuses. En cas de succès, il construit un code sémantiquement équivalent où les flots d'exécution liés aux fonctions récursives sont remplacés par des boucles affines qui permettent d'appliquer des optimisations/parallélisations polyédriques de boucles. Rec2poly génère également un mécanisme de vérification à l'exécution selon un schéma inspecteur-exécuteur pour garantir la validité du code itératif généré pour les différentes données d'entrée. Rec2Poly est organisé en trois phases principales: (1) Phase d'analyse statique et de préparation du code ; (2) Phase de profilage hors ligne ; et (3) Phase de génération du code Inspecteur-Exécuteur. Les analyses et les transformations dans ces

phases sont principalement mises en œuvre au fur et à mesure via des passes LLVM sur la représentation intermédiaire du code.

Au départ, Rec2Poly effectue une analyse statique. Il analyse en profondeur le code récursif cible afin d'identifier les fonctions récursives. Il identifie également les fonctions qui peuvent être appelées par ces fonctions récursives, ou qui peuvent invoquer elles-même, directement ou indirectement, des fonctions récursives. Nous appelons ces fonctions identifiées, des *fonctions impactantes*. Ensuite, Rec2Poly construit la tranche arrière statique (*Backward Static Slice* (BSS)) de chaque instruction de stockage en mémoire (écriture) dans les fonctions impactantes, et collecte les identifiants de tous les blocs de base qui contiennent au moins une instruction impliquée dans le calcul de l'adresse de la mémoire cible ou de la valeur stockée, c'est-à-dire dans le BSS. Nous appelons ces blocs de base, des *blocs de base impactants*. Ensuite, Rec2Poly prépare le code pour les phases suivantes et effectue la globalisation des variables locales. Ceci est réalisé en insérant, au début d'une fonction impactante en représentation intermédiaire, un compteur d'invocation, et en transformant chaque structure de données ou variable locale en une structure de données globale indexée à l'aide de ce compteur. Cela permet de garder une trace des variables initialement locales dans les fonctions qui nous intéressent, et de résoudre les dépendances entre leurs différentes invocations.

Ensuite, ce code globalisé étendu est ré-analysé pour obtenir les informations nécessaires liées à la récursivité, en tenant compte des modifications appliquées au code récursif d'origine. En utilisant les informations d'analyse ainsi collectées et la version étendue du code récursif cible, Rec2Poly effectue la deuxième phase, la phase de profilage lors de l'exécution, en générant une version instrumentée du code cible. Le code instrumenté est obtenu en ajoutant à chaque fonction impactante, les instructions pour générer la trace de sortie. La trace générée décrit le flot de contrôle et le comportement mémoire lors de l'exécution du programme; elle est composée des identifiants des blocs de base impactants, des valeurs des compteurs d'invocation et des adresses mémoire référencées dans les blocs de base impactants, via des instructions de chargement (lecture en mémoire) et de stockage (écriture en mémoire).

Une fois le code instrumenté exécuté, la trace générée est donnée comme entrée à une version étendue de l'outil logiciel de reconnaissance de boucle imbriquée NLR (Nested Loop Recognition).

L'algorithme NLR prend en entrée une trace d'exécution d'un programme composée de valeurs entières et de constantes, et construit une séquence de nids de boucles qui produisent la même trace d'origine lors de l'exécution. Dans Rec2Poly, on utilise une version avancée de NLR qui génère, si possible, une représentation de l'ensemble de la trace faite de boucles, voire de boucles affines calculant des expressions affines. Les informations d'analyse de code et le résultat de profilage obtenus, c'est-à-dire le modèle de boucle ainsi généré, peuvent ensuite être utilisés par Rec2Poly pour accomplir la phase de génération et d'optimisation de code, et construire une version optimisée compte tenu du code récursif étendu. Pour cela, Rec2Poly génère un code itératif composé de séquences de nids de boucles optimisables remplaçant les fonctions impactantes du code récursif d'origine.

Étant donné que les boucles de remplacement peuvent être des boucles entièrement affines, elles peuvent alors bénéficier de transformations d'optimisation et de parallélisation polyédriques. Pour cela, Rec2Poly utilise le compilateur polyédrique Pluto. Sinon, une analyse de dépendance dédiée peut encore être utilisée et les boucles générées peuvent être parallélisées en utilisant OpenMP. Le modèle de boucle affine, obtenu à partir du profilage hors ligne basé sur une exécution du code récursif cible, peut également être utile pour les optimisations et exécutions de code ultérieures, même pour différentes entrées de mêmes tailles. Cependant, étant donné que les autres exécutions sont spéculatives, leur validité doit toujours être garantie au moment de l'exécution.

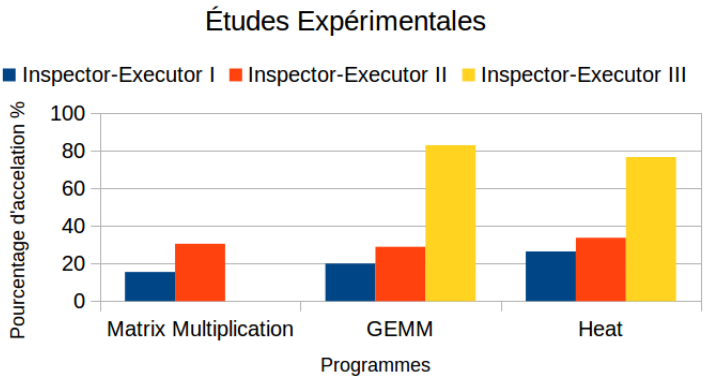
Le programme optimisé généré est vérifiable au moment de l'exécution. C'est grâce au code d'inspecteur parallèle rapide généré par Rec2Poly. Son rôle est de collecter les informations manquantes et de vérifier, au moment de l'exécution, que les boucles affines générées s'exécutant en

parallèle se comportent toujours en conformité avec le code récursif d'origine. Enfin, le code final est généré sur la base du schéma inspecteur-exécuteur composé : du code inspecteur, du code exécuteur contenant les boucles optimisées et du code récursif d'origine
Ce code peut être en outre optimisé à l'aide des optimisations du compilateur LLVM / Clang disponibles.

