Développement d'outils de visualisation de données pour l'exploration et la compréhension de l'architecture des compilateurs, appliqué au compilateur LLVM

Rapport de stage de L3 Informatique

Mai 2021 - Juillet 2021

Melvyn Bertolone Lopez Serrano, sous la supervision de Gabriel Radanne,
co-supervisé par Laure Gonnord

Remerciements

Je tiens avant tout à exprimer ma reconnaissance à Gabriel Radanne pour m'avoir proposé d'effectuer ce stage au sein de son laboratoire de recherche.
Je le remercie pour son implication, son soutien et la confiance qu'il m'a accordée pour ces travaux.

Je souhaite remercier Laure Gonnord pour le temps qu'elle a accordé au suivi du bon déroulement de mon stage.

J'adresse mes remerciements au personnel, stagiaires et doctorants du Laboratoire de l'Informatique du Parallélisme pour leur bienveillance et le temps qu'ils m'ont consacré.

Merci aux enseignants de L'Université Claude Bernard Lyon 1 pour la qualité de leurs enseignements, la transmission de leurs connaissances et de leur passion.

Je remercie ma famille et mes amis pour leurs soutiens.

Enfin, merci à Axelle qui m'a soutenu bien au-delà de ce stage.

Liste des abréviations

CASH : Compilation and Analysis, Software and Hardware
LIP : Laboratoire de l'Informatique du Parallélisme
CNRS : Centre National de la Recherche Scientifique
ENS Lyon : l'École Normale Supérieure de Lyon
INRIA : Institut national de recherche en sciences et technologies du numérique
UCBL : Université Claude Bernard Lyon 1
MILYON : Laboratoire d'Excellence Mathématiques et Informatique à Lyon
SLAS : Synchronous Languages meet Asynchronous Semantics
AST : Abstract Syntax Trees
UQAM : Université de Québec à Montréal
CQL : Cypher Query Language
diff : différentiel sémantique
LLVM : Low Level Virtual Machine
IR : Intermediate representation
PM : PassManager

Table des matières

1 Introduction

1.1 Contexte du stage

Ce stage [1] a été proposé et encadré par Gabriel Radanne et Laure Gonnord dans l'équipe *Compilation and Analysis, Software and Hardware* (CASH) dirigé par Matthieu Moy du LIP. Le LIP est associé au CNRS, ENS Lyon, INRIA, l'UCBL et fait partie du MILYON.

1.2 L'équipe CASH

CASH [2] est une équipe de recherche commune à l'Inria Grenoble Rhône-Alpes et au *Laboratoire de l'Informatique du Parallélisme* (LIP) localisée à l'*École Normale Supérieure de Lyon* (ENS Lyon).

L'équipe CASH travaille sur des thématiques autour de la représentation des programmes parallèles, l'expressivité et évolutivité des analyses statiques, la compilation et l'ordonnancement de l'exécution des programmes flot de données, ainsi que sur la génération et la simulation de matériel. Elle est composée de 7 membres permanents travaillant sur les différents projets énoncés ci-dessous :

— **POLYTRACE** [3]
— **SLAS** [4]
— **CAPESA** [5]
— **CODAS** [6]

1.3 Le projet CAPESA

Le projet CAPESA s'inscrit dans un partenariat franco-canadien entre Inria (France) et le département d'informatique de l'UQAM [7] (Canada), il est porté par Laure Gonnord (équipe CASH) et Sébastien Mosser (UQAM).
Le projet CAPESA propose l'étude des transformations de code en termes de "*différentiel sémantique*". Cette notion sera définie grâce à un code de représentation intermédiaire tel qu'un arbre de syntaxe abstrait (AST) ou un graphe de flux de contrôle. L'objectif n'est pas seulement de calculer ces "différentiels sémantiques" mais de les manipuler dans différents contextes : soit, être capable d'appliquer ces différences sémantiques sur des programmes différents des originaux, quantifier l'interférence entre deux différences sémantiques, ou plus généralement étudier les relations entre ces différences. Cette approche sera validée expérimentalement en validant des problèmes venant des domaines d'expertise des équipes du projet, sur l'analyse statique des passes des compilateurs (expertise de CASH), et sur les *commits* git (expertise de l'UQAM).
Le travail de ce stage participe au projet en donnant des outils de visualisation et de compréhension des passes du compilateur.

