Victor MALOD
INFO4
Rapport de stage 2021

Co-auteur : Nicolas PALIX

# Développement sur le compilateur de PhaistOS

Tome Principal
et
Annexe

27 pages

2020 - 2021
3 Mai - 31 Août

# Remerciements

Je tiens tout d'abord à remercier le Laboratoire d'Informatique de Grenoble de m'avoir accueilli, ainsi que mon tuteur Pr. Nicolas Palix qui a pu m'offrir cette opportunité, et m'encadrer tout au long de celui-ci. Sa disponibilité, ses conseils et sa pédagogie m'ont permis de gagner en expérience professionnelle.

Je remercie aussi Pr. Vincent Danjean pour ses conseils quant à la rédaction du rapport et la préparation de la soutenance, sans oublier sa fameuse tarte au c'meau, spécialité de sa région.

Je remercie Guillaume Vacherias, étudiant de ma promotion, en stage avec Nicolas, qui m'aura apporté conseils et compagnie malgré l'épidémie.

**Résumé**

Ce rapport de stage résume mon expérience de travail au sein du Laboratoire d'Informatique de Grenoble durant ma quatrième année d'études universitaires. Afin de valider mon année d'étude à l'école d'ingénieur de Polytech Grenoble, ce rapport sera fourni ainsi qu'une soutenance à un jury responsable de m'évaluer.

Je faisais parti de l'équipe Erods et ma mission consistait à tester et améliorer le compilateur d'un langage de programmation dédié à l'ordonnancement d'entrées/sorties d'un système linux. L'outil en question s'appelle PhaistOS et a été développé par Nick Papoulias, un chercheur postdoctoral qui ne travail maintenant plus sur le projet.

Durant ce stage différentes tâches m'ont été attribuées par mon tuteur, j'ai commencé par m'approprié le sujet en lisant la documentation et le code qu'avait écrit Nick. J'ai ensuite fait de la documentation puis du développement sur le compilateur. J'ai finis mon stage par des tests de performances et l'écriture d'un papier scientifique portant sur le compilateur.

≺ · — · ≻

This report summarizes my internship at the Laboratoire d'Informatique de Grenoble during my fourth year of university studies at Polytech Grenoble. This report will be evaluated, along with my defense before a jury of the school.

I was part of the Erods team and my mission was about testing and improving the compiler of a new specific language designed to customize policies in input/output scheduling. The tool's name is PhaistOS and is currently working for linux operating system. It was developed by Nick Papoulias, a postdoctoral research fellow that is no more working on the project.

During my internship, I was given different tasks by my tutor, and I started by understanding the tool by reading documentation and reading code lines wrote by Nick. After that, I wrote some documentation and then developed on the compiler. I finished my internship by testing the performances and by writing a scientific paper about the compiler.

# Table des matières

# Table des figures

# 1 La structure d'accueil

## 1.1 Erods

Mon stage a porté sur le projet "VeriAMOS" (Verified Abstract Machines for Operating Systems), au sein de l'équipe Erods, accompagnée par les équipes Antique (INRIA Paris) et Whisper (Université de Sorbonne). Erods est une équipe de laboratoire qui étudie principalement la construction et la gestion de sytèmes cloud, en travaillant sur différentes facettes de ces derniers, comprenant la prise en charge d'environement d'execution distribué robuste et efficace.

## 1.2 Laboratoire d'Informatique de Grenoble

Erods fait parti du Laboratoire d'Informatique de Grenoble (LIG) avec 23 autres équipes réparties sur 5 axes de recherches (voir Figure 1). Ce laboratoire, reconnu internationalement, se base sur les sciences informatiques et cherche à approfondir ses concepts, allant jusqu'à la réalisation de maquettes innovantes qui anticipent les usages. Le laboratoire de recherche universitaire se situe sur le campus grenoblois de Saint Martin d'Hères (Université Grenoble Alpes (UGA)) et est composé de 500 chercheurs et enseignants chercheurs.

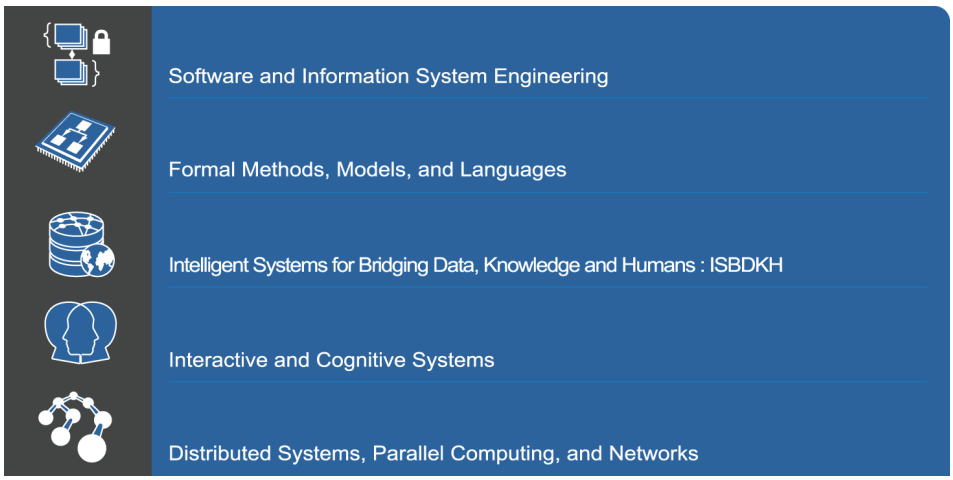


FIGURE 1 – Les différents axes de recherche du LIG.

## 1.3 Environement de travail

Mon employeur officiel reste donc l'université, c'est-à-dire l'UGA. Mon équipe et moi-même étions localisés sur le campus, plus précisément dans le bâtiment "IMAG", au quatrième étage consacré à la recherche dans le domaine informatique. Au début du stage, le mode de travail en présentiel m'était interdit par le labo à cause de la situation sanitaire liée à la Covid. Puis progressivement j'ai pu me rendre à mon bureau trois jours par semaine : les lundis, mardis et vendredis (voir sur la Figure 2).

FIGURE 2 – Bureau de travail et balcon vu de la fenêtre.

# 2 PhaistOS, le fruit du projet VeriAMOS

"PhaistOS" est le nom d'un Langage Dédié, ou Domain Specific Language (DSL) en anglais, c'est un langage de programmation créé pour le domaine de l'ordonnancement des entrées/sorties (E/S) d'un système d'exploitation (SE). Il a pour but de faciliter la conception et la validation de politiques d'E/S écrites par un utilisateur. Il est le fruit issu du projet VeriAMOS visant à s'attaquer au problème de vérification d'une classe de services de systèmes d'exploitation.

## 2.1 Pourquoi PhaistOS ?

Il faut tout d'abord expliquer pourquoi VeriAMOS a lieu d'être. En effet ce projet a pour but d'améliorer la vérification de systèmes d'exploitation, qui reste aujourd'hui compliqué pour deux raisons principales. Premièrement, les propriétés mathématiques des algorithmes sous-jacents sont généralement difficiles à exprimer, deuxièmement, la structure du code qui implémente ces algorithmes dans le SE est en général très complexe, comme par exemple avec Linux, écrit en C.

Afin d'éviter de se retrouver dans cette situation d'un niveau de difficulté trop élevé pour effectuer une vérification efficace, VeriAMOS propose d'utiliser une technique d'abstraction basée sur un DSL (comme le fait Bossa [2]), ainsi qu'un analyseur statique, qui, à froid, pourra vérifier un ensemble de services fournis par le SE, comme par exemple celui de l'ordonnanceur des requêtes d'E/S.

Dans cette approche de développement système, le programme écrit avec le langage du DSL pourra être compilé pour générer un code source dans le langage du système et qui implémente le service visé (plus de détails Partie 3). En limitant ainsi la capacité d'expression du langage, le DSL contraint le programmeur et permet d'éviter une mauvaise utilisation des outils du langage cible, permettant ainsi d'améliorer la robustesse des services concernés du SE. Le développeur pourra donc se focaliser sur l'écriture de sa politique système de haut niveau et déléguera la génération de code au compilateur du DSL.

VeriAMOS offre une opportunité de formaliser et d'implémenter une nouvelle approche à la vérification de systèmes d'exploitation, sur différents services. Ici nous

nous focaliserons sur celui de l'ordonnanceur d'E/S du système, mais nous pensons que cette approche reste générale et pourrait s'appliquer sur tous types d'ordonnanceurs, de systèmes de fichier, etc.

## 2.2 Mon rôle dans ce projet

Pour résumer, PhaistOS se décompose donc en deux parties : le compilateur et l'analyseur, ma mission portant uniquement sur le compilateur, ou autrement dit, le générateur de code C.

En effet, le but de ma mission était de faire évoluer le compilateur actuel pour qu'il puisse produire un code qui se rapproche de celui d'un module Linux. Un module Linux étant une extension du noyau du système qui peut être branchée et débranchée à volonté, à chaud, c'est-à-dire pendant que le système est chargé en mémoire et s'éxecute. À cet objectif se rajoute celui de tests d'analyses comparatives (qu'on appellera "benchmark" par la suite) ayant pour but de montrer que l'utilisation d'un DSL n'a que très peu d'impact sur les performances finales du code généré. Bien évidemment, d'autres tâches me seront assignées, comme celle de la documentation, ou de la participation à la rédaction du papier scientifique portant à décrire le fonctionnement de PhaistOS (plus de détails sur ma contribution au projet dans la Partie 4).

