Projet TuxML :
Post-mortem

Corentin Chédotal Gwendal Didot Dorian Dumanget
Antonin Garret Erwan Le Flem Pierre Le Luron
Alexis Le Masle Mickaël Lebreton Fahim Merzouk

Encadrés par Mathieu Acher

4 Mai 2018

Remerciements

L'équipe du projet TuxML souhaiterait remercier monsieur Olivier Barais pour avoir pris le temps de présenter Docker et son utilisation ainsi que pour les pistes de travail concernant son emploi dans le cadre de notre projet.

Le logo du projet utilise des *emojis* modifiés du projet Noto mis à disposition gracieusement sous la licence Apache. Seul l'emoji `1f473 1f3fb` a été modifié avant son inclusion dans le logo afin d'en extraire et recolorer son turban. Puis l'ensemble du logo a été recoloré pour donner un aspect "plan de construction".
Le logo Linux Tux ainsi que le fond font partie du domaine public.

Table des matières

1 État du projet

1.1 Objectifs

À long terme le but du projet TuxML est de fournir un exécutable permettant à un utilisateur "novice" de faire choisir les options de compilation de son noyau Linux à une machine ayant appris par *Machine Learning* les interactions entre ces 14000 options et le bon fonctionnement du noyau (compilation et viabilité) ainsi que dans certaines fourchettes de métriques bien précises telles que la taille du noyau, sa vitesse de *boot*...

À plus court terme l'intention était d'avoir un système en place permettant de produire les échantillons nécessaires à la réalisation du *Machine Learning* ainsi que la réalisation d'une preuve de concept quant à l'étude des noyaux produit (ici avec le fait qu'ils compilent ou non, voir qu'ils marchent ou non).

1.2 État actuel du projet

A ce stade, le projet en l'état nous permet de lancer des compilations en masse du kernel linux selon des configurations aléatoires. Ces compilations se complètent automatiquement en installant les paquets qu'il manque à son bon fonctionnement. Distribué sur une grille de calcul, TuxML est capable de remplir une base de données avec les données de compilation telles que la taille du noyau généré, son temps de compilation, si il compile ou si il boot et plus encore.

2 Les points positifs

2.1 Docker

Un premier problème assez général était la compatibilité du projet TuxML avec différentes machines. En effet, une compilation avec un certain nombre d'options de configuration du noyau Linux peut donner des résultats différents (Temps de compilation, taille du noyau ...) selon la machine qui effectue cette compilation et son système d'exploitation. Nous en sommes venu à utiliser *Docker* sur conseils de Monsieur Olivier BARAIS afin de pallier à ces soucis.

Docker nous a permis de créer une image Debian personnalisée sur laquelle se trouve les scripts de compilations et pouvant donc être exécutée sur toute machine disposant de Docker. Chaque membre de l'équipe peut donc exécuter le projet depuis sa machine personnelle (cf. `MLfood.py`) car quelque soit le système d'exploitation installé, le projet s'exécutera à l'intérieur de l'image Debian personnalisée de Docker, indépendamment du système hôte.

2.2 Compilation automatique

L'un des premiers problèmes auquel l'équipe a dû faire face concerne la compilation. En effet, l'apprentissage automatique nécessite un échantillon important de compilations. Il était impossible pour nous de lancer manuellement des milliers

de compilations et d'apporter les corrections nécessaires en cas d'échec. Nous nous sommes très vite rendu à l'évidence qu'il fallait automatiser tout le processus.

TuxML fonctionne donc en trois temps : d'abord on lance la compilation, si celle-ci échoue on analyse les logs d'erreurs à la recherche de fichiers ou paquets manquants, on installe ces derniers et on relance la compilation. On effectue ces étapes jusqu'à ce que la compilation ait terminée ou qu'il soit impossible d'installer les paquets.

2.3 Envoi à la base de données

La création d'une base de données a très vite été nécessaire pour sauvegarder les résultats des compilations et pour mener les recherches par apprentissage automatique.

Initialement nous sommes partis sur l'utilisation de JHipster qui permet de générer rapidement un projet au niveau backend et frontend. Cependant nous l'avons assez vite abandonné car les requêtes au niveau de la base de données n'étaient pas assez modulables pour notre utilisation. Nous avons donc mis en place un serveur Linux, Apache, MySQL et PHP et conçu nous même notre base de données.