1. Stage d'initiation en Informatique, Unité d'enseignement dispensée à l'Université Lyon 1, dans le cadre de la troisième année de licence d'informatique de Lyon 1
2. Site web équipe CASH : http://www.ens-lyon.fr/LIP/CASH/
3. Site web du projet POLYTRACE : http://www.ens-lyon.fr/LIP/CASH/?p=438
4. Site web du projet SLAS : http://www.ens-lyon.fr/LIP/CASH/?p=399
5. Site web du projet CAPESA : http://www.ens-lyon.fr/LIP/CASH/?p=390
6. Site web du projet CODAS : http://www.ens-lyon.fr/LIP/CASH/?p=93
7. Site web UQAM : https://uqam.ca/

2 Les technologies étudiées

Résumé LLVM est un compilateur pour les programmables C, C++, Ada, Fortran …. L'infrastructure du compilateur transforme les programmes vers une représentation intermédiaire, sur laquelle il va travailler en appliquant des *passes* ("phases") d'analyse et d'optimisation pour améliorer les performances. On s'intéresse ici la visualisation sous forme de graphe du comportement du compilateur LLVM ainsi que de son architecture. Pour ce stage, j'ai participé au développement d'outils pour visualiser les dépendances entre passes et les changements que certaines passes font sur un fichier C donné.

Mots-clés Compilation, Clang, LLVM, Passes, Neo4j

2.1 LLVM

LLVM est un projet de l'Université de l'Illinois à Urbana-Champaign, dont le but est de fournir un ensemble d'outils pour l'assemblage, la compilation et le débogage de programmes. Le projet a commencé en 2000 en tant que sujet de recherche encadré par Vikram Adve et Chris Lattner. Il a initialement été développé pour la recherche sur les techniques de compilation dynamique pour les langages de programmation statiques et dynamiques. Le projet LLVM est maintenant administré par la *LLVM Foundation*.

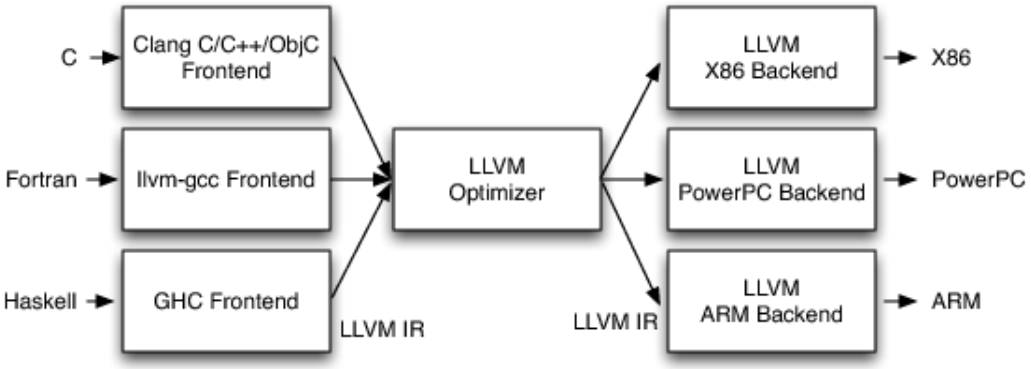

FIGURE 1 – *Architecture du compilateur LLVM*

La force de LLVM est son architecture modulaire (Figure 1). Elle est composée d'un *Frontend* permettant la compilation de langages source hétéroclites (C, C++, Ada …) dans un format intermédiaire commun (LLVM IR[8]) et indépendant[9] du langage source, d'un *Optimiseur* appliquant des passes d'analyse et d'optimisation sur la représentation intermédiaire générée par le frontend, et d'un *Backend* produisant les instructions pour un CPU cible donné (x86, ARM, WebAssembly …).

Par rapport à l'architecture historique des compilateurs où le développeur d'un nouveau langage de programmation devait écrire pour chaque CPU cible le compilateur adéquat, grâce à cette architecture le développeur doit seulement écrire un compilateur pour son langage source vers le langage intermédiaire de LLVM.

8. Manuel de référence pour le langage d'assemblage de LLVM : https://llvm.org/docs/LangRef.html
9. Basé sur la forme *static single assignment (SSA)* ou "forme à affectation statique unique" en français. https://en.wikipedia.org/wiki/Static_single_assignment_form

2.2 Les passes

Formellement, une *passe* est une fonction appliquée sur un bloc de code. On peut distinguer deux types de passes : les passes d'*analyse* et les passes de *transformation*.

Les passes d'analyse prédisent le comportement d'un programme sans le modifier, elles lisent et annotent le code, collectent les données annotées et utilisées lors des transformations. Les passes de transformation modifient le programme en utilisant les informations collectées pendant les passes d'analyse.

Les passes, qu'elles soient d'analyse ou de transformation, sont aussi classifiées selon leur façon d'analyser le code source :

— **Les passes sur les modules opèrent** sur les fichiers entiers, peuvent créer et supprimer des fonctions.
— **Les passes sur les fonctions** ne peuvent pas effectuer d'optimisations procédurales ni créer ou supprimer de nouvelles fonctions.
— **Les passes sur les boucles et blocs de base.**

Il y a une notion importante à souligner (sur laquelle nous reviendrons plus tard dans ce rapport) : certaines passes ont besoin d'autres pour s'appliquer correctement, il y a donc une notion de *dépendance* entre passes.

2.3 Ordonnancement des passes

Tous les outils de LLVM utilisant des passes utilisent le *PassManager* [10]. Le PassManager est l'ordonnanceur des passes, il en prend une liste, vérifie que les dépendances entre celles-ci soient validées puis les ordonne pour les lancer de manière efficace.

Il y a deux méthodes pour réduire le temps d'exécution de la série de passes :

— **Partager les résultats d'analyse** : le PassManager enregistre les analyses disponibles, celles qui ont été invalidées, et celles devant être lancées . Cela permet au PassManager de ne pas recalculer les mêmes choses si ce n'est pas nécessaire.
— **Créer un chemin d'exécution des passes** : Pour chaque passe, le PassManager va passer une à une chaque fonction du programme et y appliquer la passe, et ce, jusqu'à ce que l'entièreté du programme ait été analysé

L'outil `opt` permet de lancer les passes dans l'ordre spécifié par l'utilisateur. L'outil `clang` inclut des combinaisons de passes dites "standard" avec notamment les options -O0, -O1, -O2, . . .et permet aussi à l'utilisateur de pouvoir désactiver certaines passes lors de l'exécution de ces combinaisons [11].

2.4 Exemple : compilation d'un programme C

Pour illustrer le fonctionnement pratique de LLVM, nous montrons comment un programme C source (Figure 2) est transformé dans la représentation intermédiaire LLVM (Figure 3), puis finalement, après l'application d'un certain nombre de passes, en un code assembleur spécifique (Figure 4)

10. Documentation PassManager LLVM : https://llvm.org/docs/WritingAnLLVMPass.html#what-passmanager-does
11. Nous verrons plus tard dans ce rapport le travail d'Avril de Goër de Hervé sur les relations entre passes.

```c
int foo(int p) {
  int *a = &p;
  *a = 100; // instruction inutile ?
  *a = 200;
  *a = p;
  return *a;
}

int main(void){
  foo(10); //appel de fonction
  return 0;
}
```

FIGURE 2 – *Exemple jouet : le code source*

```
; ModuleID = 'src/dse_ptr.bc'
source_filename = "src/dse_ptr.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind uwtable
; Il s'agit de la fonction foo
define dso_local i32 @foo(i32 %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32*, align 8
  store i32 %0, i32* %2, align 4; stockage de p dans la variable %2
   ; [autres instructions coupées]
  %9 = load i32, i32* %8, align 4
  ret i32 %9
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {; main
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 @foo(i32 10) ; appel de fonction
  ret i32 0
}
```

FIGURE 3 – *Représentation intermédiaire de l'exemple jouet dans LLVM (extrait)*

2.5 Neo4j

Afin de réaliser les visualisations du fonctionnement de LLVM, nous avons fait le choix d'une représentation sous forme de graphe, pour ce faire nous avons décidés d'utiliser la technologie Neo4j. Neo4j [12] permet le stockage de données sous forme de graphe et la visualisation/récupération d'informations de celui-ci avec des requêtes utilisant le *Cypher Query Language*. Dans la base de donnée Neo4j les données sont stockées ainsi :

- **Les noeuds** : Les nœuds sont des entités de donnée. Ils sont composé d'un *label* et de *properties*.
- **Les relations** : Les nœuds sont reliés entres eux à l'aide de relations (*relationship*). Les relations ont, elles aussi, des propriétés spécifiques.

12. Site web Neo4j : https://neo4j.com/

```
out/dse_ptr.out:     format de fichier elf64-x86-64

<foo>: ; chargement des arguments, puis calculs
  push   %rbp
  mov    %rsp,%rbp
  mov    %edi,-0x4(%rbp)
  lea    -0x4(%rbp),%rax
  mov    %rax,-0x10(%rbp)
  ; instructions coupées ici
  retq
  nopw   %cs:0x0(%rax,%rax,1)
  xchg   %ax,%ax

<main>:
  push   %rbp
  mov    %rsp,%rbp
  sub    $0x10,%rsp
  movl   $0x0,-0x4(%rbp)
  mov    $0xa,%edi
  callq  401110 <foo> ; appel de fonction
  ...
```

FIGURE 4 – *Assembleur x86 généré pour l'exemple jouet : sans option supplémentaire, l'instruction inutile n'est pas supprimée.*

3 Historique des travaux du projet

Dans cette section, je rapporte les travaux des stages déjà effectués dans le cadre du projet CAPESA, notamment les contributions autour de la compréhension du compilateur LLVM [13]. Les travaux sont cités par ordre chronologique.