Lors de l'exécution du programme généré, le code récursif d'origine est exécuté simultanément avec l'inspecteur en utilisant les threads POSIX. En effet, au cas où une erreur de spécification serait détectée au moment de l'exécution, le code entier de l'inspecteur-exécuteur doit être annulé et un état correct de l'exécution du programme doit être récupéré avec la plus faible surcharge en temps possible. Le thread exécutant le code récursif poursuit son exécution tant que l'inspecteur est dans son processus de vérification; lorsqu'une exécution correcte est assurée, le thread est abandonné et l'exécuteur lance ses boucles optimisées parallèles.

Expériences :

Les programmes ont été compilés avec le compilateur Clang version 6.0, et les options -O3 -march=native. Ils ont été exécutés sur deux processeurs Intel Xeon E5-2650 v3 à 2,30 GHz de 10 cœurs chacun. Les programmes parallèles ont été exécutés en utilisant 20 threads sur les 20 cœurs de la plate-forme matérielle.



Nous avons conduit des expériences sur trois programmes pour tester Rec2Poly. Notre première expérience est pédagogique, menée sur une implémentation récursive en C de la multiplication de matrices. Notre deuxième expérience a été menée sur une autre implémentation en C de la multiplication matricielle récursive (GEMM) manipulant des sous-matrices par dichotomie successive jusqu'à un seuil donné. Notre troisième expérience a été menée sur le programme Heat qui est une implémentation récursive en C d'un calcul de stencil. Notons que dans Rec2Poly, nous avons la possibilité de générer des inspecteurs optimisés. Toutes les optimisations ne peuvent pas être activées pour chaque programme. Par exemple, une optimisation peut permettre à l'inspecteur d'ignorer et d'éviter de vérifier les accès mémoire redondants. Une autre peut permettre à l'utilisateur de sélectionner des blocs de base qu'il estime lourds, pour que Rec2Poly donne la priorité dans le processus de vérification et lors de la répartition de la charge aux threads parallèles associés dans l'inspecteur. L'optimisation de l'inspecteur la plus puissante est celle qui stoppe automatiquement les boucles coûteuses existantes dans le code après quelques itérations, lorsque cela est possible. Dans l'histogramme présenté, nous montrons les gains de performance en générant différents inspecteurs-exécuteurs pour chacun de nos programmes expérimentés. Chaque programme a été testé avec un niveau différent d'optimisation de l'inspecteur, en fonction des besoins.

Conclusion et Perspectives :

A notre connaissance, Rec2Poly est la première solution d'optimisation de programme spéculative impliquant la réécriture du code cible. Notre approche a été inspirée par l'approche implémentée dans Apollo. Mais au lieu de cibler les boucles non affines, Rec2Poly cible les fonctions récursives. Le système Rec2Poly est original car : (1) il recherche un comportement d'exécution conforme au modèle polyédrique dans les fonctions récursives, et (2) il utilise un schéma inspecteur-exécuteur pour vérifier non seulement les motifs d'accès à la mémoire comme d'habitude avec un tel schéma, mais aussi le flot de contrôle par rapport à celui des boucles affines. Nous avons montré qu'en utilisant une telle

approche, certains programmes récursifs peuvent profiter de l'optimisation efficace des boucles affines, et même profiter des transformations avancées du modèle polyédrique.

Cependant, alors que le mécanisme inspecteur-exécuteur est adapté à de telles optimisations spéculatives, la performance finale est le plus souvent liée à la performance de l'inspecteur. Dans l'avenir, il serait opportun de compléter l'implémentation de Rec2Poly, en étudiant encore d'autres stratégies pour réduire encore plus la surcharge en temps de l'inspecteur. De plus, il serait intéressant de mettre en œuvre une technique de profilage en ligne comme c'est le cas dans Apollo. Il serait aussi intéressant de considérer les fonctions récursives qui présentent un comportement non linéaire. Par exemple, Rec2Poly peut être étendu pour capturer et gérer les fonctions récursives se comportant comme des boucles mais avec des bornes supérieures variables; bien que de telles boucles ne soient pas affines, elles peuvent être intelligemment optimisées. En outre, des stratégies de ré-ordonnancement des instructions, afin d'obtenir un comportement linéaire, peuvent également être étudiées.

Publications :

Salwa Kobeissi, Alain Ketterlin, Philippe Clauss. *Rec2Poly: Converting Recursions to Polyhedral Optimized Loops Using an Inspector-Executor Strategy*. SAMOS 2020: Embedded Computer Systems: Architectures, Modeling, and Simulation, pp.96-109, 2020.

Salwa Kobeissi, Philippe Clauss. *The Polyhedral Model Beyond Loops Recursion Optimization and Parallelization Through Polyhedral Modeling*. IMPACT 2019 - 9th International Workshop on Polyhedral Compilation Techniques, In conjunction with HiPEAC 2019, Jan 2019, Valencia, Spain.

Salwa Kobeissi, Adnan Utayim, Mohamad Jaber, Yliès Falcone. *Facilitating the Implementation of Distributed Systems with Heterogeneous Interactions*. IFM 2018 - 14th International Conference on integrated Formal Methods, Sep 2018, Maynooth, Ireland. pp.1-19.