2.4 Les résultats du Machine Learning

Après la récupération de plusieurs milliers de données, il a été possible de commencer à vraiment s'occuper à la partie Machine Learning, pour voir si l'on pouvait déjà obtenir des résultats intéressants. Bien que le nombre de données soit encore insuffisant pour obtenir des résultats précis, on a déjà pu faire quelques essais sur les données que nous possédions.

Nous voulions, dans un premier temps, réussir à obtenir la taille d'un kernel dont nous connaissions la configuration grâce au `.config` fourni. Nous arrivons, avec un jeu de données de 1300 compilations, à déterminer la taille du kernel avec un écart moyen absolu de 30Mo, ce qui peut sembler beaucoup mais sachant qu'en `Machine Learning`, plus il y a de choix possible, plus il faut de samples, nous sommes assez confiants quant au fait qu'augmenter le jeu de données, augmentera aussi la précision de la prédiction.

Nous espérons qu'à l'avenir il sera possible de déterminer d'autre caractéristiques, telles que le temps de compilation ou les configurations correctes (qui boot). Nous avons même envisagé la possibilité d'un menu de configuration permettant, à l'aide de caractéristiques spécifiques, de pouvoir en ressortir une configuration qui se rapprocherai au mieux de celles-ci.

2.5 L'équipe TuxML

2.5.1 Organisation et collaboration

L'assiduité à toutes les séances et ce durant les quatre heures à chaque fois de toute l'équipe a vraiment créé une grande cohésion et une excellente organisation. Ainsi il a été rapidement possible de se coordonner et de commencer à faire des

binômes plus spécialisés dans des domaines ou sur certaines questions, que nous découvrions au fur et à mesure de l'avancement du projet parfois. Ainsi les forces de chacun permettaient à tous de progresser et au projet d'avancer. Notons par exemple la partie *Machine Learning* qui nécessita un très gros effort de documentation et de recherche bibliographique d'une partie de l'équipe ou encore la gestion des bases de données et des interactions avec celles-ci. La prise de temps à intervalles assez régulier (environ une fois par mois et demi) pour discuter tous ensemble et aviser de ce qui a été réalisé par tous, de l'avancement de la correction des bugs rencontrés, des éventuelles difficultés d'un groupe a été un vrai plus.

2.5.2 Encadrement

La présence de M. Acher à presque toutes les séances et sa constante disponibilité sur *Slack* méritent d'être notées. En effet en plus de permettre de se rassurer quand à l'avancement ou la direction prise parfois par le projet il a pu répondre ou en tout cas guider les recherches sur le domaine du ML que nous ne connaissions pas auparavant.

3 Les points négatifs

3.1 Les dépendances

Notre approche au début du projet était d'installer une liste d'un certain nombre de paquets nécessaires pour la compilation (fichier de dépendances dans TPDIM.py), mais au fil du temps nous nous sommes rendu compte que ces paquets ne sont pas forcément indispensables à chaque compilation. C'est à dire que la liste des dépendances varie en fonction du fichier de configuration *(.config)* du Kernel.

Par conséquent, un binôme a divergé de l'autre partie du groupe pour étudier une autre approche (cf. *tuxml_ depML.py*). Il s'agissait de lancer une campagne de compilation à part et de collecter les données sur le *.config*, sur l'environnement et sur les fichiers manquants à la compilation pour essayer de faire une corrélation avec du Machine learning afin de prédire les dépendances de telle ou telle option activée dans le fichier de configuration du noyau.

L'intérêt du Machine learning des packages à long terme est d'éviter de devoir installer plus de paquets qu'il n'en faut, comme c'est le cas maintenant avec les dépendances installées dans TPDIM.py. Même si on n'en a pas besoin dans quelques cas, par exemple : le paquet *"openssl"* n'est pas toujours indispensable.

3.2 Problèmes spécifiques à l'environnement

L'utilisation de Docker à beaucoup facilité le déploiement sur différent type de machines parfois très hétéroclites. Mais contrairement à ce que l'on aurait pu penser au début, l'utilisation de docker avec une image d'un système unique (Debian 9) ne permet pas forcément une solution immédiatement portable sur n'importe quel système clé en main, et ne masque pas toutes les différences entre les systèmes hôtes. Les différences d'environnement de compilation rendent parfois beaucoup plus compliqué l'écriture d'une solution standard.

Par exemple dans `tuxml_environment.py`, la récupération du type de disque est compliquée : la solution standard marche bien dans un environnement courant (une station de travail) mais sur des machines virtuelles, container Docker avec version différente de Docker, grille de calcul avec un environnement plus spécial, elle est plus hasardeuse. Il a fallu ajouter du code pour prendre en compte ces cas spéciaux. L'utilisation par Docker de partitions virtuelles rend compliqué la récupération de la partition physique (donc du type de disque) où se trouve tuxml. Actuellement cette partie plante systématiquement sur les hôtes utilisant des partitions logiques LVM.

Pour la gestion des dépendances, même avec une image Docker unique, les paquets requis peuvent parfois dépendre de l'hôte qui héberge le conteneur, par exemple pour les pilotes matériels.

3.3 Analyse de logs et paquets manquants

L'analyse de logs a été pendant quelques temps sources de problèmes. En effet, sur la sortie d'erreurs plusieurs formats de messages peuvent indiquer un paquet ou un fichier manquant. Il n'y a pas de format unique et cela nous a posé, au départ, quelques problèmes. Il a fallu plusieurs dizaines de compilations avant d'avoir un ensemble de patterns qui couvraient un maximum d'erreurs indiquant un paquet manquant.

On a aussi été confronté à un autre problème : lorsque les patterns renvoient un fichier manquant, on utilise la commande `apt-file search <nom_du_fichier>` (commande Debian) pour lui associer un paquet. Cependant, certaines fois, cette commande ne renvoie rien et, ne trouvant pas le paquet, on est incapable de reprendre la compilation. Ce problème était récurrent avec les fichiers `aicdb.h` et `as68k.h`. Aujourd'hui nous ne pouvons toujours pas compiler correctement lorsque ces deux fichiers sont nécessaires, fort heureusement il est assez rare de les rencontrer.

3.4 Du problème des faux négatifs de *boot*

Le développement des tests du bon lancement du noyau ont été particulièrement ardus et sont encore source de problème. En effet les différentes techniques découvertes avant celle effectivement mise en place avaient de nombreux problèmes. Ainsi si certains utilisaient une architecture matérielle très lourde d'autres faisaient planter volontairement une machine et détectait son plantage, technique difficilement envisageable sur une grille de calcul. La vraie difficulté apparues assez tôt et couramment dans les techniques vues et toujours présentes dans celle utilisée par la fonction de test est un problème de faux négatifs. En effet il a été découvert par l'expérience que les temps de *boot* de noyaux peuvent être radicalement différent mais surtout que certains cas extrêmes rendent difficile l'établissement d'une fourchette de *timeout*[1]. En effet un noyau se lançant après plus de dix minutes d'attente ayant été découvert par hasard lors de tests. Mais en plus de ce pur aspect temps il y a aussi une grande difficulté à pouvoir garantir qu'un noyau n'a pas fonctionné car il ne fonctionnera

1. *Timeout* rendu nécessaire par la découverte de possibilité de boucle infinie dans le *boot* d'un *kernel*, cf manuel de reprise de code

jamais (vrai négatif alors) contre un noyau qui aurait pu marcher mais qui n'a pas pu dans l'environnement de test [2] (faux négatif). La détection de ces faux négatifs est impossible, il faut alors faire des efforts pour les réduire, par exemple en essayant de déterminer et de créer un bon environnement de test compatible avec le noyau testé. Cependant c'est un problème difficile dont l'étude a seulement commencé.

3.5 Des faux départs

3.5.1 JHipster

Pour pouvoir sauvegarder les données récupérées lors d'une compilation, nous avons décidé de mettre en place une base de données. Avec les conseils de notre professeur encadrant, nous avons décider d'utilisé le système `JHipster`, qui après avoir fonctionné un temps, est tombé en panne sans raisons apparentes. Or nous avions des problèmes avec l'utilisation de la base de données proposée avec Jhipster, nous avons donc décidé de partir sur une solution LAMP classique.

3.5.2 Autotest

Autotest et l'étude de son emploi a été au final une très grande perte de temps. En effet son architecture certes éprouvée mais particulièrement difficile à mettre en place de par sa complexité et visiblement incompatible avec une grille de calcul rendait son emploi impossible. Malheureusement la découverte de ces problèmes ne s'est fait qu'au fur et à mesure de l'étude de sa documentation et de la mise en place de tests à petite échelle. L'un d'entre eux ayant d'ailleurs résulté en un plantage complet d'une machine virtuelle, ce qui était en fait vraisemblablement attendu par Autotest. Avec le recul, il apparaît que, face à un outil qui semblait résoudre une très grande partie des problèmes que nous rencontrions, une forme d'acharnement à le faire marcher se soit mise en place. Ce qui a donc résulté en une perte de temps puisqu'Autotest a été abandonné pour les raisons citées plus haut. Heureusement et grâce à une bonne répartition des tâches le reste du développement fut que très peu impacté durant l'étude et les tentatives de mise en place de cette suite logicielle.

4 Autres aspects

4.1 L'emploi du Python

Le choix du langage de programmation a utiliser fut un des premiers qui avait été fait. Tous réuni nous avions fait une liste des langages de programmation que nous connaissions avec notre niveau d'aisance à développer en ceux-ci ainsi que les points forts et points faibles que nous percevions de ces langages. Le choix fut difficile mais nous nous sommes finalement mis d'accord sur le Python.

Sa première particularité qui nous a très vite attiré était sa modularité. En effet que ce soit en Python 2 ou 3 il existe un très grand nombre de bibliothèques

2. En effet QEMU (émulateur utilisé pour tester le noyau) a ses propres pré-requis au bon fonctionnement du noyau et a besoin d'un système de fichiers donné.

permettant une très grande diversité d'applications. De plus on restait sur des paradigmes de programmation "classiques" et bien connu par l'équipe avec l'itératif et le programmation orientée objet. De plus nous avions remarqué l'utilisation courante de ce langage dans la réalisation de programmes et applications mathématiques. Ce fut le bon choix en effet car, ce que nous ne savions pas au moment du choix du langage, de nombreuses applications de *Machine Learning* simples d'utilisation se programment en R ou en Python.

Cependant le Python ne faisait pas partie des langages connus par la majorité de l'équipe, loin de là. Ainsi ce choix quand bien même très éclairé et fait à l'unanimité est responsable de certaines difficultés dans le développement au début du projet. En effet la rigidité du Python dans certains domaines et sa syntaxe un petit peu particulière fut à l'origine de quelques déboires mais une fois de plus la bonne cohésion de l'équipe a fait que ceux qui avaient un bon niveau déjà aidaient les autres.

4.2 Le travail avec la grille de calcul

Le *Machine Learning* implique d'avoir de très nombreux échantillons [3], pour obtenir ceux-ci il n'était pas envisageable de tout produire à la main. Ce faisant très vite est venu l'idée d'utiliser une grille de machines reliées en réseau. Deux approches différentes mais suffisamment proches pour être étudiées en même temps ont été utilisées.

4.2.1 Les machines du réseau de l'ISTIC

Pour pouvoir obtenir un nombre conséquent de compilations dans notre base de données [4], nous avons décidé d'utiliser les ordinateurs du réseaux de l'ISTIC. Pour ce faire nous avons décidé de développer un script utilisant une liaison ssh pour lancer une suite de compilation grâce a MLfood.py sur les ordinateurs de l'ISTIC. Le script était fonctionnel jusqu'à ce que, bien que ce soit de simple suppositions, il y ait eu un changement dans la façon dont les commandes ssh sans interactions était acceptées par les ordinateurs de l'ISTIC. C'est pour cette raison que le script n'a pas été mit dans le manuel de reprise de code ni dans le manuel d'utilisation.

4.3 La particularité d'un projet de recherche

Ce projet fut pour toute l'équipe une première approche du domaine de la recherche. En effet l'application du *Machine Learning* aux options du noyau Linux est un domaine encore inexploré. Or les approches de recherche avec beaucoup d'étude bibliographique d'exploration de l'état de l'art mais surtout d'expérimentation et d'empirisme étaient assez déroutantes et inhabituelles au début. En particulier l'absence de "réponses" à certaines questions que l'on pouvait se poser était assez différent des approches des autres plus petits projets de l'année. Ici il était parfois normal de ne pas avoir de point de repères en terme de temps d'exécution, de nombres d'échantillons requis ou d'autres données de ce genre. La découverte de celles-ci (par empirisme ou inférence par rapport à d'autres données par exemple) faisant

3. Ici par "très nombreux" on entend plusieurs milliers voire dizaines de milliers minimum.
4. De l'ordre de quelque milliers ici

justement parti de l'exploration inhérente à la recherche. Mais ces challenges étaient bien reçus et étaient sources d'inspiration et de motivation pour tous les éléments de l'équipe et ont permis d'en apprendre beaucoup sur de nombreuses technologies et techniques.