3.1 Outillage pour l'étude de l'impact de l'ordre des passes de LLVM par Bruyat [2019]

Dans le contexte d'un travail d'orientation en Master (POM) de l'université Claude Bernard Lyon 1, Julian Bruyat a réalisé les premiers travaux de ce projet. Il a réalisé une étude technique de l'infrastructure de LLVM et notamment des outils disponibles, qu'il a documentés. Il a aussi mis en place un ensemble de scripts permettant l'exécution d'une suite de tests par LLVM [14]. En particulier, il a utilisé LLVM Integrated Tester (LLVM LIT), l'outil de LLVM pour réaliser des tests, pour coordonner l'utilisation d'autres outils et agréger leurs résultats. Pour chaque fichier (source code C, par exemple), l'outil est capable de rapporter le temps d'exécution avec *TIMEIT* (intra-LLVM) ou avec *PERF*, le temps de compilation, ainsi que la taille des exécutables créés.

Julian a notamment travaillé sur l'optimisation de l'ordre des passes et comment intégrer un choix dynamique de l'ordre des passes. Il donne aussi des pistes d'amélioration pour son travail, comme le fait de pouvoir récupérer le temps d'exécution et la taille des exécutables, ainsi que la possibilité de développer de nouvelles métriques. Il propose également une comparaison de la

13. Les outils cités sont présents dans le git du projet CAPESA/LLVM, auquel j'ai accès.
14. La suite de tests ou *test-suite* en anglais, contient des programmes de référence et de test. Les programmes sont livrés avec des sorties de référence afin que leur exactitude puisse être vérifiée.