## 2.3 Pré-requis

Avant de commencer ma mission, il a fallu que je me familiarise avec mon environement de travail, surtout avec les outils que j'allais utiliser. Pour comprendre et bien aborder ce qui allait suivre, j'ai du lire des extraits de documentation, des pages de wikis, et même des fois des forums.

Tout d'abord, comprendre comment les ordonnanceurs d'E/S fonctionnaient sous Linux était la priorité. À savoir, que chaque ordonnanceur est considéré comme un module (intégré de base dans le SE ou non), et que l'ordonnanceur actif est contenu dans un fichier particulier [4] du système (changer le contenu du fichier change l'ordonnanceur actif).

Il a fallu ensuite que je me familiarise avec l'environement de travail virtuel qui m'était proposé, c'est-à-dire ce qui existait déjà sur le projet, comprenant la documentation, les exemples, le moyen employé pour implémenter PhaistOS dans le système et pour finir le code source de PhaistOS des différents dépôts de système de versions Git hébergés sur la plateforme Gitlab.

Plus tard, j'ai dû utiliser un réseau de serveurs français dédié à la recherche appelé Grid'5000 pour effectuer des tests sur des machines plus performantes que ma machine personnelle. Pour cela j'ai dû respecter une charte d'utilisation et apprendre les commandes propres à l'utilisation de la plateforme. À ce niveau-là du stage, une connaissance fondamentale des systèmes d'exploitation était nécessaire pour chaque installation sur les machines distantes.

Et pour finir, je devais maîtriser et mettre en place des outils de benchmark que nous avions désigné avec mon tuteur.

# 3 PhaistOS, le générateur d'ordonnanceurs d'E/S

## 3.1 Les planificateurs d'E/S

Dans leur fonctionnement, les ordonnanceurs de lecture (sortie) et d'écriture (entrée) tentent d'améliorer le débit en réorganisant l'accès aux requêtes dans un ordre linéaire basé sur les adresses logiques des données sur le disque, en essayant de les fusionner. Bien que cela puisse augmenter le débit global, certaines requêtes d'E/S peuvent attendre trop longtemps, provoquant des problèmes de latence et des fois de famine. Les planificateurs d'E/S tentent d'équilibrer le besoin d'un débit élevé tout en essayant de partager équitablement les requêtes d'E/S entre les processus.

Différentes approches ont été adoptées pour divers planificateurs d'E/S et chacun a ses propres forces et faiblesses et de manière générale, il n'y a pas de planificateur d'E/S par défaut parfait pour toute la gamme de requêtes d'E/S qu'un système peut rencontrer.

## 3.2 État de l'art

Historiquement, les premiers algorithmes d'ordonnancement qui ont été développés se sont basés sur la structure interne des disques durs de l'époque, que l'on retrouve aujourd'hui encore dans beaucoup d'ordinateur. Cette structure organisée par secteurs de données sur plateaux était lue par une tête de lecture unique, limitant ainsi les actions possibles sur le disque à une seule écriture ou une seule lecture à la fois. Les algorithmes de l'époque ont donc été développés en conséquence, et sont devenus au fur et à mesure du temps obsolètes. La sortie de nouvelles technologies à la mémoire flash, comme les disques SSD (Solid-State Drive) ont offert la possibilité de pouvoir effectuer plusieurs lectures et plusieurs écritures de manière simultanée. Ces nouvelles technologies ont donc donné naissance à de nouveaux algorithmes, comme BFQ en 2003, ou Kyber, développé en 2017 par Facebook.

Aujourd'hui dans Linux il existe plusieurs ordonnanceurs, tous ayant une utilité situationnelle et de manière générale, ils se distinguent entre deux catégories. Premièrement il y a les ordonnanceurs à file d'attente simple, ce sont les premiers qui ont existé et qui sont maintenant dépréciés dans les versions de Linux actuelles (depuis la version 5.3). Puis il y a les ordonnanceurs à file d'attente multiples, leur tâche est de distribuer les requêtes d'E/S aux différents fils d'exécution du noyau qui seront ensuite distribués aux différents processeurs. On retrouvera une liste exhaustive des planificateurs existant dans Linux en annexe dans la Table 2.

Tous ces ordonnanceurs ont cependant tous un point commun, ils sont écrits à la main par un développeur du noyau Linux. Le projet VeriAMOS essaye de changer cet aspect en proposant au développeur n'ayant pas toute la connaissance du noyau linux de pouvoir créer son propre planificateur facilement grâce à PhaistOS.

## 3.3 Arborescence du projet

PhaistOS est un DSL comme nous l'avons compris, il a été développé par Nick Papoulias (ancien membre de l'équipe Erods) comme sujet de post doctorat. Il s'est attelé au développement de PhaistOS avec le langage de programmation Ocaml et

a organisé l'architecture du projet d'une certaine manière, que l'on retrouve dans la Figure 13 en annexe.

Dans cette arborescence, on retrouve deux sections principales, et des sections secondaires. Concernant les sections principales, on retrouve évidemment le DSL, qui contiendra tout l'outillage nécessaire pour couvrir les objectifs de VeriAMOS, et on retrouvera aussi "Matryoshka", qui permettra au prochain developpeur de mettre en place son environement de travail pour effectuer ses tests assez facilement. Les sections secondaires comprennent les tests de performances ainsi que la documentation générale, mais aussi d'autres moins significatives, qui n'ont pas été couvertes par mon stage (comme par exemple les fichiers de configuration Gitlab, ou les images Docker qu'avait mis en place Nick à l'époque).

## 3.4 Fonctionnement du DSL

Pour que PhaistOS puisse produire un code C à partir d'une politique, il faut d'abord que celui-ci puisse lire cette politique et l'interpréter. Avant de présenter ce mécanisme, présentons le langage dédié. Pour l'illustrer et mieux le comprendre dans la partie qui va suivre, nous nous baserons sur l'exemple type de l'ordonnanceur MQ-Deadline (description de l'ordonnanceur en annexe, Table 2).

### 3.4.1 Le langage de PhaistOS

La politique utilisateur est écrite avec le langage du DSL PhaistOS et permettra de scripter le fonctionnement du planificateur d'E/S. Elle est inspirée des précédents DSLs Bossa [2] et Ipanema [3], desquels elle soutire une conception basée sur des événements. Le DSL PhaistOS définit donc 7 événements, listés dans la Table 1. En plus de devoir définir chacun de ces événements, le développeur doit aussi spécifier l'état de l'ordonnanceur (voir un exemple avec la politique deadline personnalisée, Figure 3) et certaines fonctions personnelles si nécessaire.

| **Événement** | **Description** |
|---|---|
| `INIT` | Initialisation de l'ordonnanceur |
| `EXIT` | Nettoyage à la suppression de l'ordonnanceur |
| `INSERT` | Enregistrement d'une nouvelle requête à traiter |
| `REMOVE` | Suppression d'une requête en attente |
| `DISPATCH` | Retourne la prochaine requête à envoyer pour le disque |
| `MERGE` | Procède à la fusion de deux requêtes |
| `HAS_WORK` | Indique s'il existe des requêtes en attente |

Table 1 – List des événements PhaistOS

```
1  int read_expire = HZ / 2;
2  int write_expire = 5 * HZ;
3  int writes_starved = 2;
4  int fifo_batch = 16;
5
6  POLICY {
7      list fifo_list[2];
8      int fifo_expire[2];
9      int fifo_batch;
10     int writes_starved;
11     int batching;
12     int starved;
13 }
```

FIGURE 3 – État de la politique Deadline implementé dans PhaistOS

Dans la Figure 3, les Lignes 1 à 4 décrivent des constantes spécifiques tandis que l'état interne de la politique est décrit par la structure `POLICY` des Lignes 6 à 13. Grâce à ces variables, le développeur pourra assez simplement reproduire le comportement de l'algorithme deadline visé en se servant de deux listes (Ligne 7), qui sont accompagnées d'une API proposant des opérations sémantiques comme `init`, `append`, `next_request` ou encore `is_empty`. La partie principale du planificateur se trouvera dans d'autres blocs que `POLICY`, comme par exemple le bloc `EVENTS` qui aura à charge de contenir la définition de chacun des événements de la Table 1, la Figure 12 en annexe donne un aperçu de comment sont gérés certains événements pour la politique deadline. Par exemple l'événement `INIT`, appelé à l'initialisation de l'ordonnanceur, a pout but d'initié l'état interne de la politique décrit dans la Figure 3. Durant son exécution, l'ordonnanceur recevera des requêtes et fera l'appel à l'événement `INSERT` pour les prendre en compte, et appelera `DISPATCH` pour élir la prochaine à envoyer au disque. `REMOVE` sera appelé après un `DISPATCH` ou un `MERGE`, pour vider les listes des anciennes requêtes. L'événement `MERGE` est appelé par le système et correspond à la fusion de deux requêtes ou plus qui peuvent voir leur adresses logiques regroupées en une seule.

