Techniques d'émulation pour une architecture virtuelle

Felix-Antoine Ouellet
Départment d'informatique
Université de Sherbrooke
felix-antoine.ouellet@usherbrooke.ca

Abstract

1 Introduction

Créer un émulateur est une tâche considérée, à juste raison, comme étant très difficile. Déjà, au niveau technique, il faut que la personne souhaitant programmer un émulateur soit très à l'aise avec le langage de programmation choisi pour mettre en place ce système. De surcroît, des notions de programmation en langage d'assemblage sont requises pour, au minimum, comprendre l'architecture à émuler et, au mieux, optimiser l'émulateur développé. Au niveau de la théorie, de solides connaissances en structures de données et en systèmes d'exploitation seront plus que nécessaires pour mener à terme un tel projet. Mais, plus que tout, l'accès à une bonne documentation du système à émuler et des techniques d'émulation sera essentiel dans un tel projet. Malheureusement, les ressources en ligne sur les techniques d'émulation sont parcimonieuses et il se peut donc qu'une personne ait bien de la misère à débuter un projet d'émulation pour cette raison.

Le présent article a donc pour but de faire connaître les techniques utilisées dans le développement d'un émulateur pour une architecture virtuelle. La section 2 s'attardera justement à décrire l'architecture virtuelle ciblée. La section 3 mettra en relief diverses techniques venant du milieu des interpréteurs pouvant servir à mettre en place un émulateur. La section 4 décrira les principaux problèmes liés à la mise en place d'un recompilateur dynamique et les approches classiques utilisées pour les résoudre. Finalement, la section 5 résumera le contenu de l'article pour quiconque serait tenté de programmer un émulateur et ouvrira des portes vers d'autres techniques n'ayant pas été abordées dans le cadre du présent article.

2 Chip16

2.1 Historique

Avant de pouvoir décrire l'architecture virtuelle Chip16, il faut parler de son prédécesseur: Chip8[REF]. Chip8 est une spécification décrivant une architecture virtuelle et le jeu d'instructions de cette architecture. Le but recherché par cette spécification était d'offrir une plateforme simple à programmer et qui permettrait de développer rapidement des jeux vidéo. La popularité de la plateforme à fait en sorte que plusieurs dérivés, se voulant tous des évolutions, furent produits tel que Chip-48 et Super Chip-48.

La simplicité de cette architecture lui permit donc d'être la suggestion par défaut à ceux désirant s'initier à la programmation d'émulateurs. Cependant, au fil du temps, la documentation originale de cette architecture se révéla comme étant incomplète. Ainsi, certains de ces comportements les plus particuliers étaient passés sous silence dans la spécification originale. La communauté de l'émulation a donc décidé de créer elle-même une architecture virtuelle plus prévisible et mieux

documentée, Chip16[REF]. Encore une fois, le but recherché était d'offrir une entrée en douceur dans le monde de l'émulation.

2.2 Caractéristiques

Chip16 possède un jeu d'instructions réduit comptant 73 instructions. Chaque instruction requiert un espace mémoire de 4 octets et nécessite précisément 1 cycle pour s'exécuter. Le processeur de cette architecture est cadencé à 1 MHz. Au niveau de la mémoire, Chip16 dispose de 64 KB divisé de la façon suivante. Les adresses de 0 à 0xFDEF sont disponibles pour mettre en mémoire une ROM. La pile débute à l'adresse 0xFDF0 et occupe 512 octets. Finalement, les deux ports d'entrées-sorties occupent l'espace allant de l'adresse 0xFFF0 à l'adresse 0xFFFF. En effet, les entrées-sorties dans le modèle architectural de Chip16 produisent une écriture à des adresses mémoires définies à même les spécifications de la machine virtuelle. Détail important à noter, la mémoire est ordonnancée de manière *little-endian*. De plus, Chip16 possède 16 registres à usage général de 16 bits. À cela vient s'ajouter un registre de 16 bits contenant de compteur d'exécution, un autre registre de 16 bits contenant le pointeur de pile et un registre de 8 bits servant de registre de drapeaux.

Au niveau graphique, Chip16 affiche à une résolution de 320x240 en utilisant des couleurs indexées par 4 bits. Le rythme de rafraîchissement est de 60 images par secondes et un drapeau *VBlank* est levé à chaque affichage. Le processeur peut, au besoin, se mettre en attente sur ce drapeau. En plus d'une couche primaire, Chip16 possède aussi une couche d'arrière-plan qui ne peut prendre qu'une seule couleur à la fois.

Une notion importante au niveau graphique est celle du *sprite*. Dans la spécification de Chip16, un *sprite* correspond à un bloc horizontal de 2 pixels, c'est-à-dire, un bloc deux pixels adjacents de façon horizontale. Une conséquence de ce choix est que si, par exemple, la hauteur et la largeur d'un *sprite* sont de deux, concrètement, le *sprite* aura une largeur de 4 pixels et une hauteur de 2 pixels.

La seconde notion primordiale au niveau graphique est celle de la palette de couleurs. Dans l,architecture Chip16, une palette de couleurs est définie comme une collection de 16 couleurs, toutes définies dans le format RGB et occupant donc chacune 3 octets en mémoire. