Traduction dynamique hybride matériel/logiciel de binaires x86_64 vers RISC-V

Raphaël Le Puillandre

***Abstract*—Dans ce stage nous étudions la faisabilité et la pertinence de la mise en oeuvre d'un traducteur dynamique hybride matériel/logiciel de binaires x86_64 vers RISC-V. Nous mesurons dans un premier temps les performances des traducteurs actuels entièrement logiciels, puis nous mettons en place un modèle permettant d'estimer les possibles gains en performances obtenus grâce à l'utilisation d'un traducteur matériel.**

I. Introduction

LES processeurs de type x86 sont aujourd'hui largement utilisés pour de multiples applications, que ce soit pour équiper nos ordinateurs personnels ou des serveurs de calcul. Ces processeurs existant depuis des décennies sont rétro compatibles avec tous leurs prédécesseurs, rendant ainsi possible l'utilisation directe d'une riche bibliothèque de logiciels ayant bénéficié de décennies de développement, et de logiciels parfois disponibles uniquement en version x86. Cependant ce jeu d'instructions étant la propriété d'Intel, la production et la vente de processeurs compatibles est très réglementée si bien qu'aujourd'hui, Intel et AMD sont quasiment les seules entreprises au monde à vendre des processeurs x86. Ces entreprises se focalisant sur le marché des ordinateurs personnels, il est parfois difficile pour des constructeurs d'équipement électronique de trouver des processeurs x86 adaptés à leurs besoins.

Pour contrebalancer cette situation, certains jeux d'instructions sont développés de manière libre et ouverte, permettant ainsi aux constructeurs de produire des processeurs moins chers, libérés de tous royalties. C'est le cas du jeu d'instructions RISC-V, créé en 2010 à l'Université de Californie à Berkeley. Défini avec de nombreuses extensions standard, il donne aux constructeurs la liberté d'implémenter les extensions de leur choix afin de produire un processeur parfaitement adapté à leurs besoins.

Cette modularité rend possible l'utilisation de RISC-V pour tous les types de processeurs, de la montre connectée au supercalculateur, et fait de ce standard le candidat idéal pour devenir un jour le jeu d'instructions universel. Bien qu'il soit aujourd'hui en train de s'imposer parmi les processeurs à destination de l'embarqué et des objets connectés, son adoption massive sur le marché des ordinateurs personnels et du calcul haute performance reste encore freinée par la relative pauvreté de son catalogue de logiciels compatibles.

Afin de contourner cette difficulté, des approches de traduction de binaires peuvent être mises en oeuvre avec une efficacité acceptable. La majorité des approches mises en oeuvre aujourd'hui (QEMU [1], Box64, Hybrid-DBT [6], Transmeta CMS [5]) sont axées sur la traduction dynamique des binaires. Cette technique permet notamment aux traducteurs de récolter des informations supplémentaires sur le programme qui s'exécute afin de pouvoir générer un code plus optimisé. La traduction dynamique a également l'avantage de pouvoir gérer des sauts dont la destination n'est connue qu'à l'exécution. Cependant les phases de traduction intervenant durant l'exécution représentent un surcoût important qu'il est nécessaire de maîtriser. Une solution peut être d'utiliser un composant matériel consacré uniquement à la traduction des instructions x86 en instructions RISC-V. Plutôt que de développer un composant matériel capable de prendre en charge la traduction de l'entièreté du jeu d'instructions x86, ce qui serait très coûteux étant donné la richesse du jeu d'instructions, il est plus pertinent de s'intéresser uniquement aux instructions les plus utilisées, et minimiser le temps de traduction sur ce sous-ensemble d'instructions.

Dans ce stage, nous étudions la pertinence de la mise en oeuvre d'un traducteur dynamique basé sur un composant matériel capable de traduire très rapidement un sous-ensemble des instructions, couplé à un traducteur logiciel classique prenant le relais lorsqu'il est nécessaire. Dans un premier temps nous évaluons les performances des traducteurs dynamiques de l'état de l'art, puis nous présentons une méthode permettant d'isoler le sous-ensemble d'instructions le plus pertinent à traiter en matériel. Enfin, nous mettons en place un modèle capable d'estimer les gains en performances obtenus sur un exécutable donné grâce à une traduction hybride matériel/logiciel.

II. Background

Chaque type de processeur exécute son propre jeu d'instructions, avec sa sémantique et son encodage. Pour effectuer de la traduction de binaires, il est donc nécessaire de connaître les détails de l'encodage et de la sémantique du jeu d'instructions source, et du jeu d'instructions cible.

A. Jeu d'instructions (ISA)

L'ensemble des spécifications des instructions que peut exécuter le processeur est regroupé sous le terme ISA (Instruction Set Architecture), ou jeu d'instructions en français. On sépare généralement les jeux d'instructions en deux familles différentes disputant des points de vue opposés : CISC d'un côté, et RISC d'un autre. Un jeu d'instructions CISC (Complex Instruction Set Computer) bénéficie d'une large panoplie d'instructions différentes implémentant des opérations complexes. Les jeux d'instructions RISC (Reduced Instruction Set Computer) en revanche disposent d'un ensemble d'instructions

réduit implémentant la plupart des opérations utiles. Les jeux d'instructions CISC peuvent permettre plus de confort aux développeurs assembleur, mais posent des difficultés aux compilateurs qui ont du mal à tirer parti de l'entièreté du jeu d'instructions. A l'inverse, les jeux d'instructions RISC sont plus adaptés à la compilation et moins à la programmation manuelle.

Afin de simplifier le développement et le partage de logiciels, certains jeux d'instructions sont devenus des standards. C'est notamment le cas des jeux d'instructions x86 (CISC) et ARM (RISC). De la montre connectée au smartphone en passant par les ordinateurs portables, beaucoup de ces appareils sont équipés de processeurs ARM. Cette caractéristique commune les rend tous compatibles entre eux, alors même que les processeurs équipant ces appareils restent bien différents.

Les jeux d'instructions ARM et x86 étant propriétaires, les fabricants de processeurs ne peuvent pas les implémenter librement. Au contraire, l'architecture RISC-V évoquée dans l'introduction est libre : n'importe qui peut commercialiser librement un processeur implémentant ce jeu d'instructions. Cela favorise entre autres la compétition entre fabricants, les innovations et la compatibilité des logiciels.