### 3.4.2 L'outillage de PhaistOS

Pour construire un compilateur pour notre DSL, l'ensemble de l'outillage PhaistOS s'est décomposé en trois parties : un parseur en charge de l'analyse syntaxique et de l'interprétation, des analyseurs pour valider les politiques, et quelques programmes de génération pour aider les deux parties précédentes. La Figure 4 résume à travers un schéma le récapitulatif du fonctionnement du compilateur, où l'on retrouve es trois parties : le parseur en rouge, les analyseurs en violet, et le générateur en vert.

#### 3.4.2.1 Les fichiers AST

On remarque sur ce schéma (Figure 4), que certaines entités importantes, comme le parseur, sont issues d'une génération partielle, cela signifie que certains des fichiers du code source utilisés par le parseur sont générés. Le générateur en question s'appelle "le parseur d'arbre à syntaxe abstraite interrogeable" ("Queryable AST Parser" en anglais). Son but est de transformer une description d'AST (un arbre abstrait) contenue dans un

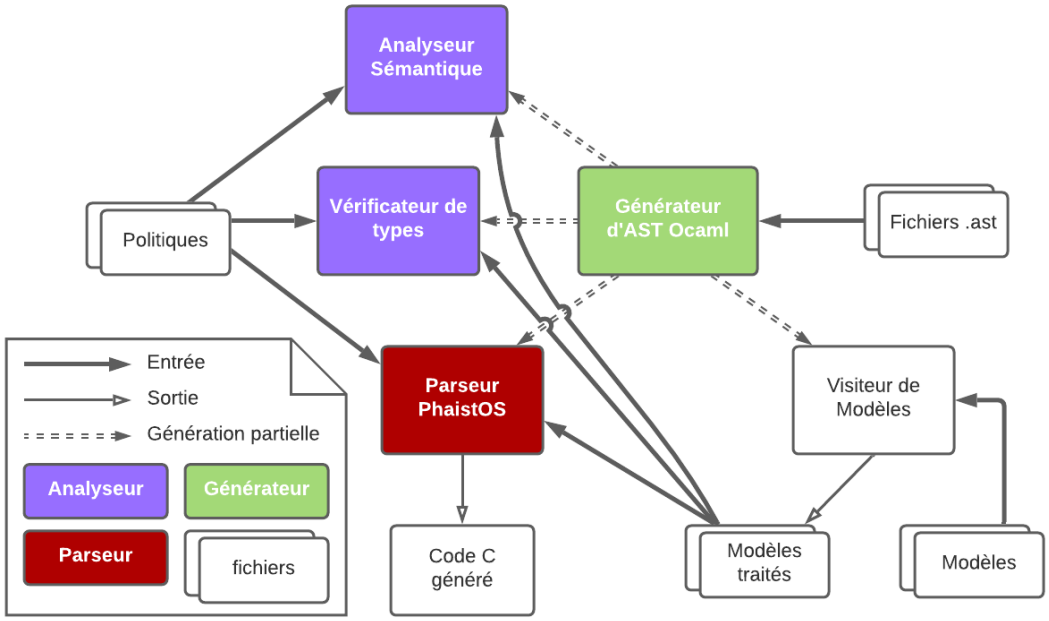


FIGURE 4 – Schéma de fonctionnement du compilateur.

fichier, en code source Ocaml pouvant modéliser chaque noeud de l'arbre (grace à un système d'objet propre au langage) et les parcourir à l'aide d'un visiteur dédié à cet arbre. Le fichier contenant ce code source généré s'appelle de manière générale `ast.ml`.

Par exemple, dans le cas du parseur, on fournira au générateur la description d'un AST qui définit la structure que doit avoir les politiques écrites par les utilisateurs. Cette description est contenue dans le fichier `phaistos.ast` , dont un extrait est disponible sur la Figure 5. Comme on peut le voir dans l'extrait à la Ligne 1, on retrouve la racine de l'arbre appelée `script`, le point d'entré qui va contenir toutes les déclarations de haut niveau d'une politique PhaistOS dans la variable `topLevelStmts`. Comme présentés dans la Partie 3.4.1, on retrouve dans ces déclarations les définitions globales, l'état de la politique et la liste d'événements (Ligne 3, Ligne 4 et Ligne 6 respectivement).

```
1  script (topLevelStmts: topLevel list)
2  topLevel () >
3      | globalDefinition (var: varDecl, initExpr: expression)
4      | policyDeclaration (vars: varDecl list)
5      ...
6      | eventDeclaration (events: event list)
7      ...
8  event (name: string, typeName: string option,
9         vars: varDecl list)
10 ...
```

FIGURE 5 – Extrait du fichier `phaistos.ast`.

Concrètement, le code Ocaml du générateur prend un fichier `.ast` en entrée et utilise les bibliothèques Menhir et Ocamllex pour parser son contenu. Il va ensuite générer une sortie sous forme de code source Ocaml qui contiendra une classe pour chaque noeud de l'AST (voir Figure 6) . Le code contiendra aussi une classe d'objet **visiteur** avec une

méthode `visit` pour chaque type de noeud, comme à la Ligne 6 où le noeud `script` accepte le visiteur en appellant la méthode `visit_script`.

```
1 script (topLevelStmts: topLevel list) =
2   object(self)
3     inherit astNode as super
4     val mutable topLevelStmts = topLevelStmts
5     // -- snip --
6     method accept visitor = visitor#visit_script (self :> script)
7 end
```

FIGURE 6 – Équivalent Ocaml du noeud `script`.

#### 3.4.2.2 Le Visiteur de Modèles

Cette visite de noeud est importante car elle permettra au parseur par la suite de la parcourir la politique utilisateur et d'effectuer des actions particulières pour chaque noeud. Le comportement à adopter pour chaque noeud est décrit dans un **modèle**, et chaque modèle arbore un langage simplifié. Les modèles sont contenus dans des fichiers à part et nous permettent de maintenir le DSL facilement sans se préoccuper des lignes de code principales. Pour comprendre comment ils fonctionnent nous allons prendre l'exemple du modèle utilisé pour le noeud `event` (Figure 7) qui va servir à générer le code C associé à un événement.

```
1 let! type =  [$
2   match my#typeName with | None -> "void" | Some x -> x $]
3 let! signature = ["
4   static inline [$use "type"$] on_[$String.lowercase_ascii
5   my#name$] (struct iosched_data* dd[{if List.length
6   my#vars > 0 then [",""]}] [$forEach my#vars ",  "$])
7 "]
8 let! default = [{
9   ignore (self#atPut my#name (String (use "signature"))) }]
```

FIGURE 7 – Le modèle `event`

Comme nous pouvons le voir aux Lignes 1 , 3 et 8, les modèles contiennent des régles dénotées par le mot clé `!let`. Chaque règle est nommée et `default` est celle qui est interprétée en première, elle contient du code embarqué entre les délimiteurs `[{` et `}]` prévus à cet effet. Dans notre cas, le code Ocaml ici présent remplit une table de hachage maintenue par le visiteur actuellement sur le noeud `event` grâce à la méthode `atPut`, `self` étant le visiteur et `my` le noeud. La table de hachage associe au noeud de l'AST son équivalent textuel en C, pour cela elle appelle la règle `signature` qui va produire une sortie textuelle (dénotée par les délimiteurs `["` et `"]`) de la signature de l'événement. Dans cette règle on peut insérer du code ayant pour but de générer une sortie sur place avec les délimiteurs `[$` et `$]` ou sans sortie particulière avec `[{` et `}]`. La règle `type` est appelée par `signature` à la Ligne 4, elle permettra d'écrire dans la signature le type de la fonction événement en cours de traitement.

Les modèles sont puissants, ils modularisent le comportement à adopter pour un noeud de l'arbre, tout en gardant un code relativement simple. On retrouve ce système de modèles dans d'autres parties du compilateur : **les analyseurs** (Partie 3.4.3) utilisent leur propres modèles pour exercer leurs vérifications statiques sur l'arbre.

Cependant, les modèles ne sont pas utilisables tels quels, le langage interne des modèles doit être interprété pour pouvoir être retranscrit en quelque chose de compilable par un langage. PhaistOS a donc dans son outillage un **visiteur de modèles**, capable de parcourir ces derniers comme le fait le parseur pour les politiques utilisateur. Mais au lieu d'utiliser le fichier `phaistos.ast`, il utilisera le fichier `templateVisitor.ast`, qui décrit la structure des fichiers modèle. Le visiteur de modèles parcours donc ces derniers et les traite pour en ressortir un code Ocaml intégrable par le parseur ou les analyseurs.

Concrètement, sur la Figure 6, la méthode `visit_script` à la Ligne 6 contiendra le contenu du modèle `script` traité par le visiteur de modèles.

### 3.4.3 Vérification statique

Avant qu'une politique soit sujette à être compilée, il faut qu'elle soit validée par les différents analyseurs statiques de PhaistOS. En réalité, comme on le voit sur la Figure 13 en annexe, il existe trois analyseurs :

1. Un analyseur sémantique de base, contenu dans `Static-Analysis` , écrit en Ocaml. Il permettra de vérifier l'arborescence et la sémantique de la politique de l'utilisateur final. Pour cela il vérifie des règles de bases, comme le parenthésage ou la présence de certains champs obligatoires.
2. Un analyseur de type contenu dans `TypeSystem`, écrit en OCaml et qui vient compléter l'analyseur précédent. Il vérifiera que les types utilisés dans la politique écrite par l'utilisateur sont corrects.
3. Un vérificateur de propriétés de sécurité concernant l'utilisation de l'API des requêtes d'E/S de Linux. Pour cela il utilise l'outil externe Celf [6] et est contenu dans `Linear-Logic-Analysis`.

Mon stage n'a pas porté sur les analyseurs statiques, cependant les analyseurs écrits en Ocaml se rapprochent de par leurs fonctionnements, au reste du DSL (comme nous avons pu le voir dans la Partie 3.4.2.2), j'ai donc pu travailler certains de leurs aspects.

### 3.4.4 PhaistOS dans Linux

A mon arrivée sur le projet, PhaistOS utilisait un noyau Linux modifié, qui lui permettait d'inscrire ses propres politiques à la compilation du noyau à travers des fichiers de configuration. Le code source de la politique générée était intégré dans celui du noyau.

L'avantage d'une telle pratique est que le noyau utilisé reste le même et que les dépendances système de PhaistOS ne sont pas pertubées par les dernières mise à jour de Linux. L'inconvénient en revanche est que chaque nouvelle politique écrite par un utilisateur doit être intégrée au noyau, qui devra être recompilé par la suite.

## 3.5 Et ensuite ?

Dans toute cette architecture (Figure 13, en annexe), PhaistOS fait preuve de forme et arrive a bien séparer les différentes tâches qui lui permettent d'arriver à sa finalité : générer un ordonnanceur d'E/S pour Linux.

Cependant, des aspects du compilateur restent à améliorer. Premièrement, la documentation du compilateur est très réduite, voir absente, que ce soit pour sa structure ou son API ; à mon arrivée sur le projet, PhaistOS était pour moi une énigme à résoudre. Étant donné que Nick n'était plus présent sur le projet, je n'avais que Nicolas Palix, mon tuteur, pour m'aider à assimiler les notions utilisée par le DSL (en faisant le parallèle avec le projet Ipanema [3]). Il faudra donc apporter un effort de documentation pour le projet, y compris dans le code, dont l'absence de commentaires rend la tâche de compréhension encore plus ardue. — "*Accompagné du mode de travail en distanciel et des tâches annexes, ce premier point m'aura prit un mois entier de mon stage.*"

Deuxièment, l'implémentation de PhaistOS dans le système n'est pas idéale car elle demande à l'utilisateur de recompiler tout son système pour qu'il puisse intégrer une nouvelle politique. L'objectif pour changer cela est de faire en sorte que les politiques générés par PhaistOS soient intégrable à chaud, dans le système.

Troisièmement, l'effort de compilation des différentes étapes de fonctionnement du DSL génèrent en somme une grande quantité d'avertissements, rendant illisible la sortie de la compilation Ocaml. Ce phénomène est principalement dû à la structure du code, qui fait de multiple inclusions d'un même fichier de fonctions, sans pour autant utiliser toutes les fonctions à chaque inclusion (plus de détails dans la Partie 4.2).

Pour finir, des détails concernant la structure ou l'implémentation du projet doivent être retravaillés, la plupart du temps ces aspects génants du compilateur sont le résultat d'une ancienne implémentation qui n'a pas été mise à jour par Nick :

1. L'utilisation majeure de scripts shell dans les dossiers du DSL empechait d'avoir une vision structurée de l'interaction que chacun pouvait avoir avec les autres. Ce problème peut être réglé avec l'ajout de Makefile.
2. Certains modèles présents dans PhaistOS sont erronés ou inutilisés.
3. Les fichiers `.ast` contiennent d'anciennes règles non-exploitées
4. Les grammaires contenues dans les fichiers `.mly` utilisées par la bibliothèque Menhir pour parser les politiques ou les modèles contiennent des conflits qu'il faut corriger, sans quoi cela ajoute encore plus de verbosité à la compilation du DSL.

Dans l'ensemble, à mon arrivée, il y avait matière à travailler. L'état actuel de PhaistOS a fait que ma mission se prétait bien à sa description dans la Partie 2.2. Chacun des points cités précedemment à fait l'objet d'une de mes contributions durant le stage, dans le but d'améliorer le compilateur, et de le rendre plus accessible pour de nouveaux mainteneurs, qui comme moi doivent travailler sur le DSL.

# 4 La modernisation et les performances du Disque de Phaistos

≺ · — · ≻

PhaistOS tire son nom de la découverte archéologique du disque de Phaistos, rappelant par son aspect un disque dur. Les lettres "OS" sont en majuscules pour rappeler le système d'exploitation (Operating System), une belle coincidence bien exploitée par Nick.

> *Le disque de Phaistos ou disque de Phaestos est un disque d'argile cuite découvert en 1908 par l'archéologue italien Luigi Pernier sur le site archéologique du palais minoen de Phaistos, en Crète.* – (Wikipédia)

≺ · — · ≻

Durant ce stage sur le projet PhaistOS, j'ai pu entretenir un journal récapitulant de mon expérience quotidienne, ce journal est disponible au lien suivant :

https://github.com/McReaper/InternshipINFO4

## 4.1 Documentation du projet

Une de mes premières contributions de mon stage, qui est aussi la plus importante, consistait à comprendre et documenter les différentes sections du compilateur PhaistOS. À mon arrivée sur le projet, chaque sous-dossier du DSL (voir Figure 13, en annexe) comportait un fichier `README. md` vide. Je n'avais donc aucune description précise du rôle de chaque sous-dossier, et la légère documentation laisées derrière lui par Nick n'était pas assez claire ou des fois trop complexe, pour que je puisse comprendre de quoi il en retournait. J'ai donc commencé à lire le code de chaque répertoire, et progressivement j'ai commencé à comprendre quel était le rôle de chacun. J'ai donc remplit le `README.md` de chaque partie du compilateur en amenant dans ma description le plus de détails possible, en décrivant même le rôle de chaque fichier du répertoire dans certains cas.

Dans chaque dossier il existait des scripts permettant de faire les appels aux exécutables, comme le visiteur de modèle par exemple, ou tout simplement l'exécutable local. Ces scripts ont été remplacés pour laisser place à des `Makefile`. Ces Makefile peuvent s'appeler entre eux, comme les scripts le faisaient avant, et possèdent une description propre à chacun dans le `README.md` du dossier associé.

## 4.2 Restructuration du code source du compilateur

Durant mon stage, je me suis rendu compte que la compilation du code Ocaml était trop verbeuse. En effet elle génèrait une quantité phénoménale d'avertissements, assez pour que le buffer de mon terminal ne puisse pas suivre la route au bout de deux compilations. J'ai donc pris la décision de restructurer le code d'un des dossiers du DSL pour montrer qu'il est possible de supprimer tous ces avertissements à la compilation du code source.

### 4.2.1 Le problème

Dans la plupart des dossiers du compilateur (`Abstract-Template-Visitor` , `Parser`, `Linear-Logic-Analysis`, `Static-Analysis`, etc.), on retrouve le fichier `ast.ml` généré par `Abstract-Syntax-Hierarchy`, dont j'explique l'origine Partie 3.4.2.1. Ces fichiers `ast.ml` se basent sur une structure prédéfinie, qui contient les classes pour chaque noeud de l'AST, ainsi que le visiteur pour les parcourir. Cependant la structure de ces fichiers n'est pas bonne car à la compilation un grand nombre d'avertissements sont générés. Tout ces avertissements proviennent de la ligne :

```
#include "visitorInternalDsl.ml"
```

Cette insctruction présente à chaque début des méthodes `visit` permet d'inclure l'API interne écrite en Ocaml nécessaire pour les modèles. Pour revenir sur l'exemple de la Partie 3.4.2.2, la méthode `atPut` vient de cette API. Seulement voilà, cette API est incluse dans le code source durant la précompilation autant de fois qu'il y a de méthode `visit`, c'est-à-dire environ trente fois. Et évidemment, chaque modèle n'utilise pas toutes les fonctions fournies par l'API, alors Ocaml produit un avertissement pour le développeur.

A ce moment là tout le monde se dit qu'inclure cette API au début du code réglerait pas mal de problèmes, sauf que la solution est bien plus complexe qu'elle en a l'air, et je comprends pourquoi Nick a procédé de cette manière.

En fait, le code possède une dépendance cyclique entre les objets qui représente les noeuds de l'arbre, et le visiteur. Car les noeuds doivent connaitre la classe `visitor` et le visiteur doit connaitre chaque classe de noeud, toutes les classes sont donc déclarées de manière récursive dans le code. De plus, l'API doit connaitre à la fois les noeuds et le visiteur, pour certaines fonctions. Cependant le langage Ocaml ne permet pas de déclarer récursivement des classes avec des fonctions, la connaissance récursive de deux entités à la sémantique différente n'est pas possible en Ocaml. C'est pour cette raison je présume que Nick a décidé de mettre le code de l'API directement dans les méthodes du visiteur.

### 4.2.2 La solution

Cependant il existe une solution capable de régler tous les problèmes qui n'était pas aisée de trouver. Après avoir tenté de déclarer l'interface de l'API en début de fichier puis de la définir en fin de fichier sans succès (après la déclaration des noeuds et du visiteur). Je me suis penché sur l'utilisation des modules, et j'ai appris qu'il était possible de les déclarer récursivement à condition que leur signature (interface) soit complète. J'ai donc obtenu la signature des classes d'objets des noeuds, de celui du visiteur et de l'API avec le compilateur `ocamlc` et l'option `-i`. J'ai ensuite revisité le code généré du fichier `ast.ml` pour déclarer sous forme de modules récursifs le visiteur, les noeuds et l'API, satisfaisant ainsi la dépendance cyclique.

A partir de ces changements, il suffisait de régler à la main les derniers avertissements, comme par exemple ceux de la bibliothèque Mehnir qui nous indiquait qu'il éxistait des conflits "shift/reduce" dans la grammaire utilisée pour parser les politiques PhaistOS. Ces conflits étaient juste dû à des ambiguïtés liés à deux règles de la grammaire que Mehnir n'arrivait pas à départager (quelle règle choisir en fonction de ce qui est lu par

le parseur).

### 4.2.3 L'impact

Grâce à ces changement, le mainteneur du compilateur gagne en visibilité lors de sa phase de rédaction du code : dès la compilation, il pourra s'apercevoir si son programme Ocaml a un problème ou non. Cependant, ces changements dans la structure du code ont une impact qui n'est pas des moindres puisqu'ils demandent de changer la génération de code pour reproduire la structure que j'ai mis en place dans toutes les autres parties du DSL, cependant ces changements que j'ai effectué dans le répertoire `Static-Analysis` ne conviendront peut être pas pour d'autre fichiers `ast.ml`.

—

J'ai donc montré qu'il était possible de se débarasser de **tous** les avertissements à la compilation d'une des parties du DSL. Un travail futur sera de reproduire cette solution de manière générale pour le reste du DSL en changeant le générateur d'AST.

## 4.3 Implémentation des politiques sous forme de modules Linux

A mon arrivée sur le stage, les politiques utilisateurs PhaistOS étaient générées séparemment dans plusieurs fichiers. Un fichier principal qui ne change jamais, qui est le squelette qui accueil quatre inclusions de fichiers secondaires, qui eux diffèrent en fonction de la politique utilisateur. Ces quatres fichiers étaient répartis dans un dossier ajouté dans le noyau Linux juste en dessous du dossier contenant les autres ordonnanceurs du système, comme par exemple BFQ, MQ-Deadline ou PhaistOS (voir Figure 8). A chaque génération, PhaistOS produisait le code des fichiers secondaires séparemment et l'utilisateur devait alors les remplacer et compiler le noyau Linux pour que sa politique puisse prendre effet.

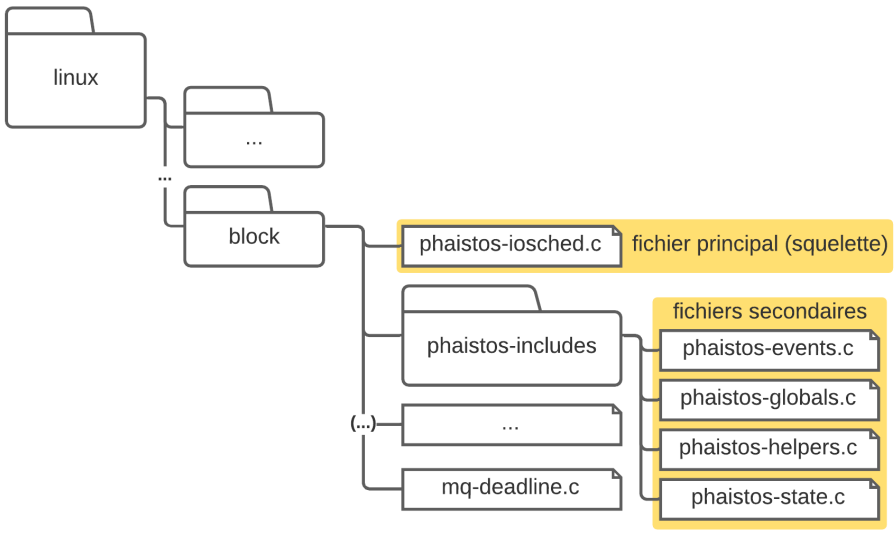


FIGURE 8 – Schéma de l'arborescence du dossier `block` dans le code source Linux, augmenté par PhaistOS

Mon objectif était ici de changer cette méthode d'implémentation, et de faire en sorte que le code généré puisse être indépendant (on peut créer autant de politique qu'on le souhaite, sans que celles-ci se marchent dessus) et qu'il soit intégrable à chaud dans le noyau.

Pour cela j'ai commencé par me renseigner sur les autres ordonnanceurs, et j'ai appris que l'ordonnanceur Kyber, développé par Facebook, était un module que l'on pouvait imbriquer et désimbriquer à volonté.

Premièrement, j'ai appris comment créer mon propre module : il me fallait une recette, autrement dit un Makefile, ainsi qu'un fichier `.mk` contenant une liste de dépendances si nécessaire. Le code généré devait avoir certaines particularités pour s'enregistrer auprès du système en tant que module, et bien heureusement PhaistOS comportait déjà ces pré-requis dans son fichier principal qui servait de squelette.

Deuxièmement, j'ai du modifier la génération pour qu'elle soit plus dynamique, cela inclut le fichier principal qui devait pouvoir s'enregistrer auprès du système sous différents noms, pour ne pas provoquer d'erreur à l'insertion du module. Comme on peut le voir dans la Figure 9, le modèle `script` qui sert de point d'entrée dans l'AST (comme on l'a vu dans la Partie 3.4.2.1) a été modifié pour que celui-ci génère le squelette ainsi que les fichiers secondaires. Dans cet extrait on retrouve les délimiteurs `[$` et `$]` qui je rappelle vont produire une sortie textuelle de leur contenu, autrement dit, dans ce cas il s'agit du nom du module choisi par l'utilisateur. Dans ce même modèle, la génération des fichiers secondaires est restée quasiment inchangée.

```
1      ...
2      .elevator_attrs = deadline_attrs,
3      .elevator_name = \"[$ !Globals.module_name $]\",
4      .elevator_alias = \"[$ !Globals.module_name $]\",
5      .elevator_owner = THIS_MODULE,
6  };
7  MODULE_ALIAS(\"[$ !Globals.module_name $]\");
8  ...
```

FIGURE 9 – Extrait du modèle `script` s'occupant de la génération du squelette

Ici, le code C qu'il y a dans le modèle de la Figure 9 va servir à définir le contenu d'une structure qui va être donnée à une fonction du système qui va lire cette structure et enregistrer un nouvel ordonnanceur d'E/S. Cette fonctionnalité n'est pas présente pour tous les types d'ordonnanceurs et peut rendre la tâche beaucoup plus complexe quand elle est manquante. Par exemple, pour le projet Ipanema [3], le noyau Linux est modifié pour pouvoir changer dynamiquement la politique d'ordonnancement des processus. Ce travail d'intégration demande beaucoup plus de temps et d'effort, mais heureusement, les développeurs Linux l'ont déjà fait dans notre cas (merci).

Troisièmement, il fallait ajouter la génération d'un fichier `.mk` qui allait permettre de lister les dépendances nécessaires au bon fonctionnement du module. Pour cela j'ai donc ajouté une règle `let! mk_file` au modèle `script`, produisant directement une sortie sans parcourir l'AST.

Pour finir, il a fallu écrire le Makefile pour gérer les différentes phases de génération du code C propre au module (fichier principale, secondaires et fichier `.mk`), ainsi que son exportation vers le dossier `Output-Modules`, dans un sous-dossier spécifique,

accompagné d'un Makefile pour compiler le module et intégrer le nouvel ordonnanceur d'E/S au système.

—

J'ai donc trouvé un moyen relativement simple de générer un code C directement compilable et intégrable dans le noyau Linux en tant que module. Pour cela un Makefile dédié à cet effet et une modification de la génération par les modèles sont à l'origine de ce changement. De plus, la modification d'un noyau Linux n'est plus nécessaire puisque la notion de module existe de base dans le système.

## 4.4 Mise en place d'une machine virtuelle pour les tests

Pour effectuer des tests de fonctionnement ou de compatibilité, j'avais besoin de travailler sur un environement de travail robuste (de bonnes performances pour les compilations noyau ou autre) et sûr (pas de pertes en cas de crash). Pour la robustesse j'ai donc utilisé les serveurs Grid'5000 et pour la sûreté j'ai décidé de reprendre ce qu'avait laissé Nick derrière lui, c'est à dire une image système enregistrée par le logiciel de virtualisation Qemu.

L'image laissée en l'état par Nick comportait le noyau Linux modifié ainsi que l'ordonnanceur Deadline généré par le DSL, il y avait aussi certains scripts présents pour changer d'ordonnanceur ou pour effectuer des tests de performances rapides. Le problème avec cette image c'est qu'elle ne m'a servit qu'au début du stage pour comprendre comment PhaistOS fonctionnait, mais j'avais besoin de créer ma propre image, avec mon propre environement pour tester la nouvelle implémentation exposée en Partie 4.3.

Pour cela j'ai modifié le contenu du dossier "Matryoshka" (voir Figure 13) sur ma branche de développement Git. J'ai principalement apporté des modifications aux scripts de mise en place de l'image Qemu, mais j'ai aussi supprimé le contenu du dossier `linux` qui contenait le SE modifié (le dossier me servira encore pour le téléchargement du noyau qui sera compilé par les scripts).

Le script principal de mise en place de Qemu et d'installation du système a été le plus compliqué à écrire. En effet ce script va permettre d'effectuer une installation propre de tout un système debian de base avec un noyau Linux version 5.13 non modifié. Ce système sera compilé et sauvegardé dans une image Qemu par ce script, pour qu'il puisse être démarré ou monté à volonté. La difficulté d'écriture d'un tel script résidait dans la maîtrise des outils qui lui sont nécessaires. Le script est disponible en annexe (voir Figure 14) mais voici un récapitulatif de ses différentes étapes :

1. Installation de dépendances, définition des variables et nettoyage des précédentes installations.
2. Téléchargement et compilation d'un noyau Linux avec la bonne configuration pour accueillir des modules.
3. Mise en place d'une image Qemu :
   (a) Création du fichier contenant l'image.
   (b) Création d'une partition au format ext4 dans l'image.

   (c) Téléchargement d'un système de base d'une debian buster (version 10) sur cette partition.
4. Montage de l'image et configuration interne du système (modifications de fichiers et d'installations importantes pour que la VM fonctionne correctement). Cela comprend la liaison virtuelle nécessaire pour que Qemu puisse voir le code source Linux (contenu dans le dossier `linux`) nécessaire pour la compilation des modules.
5. Copies des modules actuellement générés dans l'image où ils attendront d'être compilés par l'utilisateur.

Des scripts secondaires existent et ont pour but de lancer la machine virtuelle avec l'image préalablement créée. Certains permettent aussi de monter l'image dans un dossier pour qu'elle puisse être visible comme le contenu d'un disque dur.

—

La machine virtuelle reste une valeur de sûreté car elle permet au développeur d'effectuer ses expérimentations sans mettre en danger son système, ce bac à sable est l'endroit idéal pour développer des politiques d'ordonnancement personnalisées. Grâce à cette mise en oeuvre le prochain développeur qui travaillera sur PhaistOS pourra mettre en place une machine virtuelle facilement avec la configuration requise pour expérimenter ses politiques et faire des tests avec le DSL. Une fois son DSL stable il pourra ensuite passer à des choses plus sérieuses en le testant sur des machines physiques.

## 4.5 Tests d'analyse de performances

Le but dans cette partie est de montrer que l'utilisation d'un DSL n'a que très peu d'impact sur les performances finales des politiques PhaistOS. Pour montrer cela avec Nicolas, nous avons comparé la politique "Deadline" de PhaistOS et de Linux. Nous avons vérifié et compilé la politique du DSL, puis nous avons déployé ensemble un système debian version 10 avec un noyau Linux 5. 13 sur une machine des serveurs grenoblois de Grid'5000. Le code C de la politique a été importé dans un des dossiers du répertoire personnel et nous l'avons compilé en tant que module noyau. Une fois chargé, le module de l'ordonnanceur doit être activé à travers le contenu du fichier `queue/ scheduler` de chaque disque.

Pour effetucer les tests, nous avons utilisé l'outil `fio`, dont l'auteur est aussi celui de la politique MQ-Deadline de Linux. Nous avons configuré l'outil pour qu'il puisse effectuer des accès en séquences de lecture, d'écriture, puis des deux, et ensuite nous avons répété l'expérience mais avec des accès aléatoires. Les six expériences ont été effectuées sur 10 secondes, sur un fichier de 128Mio avec des blocs de données de 4Kio et une taille de file de 4 requêtes maximales (on peut avoir un maximum de 4 requêtes d'E/S à la fois sur le fichier). La machine utilisée possèdait un disque SSD SATA de 480Go de capacité de la marque Samsung.

Comme illustré dans les Figures 10 et 11, les deux algorithmes ont essentiellement les mêmes performances en terme de latence et de débit. Une description sous la Figure 10 explique comment les tests sont organisés, et ils devraient être lus paire par paire.

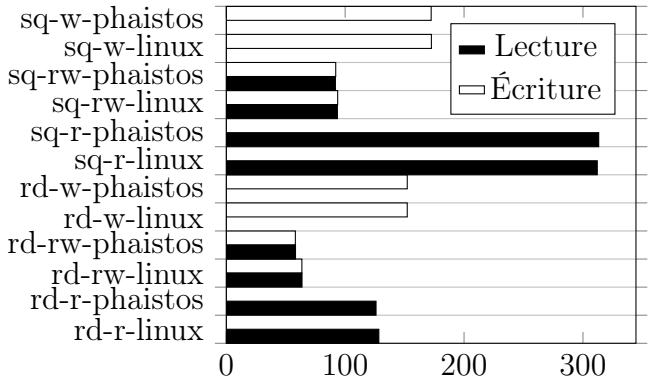


FIGURE 10 – Comparaison des performances de débit entre les ordonnanceurs "deadline" PhaistOS et Linux

Les tests sont nommés sous la forme *mode-direction-ordonnanceur* où *mode* est soit séquentiel (sq) ou aléatoire (rd) ; la *direction*, c'est la lecture (r), l'écriture (w) ou les deux ; et *ordonnanceur* indique lequel de PhaistOS ou Linux est utilisé.

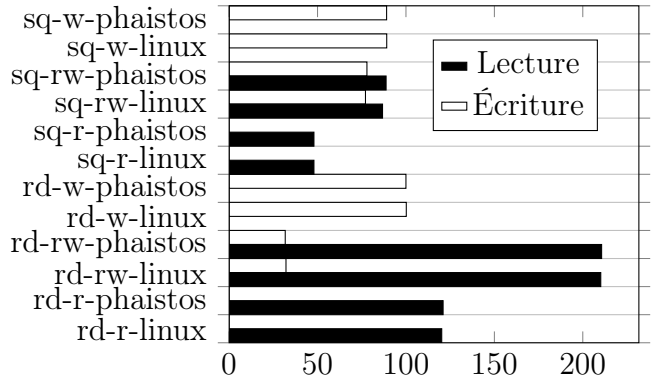


FIGURE 11 – Comparaison des performances de latence

—

Nous avons donc montré rapidement que l'utilisation du DSL n'a pas d'impact significatif sur les performances finales du binaire provenant de la génération du code. Cet aspect est important car il permet d'ajouter un avantage de plus à l'utilisation du DSL.

# 5 Rétrospection

## 5.1 Travail inachevé

Si j'avais pu continuer à travailler sur le compilateur PhaistOS, je pense qu'un grand nombre d'améliorations auraient pu lui être apportées. Cependant le temps et la priorité des tâches ont fait que certains objectifs fixés pendant le stage non pas pu être

réalisés. Voici une liste non-exhaustive de ces idées d'améliorations qu'on aurait bien aimé implémenter avec Nicolas :

1. Créer un dossier de test avec plusieurs politiques pour pousser les analyseurs dans leur retranchement, testant ainsi leur robustesse.
2. Créer une preuve de concept visant à montrer que le visiteur de modèle et la création d'un visiteur pour l'AST n'étaient pas nécessaire ; et que l'on pouvait remplacer ces visiteurs à l'aide d'une bibliothèque [5], déléguant ainsi une grande charge de travail.
3. Régler tous les soucis de compilation ocaml présents dans les différents dossiers, comme cela a été fait avec `Static-Analysis`.
4. Créer de nouvelles politiques avec PhaistOS, basées sur celles qui existent déjà, pour pouvoir tester les limites du DSL plus en profondeur, et explorer les nouvelles pistes d'améliorations.
5. Changer le code du point d'entrée des programmes ocaml pour offrir une interface en ligne de commande complète (`-help`, `-o`, etc.).
6. Étendre la batterie de tests de performances avec d'autres outils, comme NAS, Phoronix, Parsec, Sysbench, etc.

Cependant, ce n'est pas la priorité du chercheur de rendre le code plus qualitatif ou robuste. Par contre, toutes ces idées pourraient faire l'objet d'un futur stage, facilitant ainsi le travail du chercheur sur la partie importante du projet de recherche.

## 5.2 Gestion de projet

Au début du stage je n'avais pas accès aux locaux du bâtiment d'accueil, je travaillais donc depuis chez moi. Je passais des appels avec mon tuteur sur la plateforme bbb de l'UGA et on discutait de l'environement de travail et des premières tâches à réaliser. L'outil Slack nous aura aussi bien aidé pour apporter une aide à l'écrit, des fois nécessaire quand on veut partager du texte.

Les tâches principales étaient déjà définies mais restaient larges (Etude de l'existant, améliorations du compilateur, tests de performances), je les ai donc sous-divisées. J'ai adapté mes travaux au fur et à mesure, et pour illustrer ça, on peut comparer le planning de travail estimé au réel à travers les Figure 15 et 16, disponibles en annexe.

Il est à noter qu'à partir du 1er juillet, je pouvais me rendre trois fois par semaine au bureau, ce qui a grandement accéléré les choses. Changer d'environement de travail m'a bien aidé car rester chez soi devant l'ordinateur n'était pas motivant, et voir les collègues de l'école au bureau rendait le travail plus amusant.

## 5.3 Bilan personnel

Je pense que durant mon stage, j'ai été capable de voir les choses d'un point de vue plus global, plus réfléchis, contrairement à mon ancien stage de fin de DUT où j'avais le nez dans le code sans trop me préoccuper à comprendre l'environement et le contexte.

EE6 : Je pense que j'ai été capable de trouver l'information qu'il me fallait quand je bloquais, que pour trouver cette information je posais les bonnes questions (que ce soit

sur Google où à Nicolas) et que j'étais capable de les exploiter sans avoir besoin d'aide extérieure. On peut penser aux modules récursifs dans la Partie 4.2.2 où au script de mise en place d'une machine virtuelle Partie 4.4 où Nicolas m'a bien aidé concernant les outils à utiliser.

`EE4.4 :` J'ai aussi eu l'occasion de documenter et d'expliquer le compilateur à Nicolas qui n'avait pas eu le temps de gagné en expertise sur le projet auparavant. Cette capacité à présenter, documenter et même rédiger sur le compilateur a été une partie importante du stage car j'ai pu avoir l'opportunité de participer à l'écriture d'un papier scientifique rédigé en anglais concernant l'outillage du compilateur PhaistOS.

Grâce à cette expérience qui m'a été offerte par M. Palix, j'ai pu découvrir ce qu'était le travail dans le domaine publique. Cette opportunité m'aura beaucoup apportée, en terme de connaissance comme d'expérience, et j'en suis très reconnaissant. Il y a 2 ans déjà, j'avais réalisé un stage dans le domaine privé, dans l'entreprise Atos à Grenoble. Ces deux expériences m'ont permis de réaliser les différences significatives qu'il existait entre les deux domaines. Malheureusement, l'année prochaine je n'aurai pas l'opportunité de pouvoir choisir de retourner dans le domaine publique, car d'après la charte de l'école concernant les stages, une seule expérience en laboratoire est autorisée. Cela dit, j'espère avoir un choix large concernant mon prochain stage, et de pouvoir retravailler dans une atmosphère saine l'an prochain.

## 5.4 Le futur de PhaistOS

Concernant PhaistOS, il est évident que ce projet est original et a un potentiel d'exploitation dans le futur. Assez maléable et relativement simple à maintenir, il permettra d'écrire ses propres politiques d'E/S pour son système et offre des possibilités d'optimisation intéressante côté applicatif. Le programme en cours d'utilisation pourra potentiellement changer d'algorithme d'ordonnancement à chaud, pour optimiser son débit et sa latence d'E/S sur le (s) disque(s). Cette possibilité reste donc séduisante même si le DSL reste restrictif concernant son API et ses possibilités de personnalisation, c'est le parti pris pour la simplification offerte par le langage.

# Références

[1] Jonathan Corbet. Two new block i/o schedulers for 4.12. *Lwn.net*, Avril 2017.

[2] Luciano Porto Barreto et Gilles Muller. Bossa : une approche langage à la conception d'ordonnanceurs de processus. In *Journées francophones des jeunes chercheurs en systèmes d'exploitation (ASF'2002)*, Avril 2002.

[3] Lepers Baptiste et Gouicem Redha et Carver Damien et Lozi Jean-Pierre et Palix Nicolas et Aponte Maria-Virginia et Zwaenepoel Willy et Sopena Julien et Lawall Julia et Muller Gilles. Provable multicore schedulers with ipanema : application to work conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[4] Colin Ian King. Ioschedulers. *Ubuntu Wiki*, Septembre 2019.

[5] François Pottier. *Visitors (Manual)*. Inria Paris, Juin 2021.

[6] Carsten Schack-Nielsen, Anders et Schürmann. Celf–a logical framework for deductive and concurrent systems (system description). In *International Joint Conference on Automated Reasoning*, pages 320–326. Springer, 2008.

# 6 Annexes

## Définitions des événements d'une politique

```
1  EVENTS {
2      On INIT() do: {
3          init(POLICY.fifo_list[READ]);
4          init(POLICY.fifo_list[WRITE]);
5          ...
6          POLICY.fifo_expire[READ]=read_expire;
7          POLICY.fifo_expire[WRITE]=write_expire;
8          POLICY.fifo_batch = fifo_batch;
9          POLICY.writes_starved = writes_starved;
10         POLICY.starved = 0;
11         POLICY.batching = 0;
12     }
13     On DISPATCH(request rq) do: {
14         bool reads;
15         bool writes;
16         reads = !is_empty(POLICY.fifo_list[READ]);
17         writes = !is_empty(POLICY.fifo_list[WRITE]);
18         if(rq && POLICY.batching < POLICY.fifo_batch) {
19             POLICY.batching++;
20             return rq;
21         }
22         if(reads) {
23             if (deadline_fifo_request(WRITE) &&
24             POLICY.starved++ >= POLICY.writes_starved) {
25                 return dispatch_writes();
26             }
27             return dispatch_reads();
28         }
29         if(writes) {
30             return dispatch_writes();
31         }
32         return rq;
33     }
34     ...
35 }
```

FIGURE 12 – Exemple d'une définition des événements `INIT` et `DISPATCH` écrits dans le langage du DSL pour la politique Deadline.

# Arborescence du projet

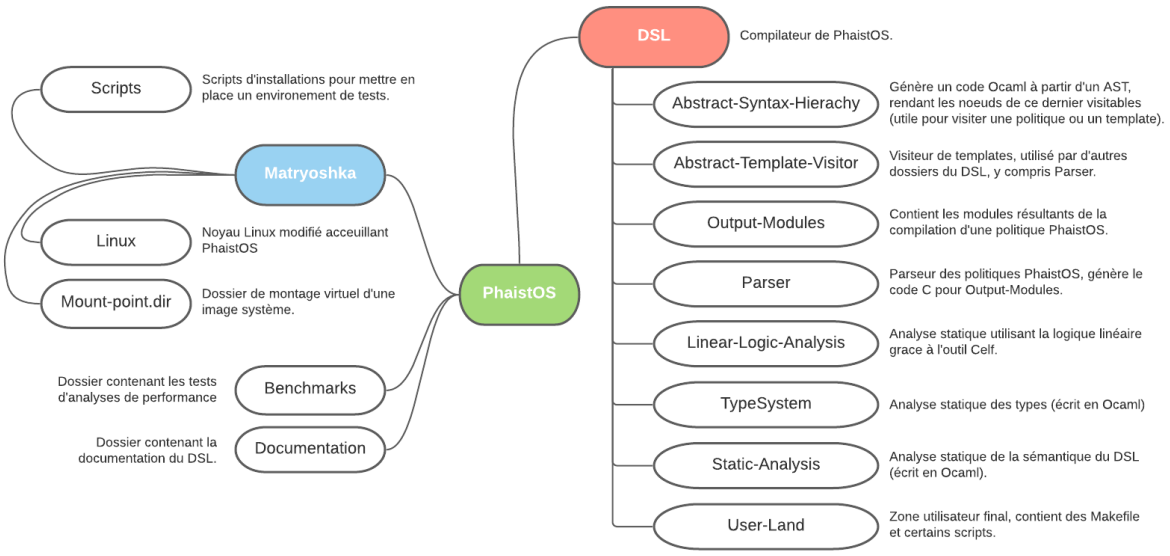


FIGURE 13 – Architecture du projet PhaistOS.

| **Nom** | **Catégorie de file d'attente** | **Description** |
|---|---|---|
| Deadline | Simple | Conçu pour régler les problèmes de famine aperçus dans d'autres ordonnanceurs à l'époque. Son algorithme utilise 3 files, à savoir une file dite "triée" (par priorité), une file de lecture et une file d'écriture. Les requêtes de la file triée sont prioritaires et si une des requêtes des deux autres files expire, elle devient prioritaire. Les lectures ont un délai d'expiration de 0.5s par défaut tandis que les écritures sont moins privilégiées avec 5s d'expiration. |
| CFQ | Simple | Conçu pour se rapprocher des processus. Possède des files distinctes par processus et se base sur la valeur "ionice" pour les priorités. Chaque file se voit attribuer un temps d'équité, pouvant provoquer ainsi des situation de "vide" si le temps attribué n'est pas utilisé. |
| Noop | Simple | Aucun tri n'est effectué avec cet algorithme d'ordonnancement, seulement des fusions de requêtes aux adresses voisines. Il peut être efficace dans de rare cas, mais cela inclu les contrôleurs de stockage avancés, qui trient eux-même leur requêtes |
| BFQ (Budget Fair Queuing) | Multiple | Conçu pour fournir une bonne réponse interactive, en particulier pour les périphériques d'E/S lents. Il n'est cependant pas idéal pour les périphériques dotés de processeurs lents car chaque opération entraine une surchage de travail assez élévée. La notion d'équité ici repose sur la taille des données demandées plutôt que sur le temps. |
| Kyber | Multiple | Créé pour les périphériques multi-files rapides comme les disques SSD ou les cartes `M.2` par exemple. Il reste relativement simple et distribu les requêtes dans deux files : celle des lectures et celle des écritures. Grâce à sa limitation du nombre de requêtes pouvant être traitées à la fois, Kyber offre un temps de service rapide pour les requêtes à haute priorités [1]. Il sera souvent retrouvé comme ordonnanceur pour des serveurs, cependant pour des machines avec un processeur plus lent, on retrouvera BFQ ou MQ-Deadline. |
| None | Multiple | C'est tout simplement l'ordonnanceur qui n'en est pas un. Aucune réorganisation des requêtes n'est effectuée, procurant ainsi une surcharge minimale. Il peut être utile pour certains appareils très rapides utilisant les technologies NVMe, comme les disques SSD. |
| MQ-Deadline | Multiple | C'est la version multi-files de l'algorithme "Deadline", polyvalent, il reste le plus souvent utilisé dû à sa faible surcharge CPU. |

Table 2 – Les différentes politiques d'ordonnancement de Linux

# Script d'installation d'une VM

```
#!/usr/bin/env bash
sudo apt-get install -y flex bison libelf-dev openssl libssl-dev parted qemu-kvm qemu make

CORES=`expr \`lscpu -e=CPU | wc -l\` - 1`
IMG=qemu-image.img
DIR=mount-point.dir
VER=5.11.22

# From scratch
rm -rf $IMG $DIR linux/

#-------------------------------------------------------------------#
#           DOWNLOAD and BUILD a linux kernel 5.13-rc7              #
#-------------------------------------------------------------------#
wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-$VER.tar.xz
tar -xf linux-$VER.tar.xz
rm linux-$VER.tar.xz
mv linux-$VER linux
cd linux
make x86_64_defconfig
make modules_prepare
make kvm_guest.config # previously "make kvmconfig"
make -j $CORES
cd ../
#-------------------------------------------------------------------#

#-------------------------------------------------------------------#
#     Setup image by CREATING it, MOUNTING it, CONFIGURING it.      #
#-------------------------------------------------------------------#
qemu-img create $IMG 3g
losetup -fP $IMG # Setup loop device
MNT=`losetup -a | grep $IMG | tail -n 1 | cut -d ":" -f 1`
# Create a partition :
parted $MNT -s mklabel msdos mkpart primary ext4 1MiB 100%
sleep 0.5 # need to wait here because /dev/loop0p1 or whatever similar can not be seen immediatly ...
mkfs.ext4 $MNT\p1 # p1 by default
mkdir $DIR
sudo mount -t ext4 $MNT\p1 $DIR # Mount loop device in directory
sudo debootstrap --arch amd64 buster $DIR # Download a debian base system (no kernel)
# We bind linux source code to our image because we need it to install modules on the debian system :
sudo mount --bind linux/ $DIR/mnt/

# Modifying the content of the mounted image.
cp /etc/resolv.conf $DIR/etc/resolv.conf # to fix potential dns problems while chroot
chroot $DIR /bin/bash -x << EOF
echo "PhaistOS" > /etc/hostname
echo "root:root" | chpasswd
cd ~
mkdir phaistos/
echo "dhclient -4" >> .bashrc
. .bashrc
apt update
apt install -y build-essential
echo '# Root is partition1
/dev/sda1 / ext4 rw,relatime,data=ordered 0 1
# The kernel source is on the drive of the host, we use the virtfs 9p to see it with qemu.
kernel /mnt 9p trans=virtio,version=9p2000.L 0 1' >> /etc/fstab
cd /mnt/
make modules_install
EOF

# Copy PhaistOS modules onto the mounted image with the Makefile.
rsync -r ../PhaistOS-DSL/Output-Modules/* $DIR/root/phaistos/ --exclude=README.md
#-------------------------------------------------------------------#

#-------------------------------------------------------------------#
#                              FINISH                               #
#-------------------------------------------------------------------#
# Completly unmount the image and ask user to launch qemu and run the install script
sudo umount $DIR/mnt
sudo umount $DIR
sudo losetup -d $MNT

echo -e "\033[32mSetted root password with : \033[5;1mroot\033[0m"
echo -e "\033[32mPlease launch QEMU with : \033[1m./bootQemuLinuxWithCustomKernel\033[0m"
echo -e "\033[32mLogin and then run : \033[1m~/build_modules.sh\033[0m"
#-------------------------------------------------------------------#
```

FIGURE 14 – Script compressé illustrant la compléxité de l'installation d'une VM.

# Diagrammes de Gantt

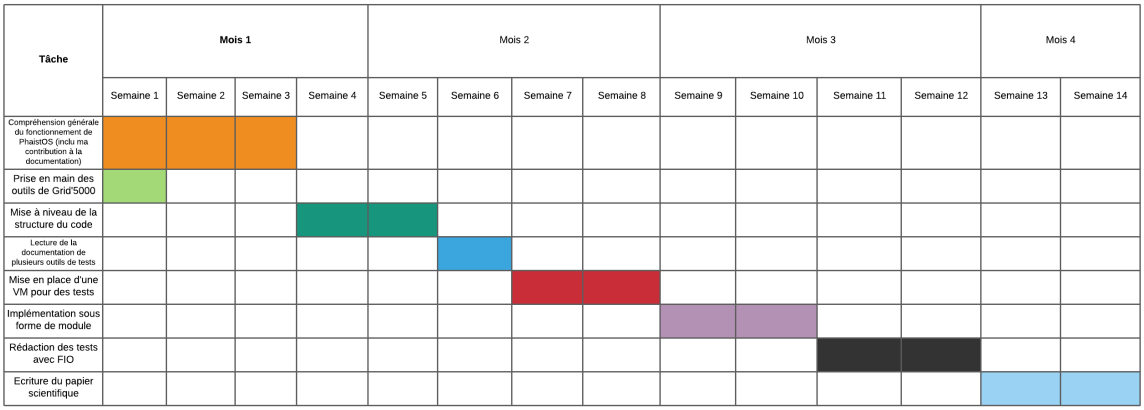


FIGURE 15 – Diagramme de Gantt estimé, profilant un planning fluide

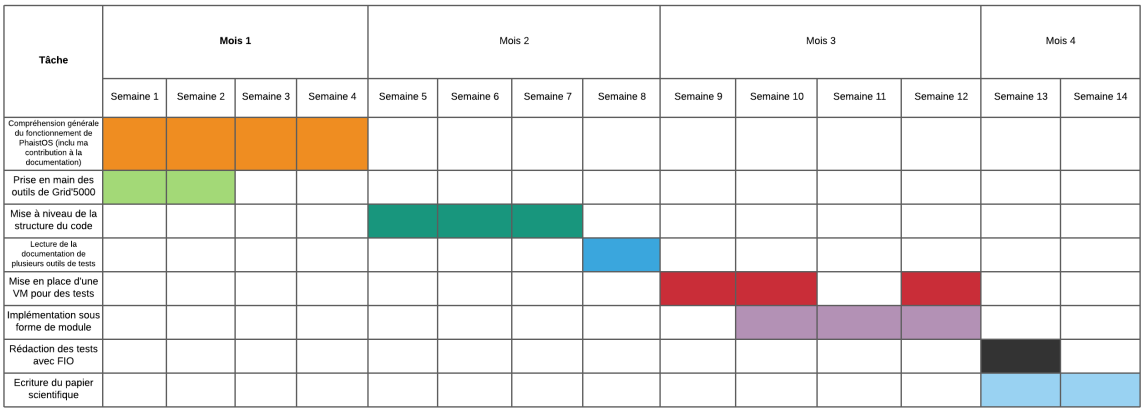


FIGURE 16 – Diagramme de Gantt réel, exposant des tâches qui se superposent

# Dos du rapport

Étudiant : Victor Malod Année d'étude dans la spécialité : INFO 4

Entreprise : Laboratoire Informatique de Grenoble
Adresse : Bâtiment IMAG, 700 Av. Centrale, 38401 Saint-Martin-d'Hères
Téléphone : 04 57 42 14 00

Responsable administratif : Pascale Poulet
Téléphone : 04 57 42 14 01
Courriel : pascale.poulet@imag.fr

Tuteur de stage : Nicolas Palix
Téléphone : 04 57 42 15 38
Courriel : nicolas.palix@imag.fr

Enseignant-référent : Vincent Danjean
Téléphone : 04 57 42 14 76
Courriel : vincent.danjean@imag.fr

Titre : Développement sur le compilateur de PhaistOS

Résumé :

Ce rapport de stage résume mon expérience de travail au sein du Laboratoire d'Informatique de Grenoble durant ma quatrième année d'études universitaires. Afin de valider mon année d'étude à l'école d'ingénieur de Polytech Grenoble, ce rapport sera fourni ainsi qu'une soutenance à un jury responsable de m'évaluer.

Je faisais parti de l'équipe Erods et ma mission consistait à tester et améliorer le compilateur d'un langage de programmation dédié à l'ordonnancement d'entrées/ sorties d'un système linux. L'outil en question s'appelle PhaistOS et a été développé par Nick Papoulias, un chercheur postdoctoral qui ne travail maintenant plus sur le projet.

Durant ce stage différentes tâches m'ont été attribuées par mon tuteur, j'ai commencé par m'approprié le sujet en lisant la documentation et le code qu'avait écrit Nick. J'ai ensuite fait de la documentation puis du développement sur le compilateur. J'ai finis mon stage par des tests de performances et l'écriture d'un papier scientifique portant sur le compilateur.