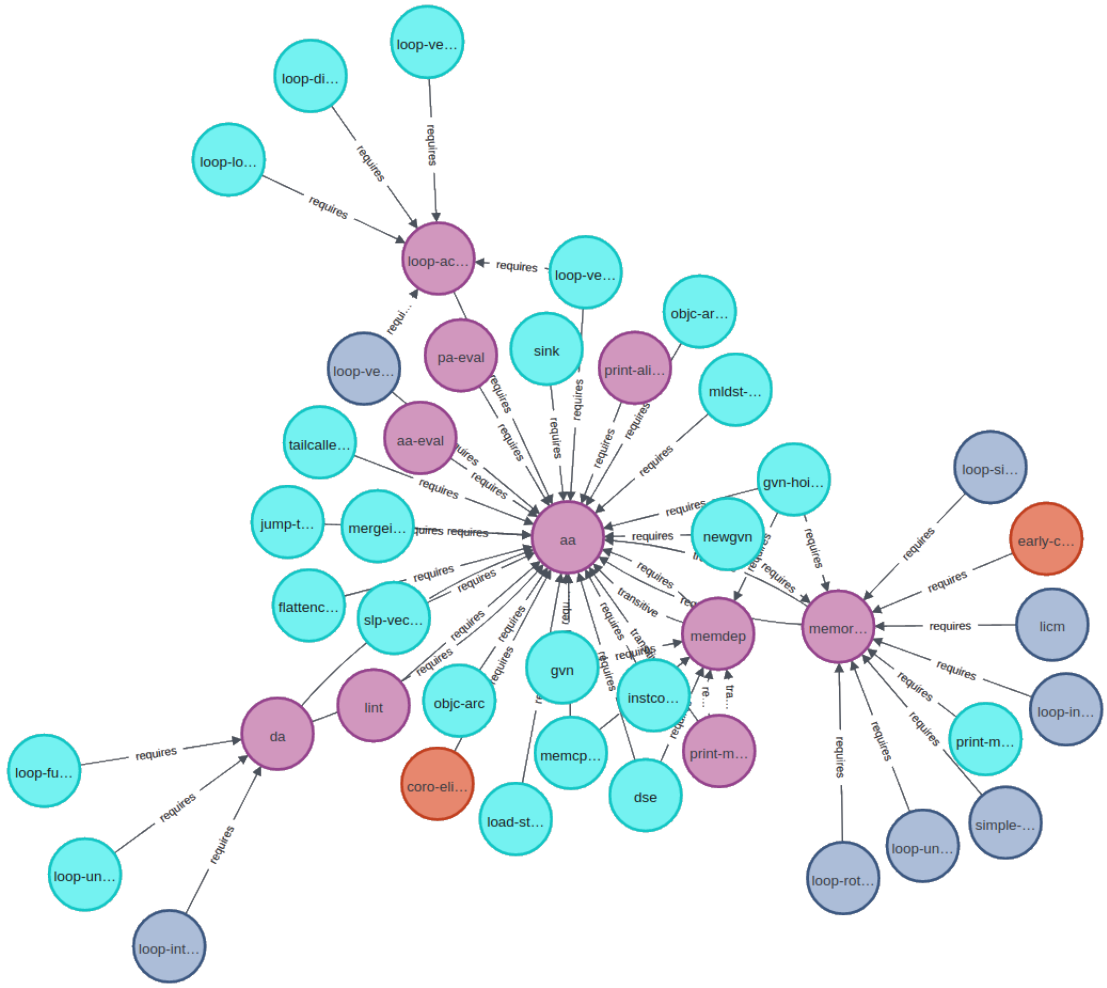

FIGURE 5 – *Représentation à l'aide de Neo4j des passes dépendant transitivement de la passe aa (Alias Analysis), image du rapport de Hervé [2019]*

représentation intermédiaire (LLVM IR) qui permettrait d'avoir des résultats plus fins, et pose l'ouverture d'une approche différente de celle d'un "brute force" [15] pour trouver le bon enchaînement des passes, notamment à l'aide d'un algorithme génétique.
Pour finir, Julian a proposé d'étudier la commutativité des passes et imagine associer à la notion de commutativité le pourcentage de programmes produisant des mesures similaires, en soulignant le cas des passes *licm* et *loop-unswitch* ayant une commutativité de 91%.

3.2 Exploration et cartographie des passes de LLVM par Michelland [2019]

Dans la suite des travaux de Julian, les travaux de stage de M1 de Sébastien Michelland ont justement porté en partie sur la commutativité des passes.

Dans son rapport de stage, Sébastion Michelland propose en premier lieu des outils permettant de mieux comprendre le système d'optimisation de LLVM, et cherche à montrer le comportement des passes et leurs interactions, puis dresse une liste des passes les plus utilisées par LLVM et réalise une carte *statique* des passes (cf Figure 5) pouvant être aisément manipulée à l'aide de Neo4j [16] permettant la visualisation des interactions inter-passes.

Sébastien Michelland a aussi développé des outils de mesure de performances pour répondre

15. l'approche actuelle consistant à tester par intuition l'ordre des passes jusqu'à en trouver un "meilleur".
16. Neo4j est un logiciel de management de base de donnée de graphe - https ://neo4j.com/

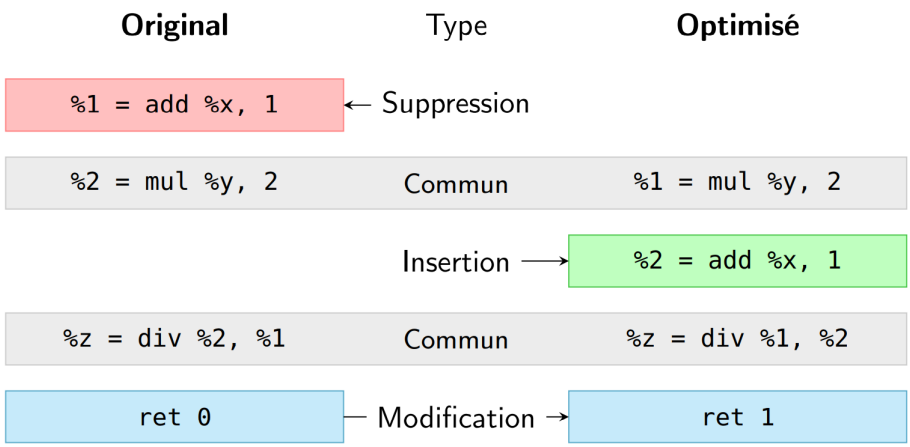

FIGURE 6 – *Diff pour le ré-ordonnancement de deux opérations, image du rapport Michelland [2019]*

aux problématiques d'identification et d'amélioration des points critiques dans la chaîne d'optimisation, calculer l'impact d'une passe sur les performances des programmes compilés . . .Sébastien propose donc l'utilisation de l'outil *perf*, plus fiable que l'outil de test de LLVM (*llvm-lit*) car celui-ci obtient ses mesures directement en collaborant avec le noyau du système d'exploitation, qui lui-même utilise les compteurs de performance du processeur. Il explique que l'utilisation d'outils de plus bas niveau permet d'avoir des informations précises et peu bruitées, comme :

— **Le temps d'exécution du programme** soit le *wall-clock time*.
— **Le nombre d'instructions exécutées**.
— **Le nombre de cycles d'horloges** écoulés pendant l'exécution.

Enfin, il a réalisé une première étude des *différences sémantiques* en utilisant l'outil LLVM-diff. Une différence sémantique est la formalisation du fait de passer d'un programme à un autre en analysant les changements effectués de l'un vers l'autre. Sur le diff d'un bloc par exemple, on identifie un ensemble d'instructions inchangées, et ce qui reste est supprimé, inséré ou modifié (cf. Figure 6). Il explique que l'outil *llvm-diff* implémente un algorithme spécifique pour le calcul des *diff* dont le principe est un parcours de graphe en largeur où les blocs sont unifiés au fur et à mesure.

3.3 Caractérisation fine des monopasses LLVM dans une cartographie par de Hervé [2019]

Dans la continuité des travaux de Sébastien Michelland, Avril de Goër de Hervé se propose de caractériser l'impact des passes en fonction de la neutralisation de certaines. Cette méthode permet de produire des informations expérimentales dynamiques et donc d'élargir la compréhension du système de passes et leurs interactions mutuelles afin d'affiner la cartographie du compilateur LLVM (Cf. Figure 7) [17]. Parmi les limitations qu'Avril identifie de ses travaux, il reste l'incertitude quant à la mesure des performances. Pour continuer sur l'étude des performances de l'impact de la neutralisation des passes dans LLVM, Avril de Goër de Hervé propose le développement de nouveaux outils de mesures plus précis.

4 Travaux personnels dans CAPESA et travaux futurs

L'objectif de ce stage qui m'a été proposé par Gabriel Radanne au sein du projet CAPESA est **le développement d'outils de visualisation d'informations pour les compilateurs** appliqués par-

17. Visualisation produite grâce à un travail interne au projet CAPESA par Jean Privat (UQAM)

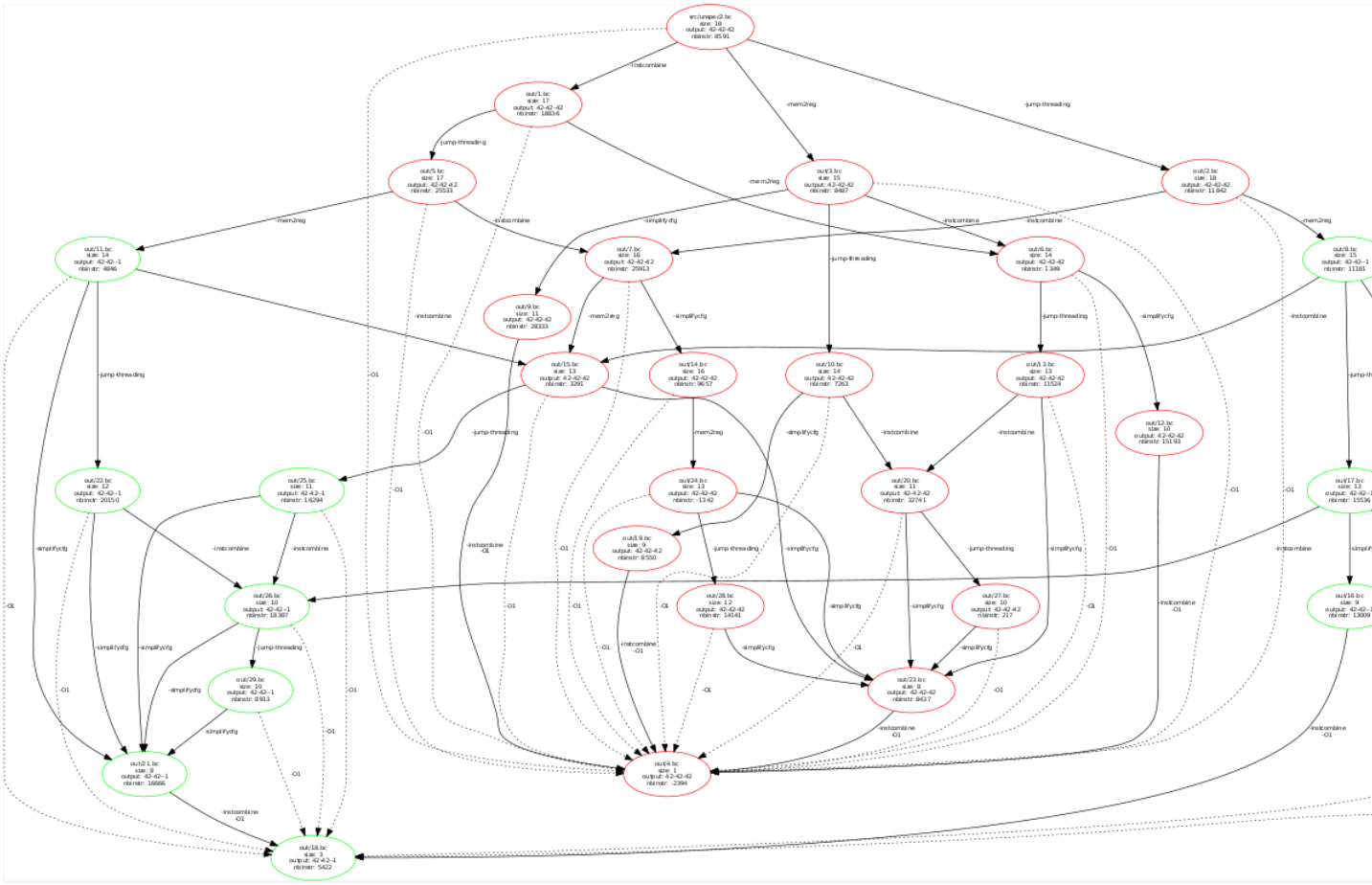

FIGURE 7 – *Relation entre fichiers en fonction de passes spécifique appliquées à la compilation*

ticulièrement au compilateur LLVM.

4.1 Organisation et conditions de travail

En raison de la situation sanitaire, le travail s'est réalisé pour la plupart du temps en distanciel (à raison d'un jour par semaine, le mardi, en présentiel).
Pour s'organiser, l'équipe a mis en place un ensemble d'outils de communication tel qu'un *Mattermost* pour l'équipe CASH et plus spécifiquement pour le projet CAPESA. Sur cette plateforme nous échangions avec les équipes de l'UQAM.
Nous réalisions une réunion à 14h les mardis afin de suivre l'avancement de mon travail, répondre à mes questions et échanger sur les besoins de chacun.
De manière personnelle, je prenais des notes à l'aide de l'outil *Notion*, me permettant de réaliser des listes de tâches, organiser mon travail, et avoir un suivi des notions que j'étudiais et de mon travail. Je partageais ensuite ces notes sur un outil interne [18] de l'INRIA afin que mes encadrants puissent y avoir accès.

4.2 Prise en main des travaux existants

Le projet CAPESA a un dépôt git sur le gitlab de l'INRIA dans lequel se situe l'ensemble des travaux réalisés dans le cadre de ce projet de recherche.
Afin de comprendre le fonctionnement de LLVM, ses rouages, ainsi que les efforts à fournir pour

18. Outil prise de notes de l'INRIA : https://notes.inria.fr/GK4pZ6MpQwWRy_qAFetaBg#

réaliser les outils de visualisation j'ai d'abord fait le choix de prendre en main les outils développés pour le projet par le passé : j'ai installé LLVM version 10 et testé le bon fonctionnement de LLVM en réalisant un travail de débogage sur des scripts afin de pouvoir compiler et tester un ensemble de programmes, appelé *testsuite*, j'ai créé une base de données Neo4j des relations entre passes à l'aide du script réalisé par Michelland [2019], j'ai pris en main et débogué un script générant un graphe (Figure 7) des relations entre différents fichiers en fonction des passes appliquées sur un fichier source. Pour finir j'ai pris en main des outils d'analyse d'exécution de programmes tels que *perf* ou *llvm-lit*.

4.3 Expression des besoins des spécialistes de compilation

Après la prise en main des outils disponibles dans l'architecture du projet actuel, l'une des missions m'ayant été confiée était l'expression des besoins des spécialistes de compilation pour le développement d'outils de visualisation. Il est vite apparu que l'ensemble des données produites par les travaux réalisés précédemment était d'un format très disparate, l'un des premiers besoins des spécialistes était donc d'agréger ces données dans un format commun.

Un des besoins plus précis des spécialistes était aussi l'étude et la visualisation des relations des passes dans le scénario d'une compilation utilisant le paramètre -O3. Il m'a aussi été demandé de mettre à jour les scripts d'aide à l'installation de LLVM afin de passer à la version 11, ceci permettant l'étude du nouveau gestionnaire de passes de celui-ci, offrant un outil donnant plus d'informations sur le fonctionnement des passes et permettant une meilleure manipulation pour l'application de celles-ci. Pour finir, il s'est avéré nécessaire de mettre en place un ensemble de requêtes dans le langage CQL (*Cypher Query Language*) afin d'aider les spécialistes à questionner une base de données Neo4j.

4.4 Développement des visualisations utilisant les métriques disponibles

Suite à l'expression des besoins des spécialistes de compilation, j'ai réalisé un système de *sondes* en poursuivant le travail précédemment fourni [19], permettant la réponse à la demande des spécialistes quant au besoin d'un format commun pour l'ensemble des données du projet et permettant donc l'importation de celles-ci dans une base de données Neo4j.

La sonde prend en entrée 2 principales données, un fichier contenant une liste de passes ou un paramètre de compilation spécifique (*-O0, -01, -03*), et un fichier code source (en C ou C++). La sonde, décrite à la Figure 8 est un script Python compilant le fichier C en y appliquant exclusivement une passe de l'ensemble des passes contenues dans le fichier, et en réitérant le processus tant qu'il y a encore des passes non appliquées sur le fichier C.

De manière concrète, pour un fichier C donné et un ensemble de n passes, le script produira et comparera n fichiers exécutables, entre eux. Chaque passe du fichier contenant les passes sera appliquée de manière unique et indépendante des autres pour produire un fichier exécutable à comparer. Soit, pour n passes, la sonde générera n fichiers exécutables. Le produit de cette sonde est deux fichiers CSV : un fichier contenant les informations sur les exécutables telles que la taille du fichier, le nom, la source, le temps d'exécution de celui-ci... Le second sur les relations entre fichiers. Ces deux fichiers sont importables dans la base de données Neo4j.

19. La notion de sonde a été définie à la suite du stage d'Avril de Goër de Hervé par les membres permanents du projet.

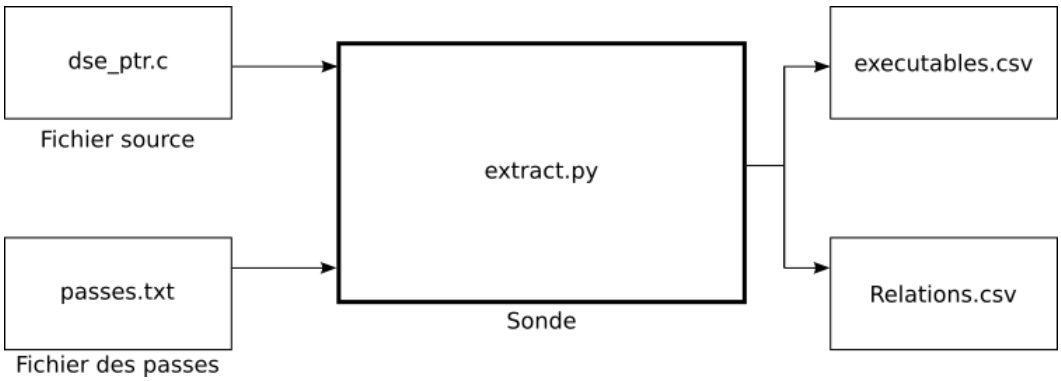

FIGURE 8 – *Schéma explicatif de la sonde*

4.5 Développement de nouvelles métriques et leur visualisation

Le développement de nouvelles métriques et leur visualisation constitue la suite du déroulement de mon stage.
Les pistes explorées seront

- La visualisation de la commutativité des passes (cf. Rapport de Bruyat [2019]).
- La visualisation des spécificités quant à l'utilisation de passes "exotiques" en fonction de langages source différent.
- La visualisation de l'impact précis d'une passe sur le temps d'exécution d'un programme.
- La visualisation du circuit d'application des passes en fonction de l'analyse statique des passes issue du travail de Michelland [2019] et des données fournies par la sonde développée dans le cadre de ce stage.

5 Conclusion

Dans le cadre de ce stage j'ai appris le fonctionnement des compilateurs, plus particulièrement celui de LLVM (du point de vu développeur), la création de bases de données de graphe, en utilisant la technologie Neo4j, ainsi que l'utilisation du langage CQL. J'ai développé mes compétences en Python et ai produit une sonde permettant l'étude dynamique des passes appliquées sur un programme source.
Ce stage m'a permis de comprendre le fonctionnement des laboratoires de recherche, et m'a conforté dans l'idée d'y travailler plus tard.
Le domaine de la compilation était nouveau pour moi et ce fut un plaisir de pouvoir le découvrir, même si je ne pense pas continuer dans ce domaine d'expertise, l'ouverture sur cette "matière" fut très enrichissante.

Références

J. Bruyat. Outillage pour l' ´etude de l'impact de l'ordre des passes de llvm. Technical report, Université Lyon1, 2019. URL https://laure.gonnord.org/pro/students/rapport_POM2019_Bruyat.pdf.

A. D. G. de Hervé. Fine-grain characterization of llvm single passes in a cartography context. Technical report, ENS de Lyon, 2019. URL https://laure.gonnord.org/pro/students/rapport_M12020_DeGoer.pdf.

LLVM. *LLVM Manual*. https://llvm.org/, 2021.

S. Michelland. Exploration et cartographie des passes de llvm. Technical report, UQAM, 2019. URL https://laure.gonnord.org/pro/students/rapport_M12019_Michelland.pdf.