B. Traduction de binaires

Pour exécuter un programme conçu pour un certain jeu d'instructions A sur un processeur utilisant un autre jeu d'instructions B, il est nécessaire de passer par une phase de traduction, afin de transformer les instructions A en instructions B. Cette phase est conceptuellement très simple à réaliser, il suffit d'associer à chaque instruction A, une ou plusieurs instructions B équivalentes. En revanche plus on cherche à générer du code optimisé, plus la tâche se complexifie. La traduction peut être effectuée complètement en amont de l'exécution, on appelle cela de la traduction statique. Cette approche a pour avantage de supprimer le coût de la traduction durant l'exécution à condition de pouvoir être effectuée complètement. Malheureusement, elle n'est généralement pas suffisante à elle seule. En effet, les programmes binaires contiennent souvent ce que l'on appelle des sauts indirects : des sauts dont la destination n'est connue qu'au moment exact où il faut sauter. Les destinations de sauts indirects correspondant à des adresses dans le programme binaire original, il faut traduire cette adresse en adresse dans le programme traduit, ce qui n'est possible qu'une fois que l'on connaît l'adresse exacte de saut.

La traduction dynamique de binaires ou DBT (Dynamic Binary Translation) consiste au contraire à traduire à la volée le code pendant l'exécution. Lorsque l'exécution d'une portion de code x86 est demandée, on fait appel à un programme intermédiaire, un traducteur dynamique, qui va traduire cette portion de code en code RISC-V, afin de pouvoir l'exécuter nativement à chaque fois que son exécution sera redemandée. La traduction dynamique permet une plus grande flexibilité dans la traduction et résoud les problèmes posés par la traduction statique. Dans les deux cas, le coeur de la traduction est le même : il faut décoder les instructions A, les remplacer par une ou plusieurs instructions B, puis les ré-encoder (exemple Figure 1).

```
Octets x86_64 :
48 c7 c0 06 00 00 00
48 c7 c2 07 00 00 00
48 f7 e2

Désassemblage :
mov $6, %rax
mov $7, %rdx
mul %rdx

Traduction RISC-V (a0 ↔ rax, a2 ↔ rdx) :
addi a0, zero, 6
addi a2, zero, 7
mul a0, a0, a2

Octets RISC-V :
00 60 05 13
00 70 06 13
02 c5 05 33
```

Fig. 1. Exemple de traduction x86_64 vers RISC-V

L'encodage d'une instruction assembleur contient toujours plus ou moins les mêmes informations quel que soit le jeu d'instructions. D'abord il y a l'opcode, un nombre correspondant au type de calcul effectué. Par exemple l'opcode de la multiplication en RISC-V est l'octet 02 et l'octet f7 en x86. Ensuite, il y a l'encodage des registres utilisés, puis du mode d'adressage (souvent complexe dans le cas de x86) et enfin de l'éventuelle constante. Contrairement à l'assembleur RISC-V, les instructions x86 sont de taille variable, les opcodes également peuvent être de 1, 2 ou 3 octets, et les instructions comportent souvent des préfixes modifiant le comportement des opcodes. Ici, toutes les instructions x86 utilisées comportent le préfixe 48, indiquant que les instructions doivent être considérées en mode 64 bits.

La traduction littérale d'une instruction en une autre instruction n'est pas le seul enjeu de la DBT. En effet, en plus de définir un jeu d'instructions, une ISA définit le modèle mémoire que devront respecter les processeurs implémentant ce jeu d'instructions. Or la validité d'une traduction est largement dépendante des modèles mémoire [9] des processeurs source et cible. Pour comprendre les modèles mémoire il est nécessaire de comprendre les mécanismes utilisés par un processeur pour exécuter les instructions, ce que l'on appelle également la micro-architecture du processeur.

C. Micro architecture de processeurs et modèles mémoire

L'ensemble des procédés et techniques utilisés par un processeur pour exécuter les instructions constitue sa micro-architecture. Il existe d'innombrables micro-architectures différentes, chacune ayant ses applications, ses avantages, ses inconvénients. Par ailleurs la micro-architecture d'un processeur est parfaitement indépendante du jeu d'instructions qu'il implémente.

La plupart des jeux d'instructions permettent uniquement de décrire des programmes séquentiels : les instructions sont exécutées les unes après les autres dans un ordre bien précis.

Cependant il existe des jeux d'instructions qui encodent directement le parallélisme des programmes. Les instructions sont alors des mots composés d'un certain nombre d'instructions atomiques qui seront toutes exécutées en même temps. Ces instructions sont appelées des VLIW (Very Long Instruction Word) [8].

Ces jeux d'instructions étant complexes à utiliser efficacement et posant des difficultés de portage sur d'autres jeux d'instructions, ils sont très peu utilisés. À la place, la plupart des processeurs haute performance actuels possèdent une micro-architecture dite superscalaire. Comme les processeurs VLIW, ils peuvent exécuter plusieurs instructions à la fois, mais déterminent eux-mêmes durant l'exécution et grâce à un composant spécial les instructions qu'ils peuvent exécuter en parallèle. Certains de ces processeurs vont même jusqu'à exécuter spéculativement des instructions avant même d'avoir exécuté toutes celles qui les précèdent. On les appelle superscalaires à exécution dans le désordre.

Ne posant pas de problèmes dans le cas d'une utilisation monothread, le parallélisme induit par les processeurs superscalaires peut en revanche causer des différences de comportement en fonction du niveau d'anticipation du processeur. Le niveau auquel les processeurs ont le droit d'anticiper les opérations qu'ils exécutent est appelé le modèle mémoire du processeur. Un modèle mémoire fort comme celui des processeurs x86 limite les prédictions qui peuvent être faites sur les instructions. En revanche, un modèle mémoire faible rend le processeur très libre quant aux prédictions qu'il fait sur les instructions à exécuter.

Comme dit précédemment, il serait intéressant de pouvoir bénéficier instantanément de tous les programmes développés pour processeur x86 sur des processeurs RISC-V. Pour cela on peut mettre en oeuvre un système de traduction de binaires.

En plus des sauts indirects, les traducteurs de binaires doivent faire face au challenge du code auto-modifiant. En effet, lorsqu'un programme modifie son propre code, il faut être capable de le détecter et de retraduire le code modifié. Ce comportement pathologique de certains programmes pourrait sembler marginal. Cependant le code auto-modifiant est au coeur d'une technique d'accélération de l'interprétation bien connue : la compilation à la volée (ou Just In Time, JIT).

D. Compilation à la volée

La compilation à la volée suit le même principe que la traduction dynamique de binaires, mais est effectuée à partir d'une représentation haut niveau d'un programme et consiste à effectuer, durant l'exécution, toutes les étapes de la compilation de cette représentation en code machine exécutable. Elle se distingue donc de la DBT qui se limite à traduire directement un code binaire vers un autre. L'application la plus fréquente de la compilation à la volée se trouve dans les machines virtuelles d'exécution.

Il y a plusieurs variantes de compilation à la volée. Dans la majorité des cas, le programme source est compilé (préalablement ou à la volée) dans une représentation intermédiaire appelée bytecode, qui est ensuite traduit dynamiquement en code machine natif durant l'exécution.

La compilation à la volée est notamment utilisée dans la plupart des implémentations de la Machine Virtuelle Java [7] et dans PyPy, une implémentation alternative de Python.

Il est donc important qu'un traducteur de binaires soit capable de gérer correctement ce type de comportement.

III. Les mécanismes de fonctionnement de la DBT

De nombreuses solutions de DBT existent déjà pour de multiples cas d'applications. Il est important dans un premier temps de comprendre leur fonctionnement, leurs différences, leurs forces et leurs faiblesses. Parmi les nombreux traducteurs dynamiques de binaires existants, nous nous intéressons en particulier à QEMU et Box64. Étant open-source et grâce à sa portabilité, QEMU est à ce jour l'un des logiciels les plus utilisés dans le portage et la conception d'émulateurs. Quant à Box64, il s'agit du traducteur dynamique de binaires x86 vers RISC-V le plus avancé actuellement, tant en terme de fonctionnalités qu'en terme de performances. En plus de ces deux logiciels, nous nous sommes également intéressés à d'autres projets dont l'approche ou l'objectif étaient similaires au sujet de ce stage.

A. Box64

Box64 est un traducteur dynamique de binaires x86_64 vers plusieurs autres architectures cibles incluant ARM64, RISCV64. Il a besoin pour fonctionner d'être exécuté sous Linux et ne peut traduire que des exécutables Linux. C'est un traducteur axé sur la performance dont l'un des principaux buts est de pouvoir faire tourner à vitesse acceptable des jeux vidéos demandants en ressources. Box64 est un traducteur dynamique, c'est-à-dire qu'il traduit le code binaire x86_64 en code natif au fur et à mesure de l'exécution. Une fois traduite, une portion de code n'a plus à être traduite de nouveau et pourra être exécutée nativement la prochaine fois que son exécution sera demandée. La traduction se fait par blocs : lorsqu'il s'agit de traduire une nouvelle portion de code, Box64 découpe un bloc de code, puis le traduit, et l'exécute d'une traite. Ensuite soit il saute à un nouveau bloc déjà traduit et l'exécute à son tour, soit il arrive dans une zone non encore traduite, crée un bloc et recommence le processus. Pour traduire un bloc de code, Box64 décode chaque instruction x86_64 du bloc, et va remplacer chaque instruction par des instructions correspondantes dans le langage machine hôte. L'architecture x86_64 étant particulièrement complexe et riche en instructions, déterminer les instructions équivalentes à chaque instruction est un travail particulièrement long et fastidieux. De plus, une fois ce traducteur de code écrit pour générer du code ARM64 par exemple, il faut reprendre tout ce travail de zéro pour pouvoir générer du code RISCV64.

B. QEMU

QEMU [1] est un émulateur rapide généraliste utilisant un traducteur dynamique binaire portable. Par portable, on entend

que QEMU résoud partiellement le problème de la réécriture du traducteur dynamique. Le premier but de QEMU est de pouvoir exécuter un système d'exploitation non modifié dans un autre système d'exploitation, peu importe l'architecture cible et l'architecture hôte. En plus de l'émulation complète de la machine, QEMU est capable d'exécuter directement des binaires Linux d'une architecture différente de l'architecture hôte. Pour régler le problème de la difficulté du portage des émulateurs tels que Box64, QEMU utilise un générateur automatique de générateur de code appelé dyngen. Les générateurs de code générés par dyngen ne transforment pas directement du code binaire de la machine cible vers la machine hôte, mais passent par une représentation intermédiaire des instructions à traduire appelée les micro-operations. Le code assembleur d'origine est décomposé en une série de micro-opérations qui utilisent des registres intermédiaires de manière à diminuer le nombre d'instructions différentes. Chaque micro-opération est codée dans une petite fonction C, et compilée par GCC, puis le code généré est récupéré par dyngen pour générer le traducteur (exemple Figure 2).

```
Code assembleur x86 :
sub $16, %rax

Exemple de micro-code généré :
movl_T0_rax          ; T0 = rax
addl_T0_im -16       ; T0 = T0 - 16
movl_rax_T0          ; rax = T0

Code C de la micro-opération movl_T0_rax :
void op_movl_T0_rax(void)
{
    T0 = env->regs[0];
}
```

Fig. 2. Exemple de micro-opérations QEMU

Ainsi pour émuler une nouvelle architecture avec QEMU, le seul travail est de traduire à la main chaque instruction en micro-opérations. Et contrairement à Box64, une fois ce travail effectué, l'architecture cible peut être émulée sur n'importe quelle autre architecture.

Cette représentation de l'architecture à émuler par un petit ensemble de micro-opérations (quelques centaines) permet certes de gagner du temps lors de l'implémentation de traducteurs dynamiques, mais résulte en des performances inférieures à des traducteurs écrits spécifiquement d'une architecture vers une autre (comme Box64).

C. Rosetta 2

Rosetta 2 est un traducteur binaire statique x86_64 vers ARM64 développé par la société Apple Inc. dans le but de faciliter la transition vers ses propres processeurs Apple Silicon qui sont sous architecture ARM. Rosetta 2 permet aux utilisateurs d'exécuter des applications compilées pour processeur x86 sur leur Mac ARM. Pour cela, lorsqu'un exécutable x86 est lancé, le système d'exploitation détecte qu'il s'agit de code x86 et lance un processus de traduction. Une fois ce processus terminé, un exécutable natif ARM a été produit, et sera exécuté à la place de l'exécutable x86 à chaque fois que ce sera nécessaire. Cette transformation est possible principalement grâce au fait que le processeuur d'Apple possède une instruction spéciale qui permet de changer de modèle mémoire. En effet l'ISA x86 impose plus de restrictions que l'ISA ARM quant à la réorganisation des instructions par le processeur. Comme dit précédemment, la traduction statique est en pratique quasiment impossible à mettre en oeuvre seule, c'est pourquoi Rosetta 2 incorpore également des outils de traduction dynamique appelés lorsqu'il faut exécuter du code qui n'a pas pu être traduit statiquement.

D. Transmeta Code Morphing Software

Le Code Morphing Software (CMS) de Transmeta [5] associé à son processeur VLIW Crusoe constitue une implémentation complète au niveau système du jeu d'instructions x86. Le code x86 est exécuté en étant traduit dynamiquement en code natif pour le processeur Crusoe avant d'être exécuté nativement sur ce dernier. Le CMS combine un interpréteur, un traducteur dynamique de binaire, un optimisateur et un système d'exécution. Ce produit avait pour but de concurrencer les processeurs x86 natifs en étant beaucoup plus efficaces énergétiquement pour une même vitesse d'exécution.

Pour y arriver, CMS a dû faire face à plusieurs défis : implémenter de manière correcte l'ensemble du jeu d'instructions x86, garantir de bonnes performances quelle que soit l'application, gérer le code auto-modifiant, reproduire fidèlement les exceptions matérielles, tout cela sans aucune garantie quant au système d'exploitation qui s'exécute, donc sans avoir accès à l'exécutable de l'application mais seulement au contenu de la mémoire stockant le code binaire du programme.

Quand le CMS doit exécuter du code pour la première fois, le code est d'abord interprété. Cela permet au logiciel de récolter des informations à propos de l'exécution. Quand le nombre de fois qu'un morceau de code a été exécuté dépasse un certain seuil, le CMS appelle le traducteur qui va traduire le bloc et l'ajouter dans un cache, puis l'exécuter nativement. Si par la suite un bloc de code n'existe pas en version native, l'exécution repasse à l'interpréteur, puis au traducteur etc.

E. HybridDBT

HybridDBT [6] est un système de traduction dynamique de binaires accéléré matériellement visant exécuter des binaires RISC-V sur une machine VLIW dans le but de réduire la consommation d'énergie sans impacter les performances. Le système est composé de deux coeurs VLIW à 4 voies générés par de la synthèse de haut niveau (HLS), d'un processeur de DBT qui gère la traduction et effectue des optimisations de code, et de 3 accélérateurs matériels : IR Scheduler, IR Generator et FirstPass Translator. Le rôle de FirstPass Translator est d'effectuer une première traduction du code, peu optimisée et la moins coûteuse possible. Le IR Generator est lui chargé de générer une représentation de plus haut niveau du code binaire à traduire, que IR Scheduler va être utilisé pour générer le code VLIW traduit.

Le niveau d'optimisation de la traduction est progressif et dépend de chaque bloc de code. Dans un premier temps,

(niveau d'optimisation 0) les instructions sont traduites de manière naïve, sans tirer parti du parallélisme du programme. Quand un bloc possède un nombre suffisant d'instructions, une représentation intermédiaire du bloc est construite afin de générer du code VLIW optimisé au maximum. C'est le niveau d'optimisation 1. Enfin, quand un bloc a été exécuté suffisamment de fois, il va subir les optimisations inter-bloc : un graphe de flot de contrôle est construit à partir des instructions de saut et permet d'extraire encore plus de parallélisme afin d'exploiter au maximum les coeurs VLIW.

IV. Support matériel envisagé

Afin d'implémenter un système d'exécution basé sur de la traduction dynamique de binaires et ciblant l'architecture RISC-V, il existe des puces intégrant un processeur RISC-V couplé à un FPGA. C'est sur ce type de puce, plus exactement une Microchip Polarfire SoC Discovery Kit que nous avons effectué nos premières expérimentations. Une partie significative de ce stage a été passée comprendre comment configurer cette carte, et notamment installer une distribution Linux dessus.

Ensuite, et pour plus de commodité, nous utilisons une VM QEMU avec Ubuntu.

V. Estimation des gains en performances apportés par une accélération matérielle

Les phases de traduction représentant un surcoût certain lors de l'exécution d'un programme, il pourrait être intéressant de traiter une partie de la traduction grâce à un composant matériel dédié. Ce composant serait chargé de la phase de traduction d'un bloc de code x86 en un bloc de code RISC-V. Le jeu d'instructions x86 étant extrêmement dense (plusieurs milliers d'instructions différentes), il serait très coûteux de traiter en matériel le jeu d'instructions x86 entier. Nous allons donc dans un premier temps présenter une méthode permettant de sélectionner les instructions les plus rentables à traiter en matériel. Grâce à ces informations, nous pourrons mettre au point un modèle permettant d'estimer le facteur d'accélération obtenu par le traitement en matériel de ces instructions, par rapport à une traduction purement logicielle, et ainsi tenter de valider la pertinence de cette approche.

Pour cela, nous nous sommes appuyés sur le logiciel Box64 qui est à ce jour l'un des traducteurs binaires x86_64 vers RISCV les plus avancés. Le choix de Box64 est d'autant plus pertinent que ce logiciel est développé dans un objectif de performance, qui est également le but recherché en effectuant la traduction grâce à un composant matériel. Box64 est libre et open-source, ce qui permet de modifier aisément son code source afin de récolter toutes sortes de données d'exécution.

A. Évaluation des performances de la traduction dynamique

1) Setup: Dans un premier temps et afin de prendre conscience du facteur de ralentissement induit par une exécution à travers une couche de DBT, nous avons exécuté la suite de benchmarks Embench-IOT sur différentes plateformes avec différents niveaux de virtualisation.

Dans un premier temps, nous évaluons le score du benchmark sur un ordinateur personnel (plateforme 1), sans aucun niveau de virtualisation. Les benchmarks sont donc compilés en langage machine x86, puis directement exécutés sur l'ordinateur. Puis, nous exécutons les mêmes benchmarks (mais compilés cette fois en langage machine RISC-V) dans une machine virtuelle QEMU (plateforme 2) installée sur l'ordinateur personnel de la plateforme 1. Ce deuxième test est donc réalisé à travers un niveau de virtualisation. Ensuite, nous installons Box64 dans la machine virtuelle QEMU, grâce auquel nous pouvons exécuter les fichiers compilés pour le premier benchmark, mais cette fois-ci à travers deux niveaux de virtualisation. Nous effectuons ce dernier test avec deux versions différentes de Box64 : une version uniquement interprétée (plateforme 4), et une version utilisant de la traduction de binaires (plateforme 3).

TABLE I
Plateformes d'exécution utilisées

(1) Zorin OS 13.3 Core sur *HP Dragonfly 13.5 inch G4 Notebook PC* (32 GB RAM, 13^{e} génération Intel® Core™ i7-1365U × 12)
(2) QEMU version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.26) exécutant Ubuntu 14.2.0-19ubuntu2, simulant une plateforme `riscv-virtio` avec 16 GiB de RAM et 8 cœurs, tournant sur le système **(1)**
(3) Box64 riscv64 v0.3.5 e4edb65c avec Dynarec, exécuté par le système **(2)**, lui-même exécuté par le système **(1)**
(4) Box64 v0.3.5 e4edb65c, exécuté par le système **(2)**, lui-même exécuté par le système **(1)**

2) Résultats des benchmarks: La Table 1 détaille les spécificités de chaque plateforme.

La Figure 3 montre les résultats obtenus par l'exécution des benchmarks sur les 4 plateformes.

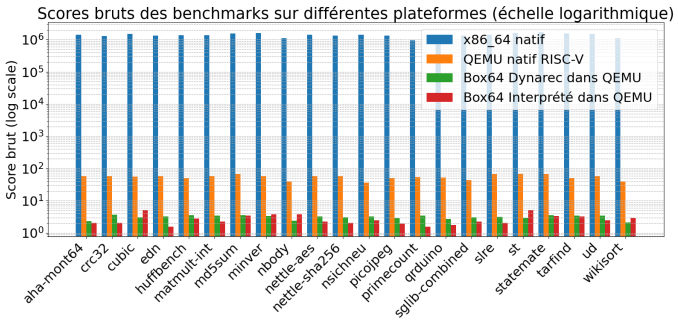

Fig. 3. Scores calculés par le benchmark embench-iot sur 4 plateformes

Le score correspondant à un test est calculé en divisant un temps mesuré sur une plateforme de référence (ARM Cortex M4 cadencé à 16 MHz) par le temps mesuré durant le test. Le score correspond donc au facteur de performance entre cette plateforme de référence et la plateforme à laquelle on s'intéresse.

Ces résultats illustrent parfaitement l'effondrement des performances constaté lors de l'exécution d'un logiciel à travers de la DBT. Ils montrent également qu'en ciblant correctement le responsable de ces performances, il serait alors possible de drastiquement les améliorer.

Les tests de Embench permettent de visualiser la différence de performances entre de l'exécution native et une exécution à travers QEMU ainsi que la différence entre une exécution sur

QEMU et une exécution dans Box64, mais la différence entre Box64 interprété et Box64 dynarec n'est pas représentative de ce que l'on peut observer dans les faits. Sur des programmes s'approchant plus de cas d'utilisation réels (exécution prolongée), Box64 dynarec est beaucoup plus rapide que box64 interprété. L'anomalie que l'on observe est probablement due au fait que les programmes de Embench exécutent peu de fois le même code, et choisir de traduire le code pour l'exécuter seulement quelques fois est moins rentable que de l'interpréter à chaque fois.

C'est pourquoi par la suite et pour juger des différences de performances, nous utiliserons d'autres tests que nous jugeons plus représentatifs d'une utilisation de la DBT.

3) Nouveaux benchmarks: Nous effectuerons nos tests sur 8 programmes différents représentant chaque combinaison des trois caractéristiques suivantes : la taille de l'exécutable, le rapport entre temps de chargement de l'exécutable et temps d'exécution total, et si le programme demande des retraductions de code (typiquement, un programme effectuant de la compilation à la volée).

Pour les exécutables légers n'effectuant pas de retraduction, nous utilisons un programme de multiplication de matrices, dont l'exécutable fait 16 ko. Pour les exécutables lourds, nous utilisons l'implémentation de Python 3 pypy (qui contrairement à l'implémentation officielle CPython effectue de la compilation JIT) et nous faisons varier certains paramètres pour le forcer ou non à recompiler du code. La masse totale des exécutables à traduire (entre exécutable principal et bibliothèques dynamiques) s'élève à environ 62 Mo. Pour l'exécutable léger avec JIT, nous utilisons un interpréteur Brainf*ck utilisant du JIT. La masse d'exécutable à traduire est environ de 20 Mo. Il n'est donc pas particulièrement léger, mais il est compliqué de trouver des logiciels effectuant du JIT à moins de quelques dizaines de Mo.

Comme nous l'avons dit précédemment, au vu de la richesse du jeu d'instructions x86, nous avons décidé de ne prendre en charge la traduction en matériel que d'un sous ensemble des instructions.

B. Choix des instructions à traiter en matériel

Nous avons d'abord tenté d'utiliser la méthode évidente consistant à choisir les opcodes par ordre décroissant de fréquence d'apparition dans le code traduit. Cette méthode s'est avérée peu pertinente, et nous la comparons à une autre méthode que nous avons développée.

Nous cherchons donc le bon sous ensemble d'instructions à implémenter en matériel afin de pouvoir traduire le plus de blocs possibles sans faire appel au logiciel.

Notre méthode consiste à construire cet ensemble itérativement à partir de l'ensemble précédent (contenant un opcode de moins). Elle suppose donc que dans l'ensemble de $n+1$ opcodes, les n premiers opcodes sont les mêmes que dans l'ensemble de taille n. Cette approche est pertinente car elle exploite une propriété naturelle de l'ensemble que nous cherchons.

Il suffit donc de construire une liste ordonnée d'opcodes dont les n premiers éléments constituent l'ensemble de n opcodes les plus pertinents à traiter en matériel.

La méthode est la suivante : On initialise une liste avec l'opcode le plus utilisé de tous les programmes d'Embench. Ensuite, on ajoute itérativement à cette liste tous les opcodes restants jusqu'à ce que la liste les contienne tous, en choisissant à chaque fois celui qui maximise un score mesurant à quel point le traducteur matériel est capable de traduire entièrement de longs blocs.

Le score calculé est le suivant : Considérant un log de toutes les instructions traduites durant l'exécution, nous effectuons la somme pour chacune de ces instructions de la taille du plus grand bloc pouvant être traduit en matériel et commençant à cette instruction, que nous divisons par le nombre d'instructions total exécutées. Si l'on considère que chaque instruction représente le début d'un bloc, ce calcul correspond à la moyenne de la longueur des plus longs blocs possibles constructibles à partir de chaque instruction.

Pour mettre en place une telle méthode, nous récupérons une liste de tous les opcodes exécutés par Box64, ainsi qu'une liste des blocs traduits durant toute l'exécution.

La Figure 4 compare les scores obtenus par le tri des opcodes que nous venons de réaliser et le tri dans l'ordre décroissant de fréquence.

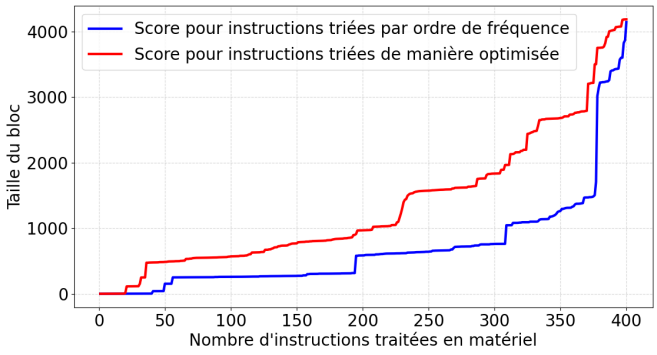

Fig. 4. Nombre moyen d'instructions consécutives pouvant être traduites en matériel

C. Modèle d'estimation de gains en performances grâce à la traduction matérielle

Grâce à cela nous avons ensuite pu construire un modèle d'estimation des gains possibles en performances. Ce modèle donne une estimation du facteur d'accélération d'un programme grâce à la traduction matérielle en fonction de deux paramètres. Le premier paramètre est le nombre d'instructions traitées en matériel et le second est le ratio de performances de la traduction matérielle par rapport à logicielle.

Grâce au classement des instructions que nous avons réalisé et une liste complète des blocs traduits par Box64 pendant l'exécution, on calcule la probabilité qu'un bloc puisse être traduit entièrement en matériel.

Pour pouvoir estimer de manière précise les gains en performances, nous avons également effectué une modification dans Box64 permettant de chronométrer le nombre de TICKS passés à traduire des blocs, exécuter des blocs, et autres. Cela nous donne les proportions de temps passé à chaque étape. Grâce à la distribution de temps obtenue, on calcule une

nouvelle distribution hypothétique en remplaçant le temps de traduction par un nouveau temps de traduction basé sur le fait qu'une certaine proportion de blocs sont traduits en matériel (plus rapide qu'en logiciel d'un certain facteur).

On calcule ensuite le ratio entre le temps total en traduction uniquement logicielle et avec traduction matérielle. Nous avons représenté sur la Figure 5 les résultats pour les 8 programmes de tests, en mettant côte à côte les tests effectués sur le même programme dont on a juste fait varier le temps d'exécution.

Pour plus de concision lors de la lecture, la signification des axes n'a pas été reportée sur chaque graphique. L'axe gradué de 0 à 70 (horizontal, à gauche) correspond au facteur d'accélération entre traduction logicielle et traduction matérielle. Le second axe horizontal (à droite) correspond au nombre d'opcodes dont la traduction est gérée par le matériel. Enfin l'axe vertical correspond au facteur d'accélération estimé entre une traduction entièrement logicielle et une traduction hybride logiciel/matériel.

En analysant ces graphiques, nous pouvons tirer trois conclusions :

1) Si programme a un potentiel d'accélération important, c'est que son exécutable est lourd (temps de traduction du même ordre de grandeur que le temps d'exécution total): Pour les programmes dont l'exécutable est significativement plus lourd que les autres et dont le temps d'exécution est faible (brainf*ck mandelbrot et pypy3 court), on peut espérer des facteurs d'accélération allant de 2 à 3. En effet dans ces cas-là le temps de traduction de l'exécutable occupe une partie significative du temps d'exécution total.

À l'inverse, pour les programmes dont le temps de traduction initial de l'exécutable est négligeable devant le temps d'exécution total (produit matriciel, brainf*ck bench.bf et pypy3 long sans JIT), les gains espérés sont très faibles (1.2 au maximum).

2) La différence d'espérance de gains entre JIT et pas de JIT est présente, mais pas significative: On peut observer entre les deux tests pypy3 longs, que le test avec JIT laisse espérer un facteur d'accélération plus important que le test sans JIT. En effet, le test avec JIT tire directement parti de l'accélération matérielle.

En revanche on ne peut pas faire le même constat sur les tests pypy3 courts. On observe même le phénomène inverse. Cela s'explique par le fait que ces tests sont courts, si bien que le temps de traduction de l'exécutable n'est pas négligeable devant le temps d'exécution total. Il se trouve que le test pypy3 court sans JIT est plus court que le test pypy3 court avec JIT. Ainsi, une proportion plus importante du temps est passée à traduire l'exécutable, et cette différence de proportions masque le fait que le programme effectue du JIT.

3) Plus le temps d'exécution du programme est important, moins la traduction en matériel a d'intérêt: On peut observer sur tous les tests, que pour deux tests mettant en jeu le même programme, mais avec des temps d'exécution différents, le gain en performances attendu est toujours plus important sur le test le plus court (1 contre 1.05 pour matmul, 1.18 contre 3 pour le programme python ne forçant pas de retraduction, 1.06 contre 2 pour le programme python forçant la retraduction et

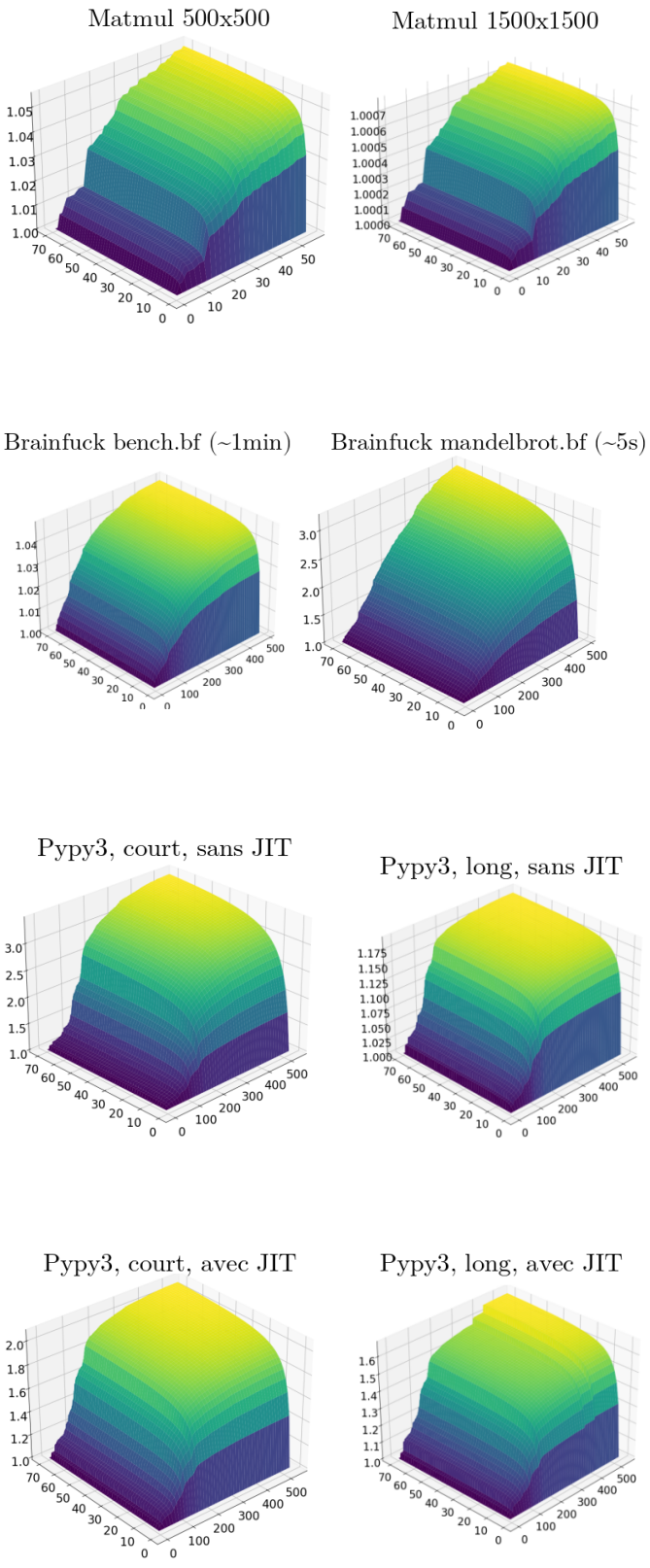

Fig. 5. Espérance de facteur de gain en temps d'exécution total pour la traduction hybride sur les 8 programmes de test.

1.05 contre 3 pour l'interpréteur brainf*ck). Ce constat illustre les limites de l'accélération de l'exécution par le traitement en matériel de la traduction. En effet, si le gain peut être significatif sur des programmes avec un exécutable de taille importante et dont le temps d'exécution est relativement court, plus l'exécution dure longtemps, même lorsque le programme effectue de la retraduction, plus l'intérêt de la traduction en matériel s'estompe car alors la majorité du temps est passée à exécuter le code machine déjà traduit. Le graphique en Figure 6 montre l'évolution du gain maximal en temps d'exécution que l'on peut attendre pour un programme python exécuté avec pypy3 et forçant de la retraduction en fonction de la durée d'exécution du programme.

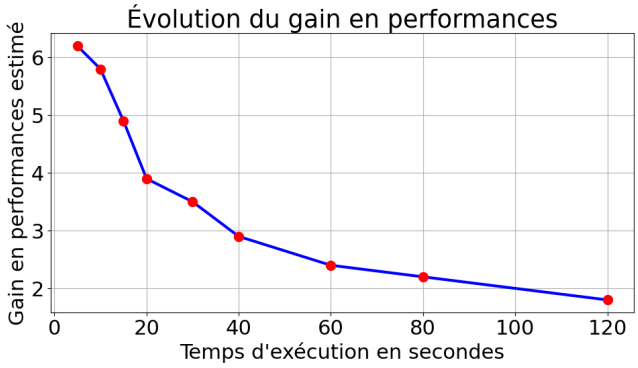

Fig. 6. Évolution du gain en performances estimé sur un programme python (pypy) utilisant de la retraduction, en fonction du temps d'exécution total.

La valeur numérique correspond à la valeur maximale que l'on peut voir sur les graphiques précédents (probabilité 1 de traduire un bloc en matériel et facteur d'accélération matériel/logiciel égal à 70).

D. L'importance du choix des opcodes à traiter en matériel

Pour tous les graphiques montrés dans la Figure 5, les opcodes étaient choisis différemment pour chaque programme, et de manière à maximiser le gain en performances obtenu. Dans une mise en oeuvre concrète, il faudrait déterminer un seul choix d'opcodes, et implémenter le traducteur matériel en fonction de ce choix.

Or, si ces résultats peuvent paraître encourageants, il est important de noter qu'ils dépendent fondamentalement du choix des opcodes qui a été réalisé, et qu'un mauvais choix entraîne un effondrement des gains en performances.

Le graphique en Figure 7 montre les estimations de gain en performances pour le même programme, mais avec un tri d'opcodes différent pour les deux programmes. Pour le Programme A, les opcodes ont été triés directement sur ses propres données d'exécution, afin de maximiser le nombre de blocs traduits en matériel. Ce premier graphique correspond donc au gain réel maximal que l'on pourrait espérer pour ce programme. Pour le programme B, les opcodes ont été triés sur l'ensemble des données d'exécution des programmes d'Embench. On remarque que les gains en performances estimés diffèrent d'un facteur au moins 2 entre ces deux

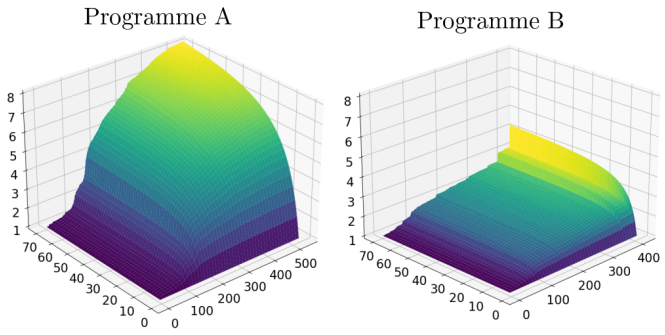

Fig. 7. Différences entre les gains estimés en performances observés avec deux tris d'opcodes différents.

graphiques. Il est donc impératif pour implémenter un traducteur matériel rapide, de trouver un tri d'opcodes universel permettant à une majorité des programmes d'obtenir le meilleur gain en performances possible.

Cependant l'exemple de la Figure 7 illustre la difficulté de ce problème. En effet, alors que les benchmarks d'Embench sont réalisés dans le but de tester un ensemble de cas représentatif des utilisations réelles, et que ces programmes pourraient être des candidats sérieux pour calculer un tri d'opcodes généraliste, il existe des programmes (probablement même une majorité) dont les résultats obtenus sont bien en dessous de leur potentiel réel d'accélération.

Ainsi, si on prend en compte ce problème, il faut reconsidérer à la baisse les performances estimées précédemment.

Pour résoudre ce problème, il faudrait identifier le meilleur ensemble de programmes jugé suffisamment représentatif de toutes les utilisations qu'on veut avoir d'un traducteur de binaires, et calculer les meilleurs opcodes sur cet ensemble de programmes. Il est alors possible que ce tri d'opcodes soit trop généraliste et donne de trop mauvais résultats individuellement sur chaque programme, ou bien que les résultats obtenus se révèlent proches de ce qu'on peut espérer au mieux.

Dans tous les cas, les estimations réalisées ici restent des indicateurs des gains maximaux que l'on peut espérer, mais ne reflètent pas et sont probablement loin de ce que l'on peut obtenir avec un tri d'opcodes général.

VI. Conclusion

L'idée originale de ce stage était d'accélérer l'exécution de programmes en DBT en accélérant la traduction grâce à un composant matériel. L'utilisation de ce composant matériel a soulevé des problèmes comme la nécessité de faire un choix sur les opcodes dont la traduction doit être traitée en matériel. De plus, outre ces considérations sur le choix des opcodes, il s'est avéré que le gain réel d'une accélération de la traduction pour des programmes à une exécution prolongée devenait trop faible pour être intéressant.

Finalement, l'implémentation d'un traducteur hybride logiciel/matériel x86 vers RISC-V ne laisse pas espérer des performances suffisantes pour être intéressante.

ANNEXE

En plus de ce rapport, un fichier MarkDown a été rédigé dans lequel on détaille davantage certains aspects du travail effectué durant ce stage et dont on ne parle pas dans ce rapport.

Ce fichier MarkDown est déposé dans un repository Github en même temps que certains des outils développés pendant ce stage et ayant notamment permis d'effectuer toutes les mesures qui ont été réalisées.

Repo Github : github.com/Bobb56/stage-traduction-binaire

Fork de Box64 permettant de récupérer les données à analyser par les outils : github.com/Bobb56/box64

REFERENCES

[1] Fabrice Bellard. *QEMU, a fast and portable dynamic translator*. USENIX Annual Technical Conference, FREENIX Track, Vol. 41, No. 46, 2005.

[2] K. Ebcioglu et al. *An eight-issue tree-VLIW processor for dynamic binary translation*. Proc. of the 1998 International Conference on Computer Design (ICCD '98), VLSI in Computers and Processors, pp. 488–495, Austin, TX, October 1998. IEEE Computer Society.

[3] S. Banerjia, W.A. Havanki, and T.M. Conte, "Treegion schedul- ing for highly parallel processors", Proc. of the 3rd Interna- tional Euro-Par Conference, pp. 1074-1078, Passau, Germany, August 1997

[4] EBCIOGLU, Kemal, ALTMAN, Erik, GSCHWIND, Michael, et al. Dynamic binary translation and optimization. IEEE Transactions on computers, 2001, vol. 50, no 6, p. 529-548

[5] Dehnert, James C., et al. "The Transmeta Code Morphing/spl trade/Software: using speculation, recovery, and adaptive retranslation to address real-life challenges." International Symposium on Code Generation and Optimization, 2003. CGO 2003.. IEEE, 2003

[6] Rokicki, Simon, Erven Rohou, and Steven Derrien. "Hybrid-DBT: Hardware/software dynamic binary translation targeting VLIW." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38.10 (2018): 1872-1885

[7] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson and M. Wolczko, "Compiling Java just in time," in IEEE Micro, vol. 17, no. 3, pp. 36-43, May-June 1997, doi: 10.1109/40.591653.

[8] Turley, Jim, and Harri Hakkarainen. "TI's new'C6x DSP screams at 1,600 MIPS." Memory 16 (1997): 32

[9] Gouicem, Redha, et al. "Risotto: A dynamic binary translator for weak memory model architectures." Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. 2022.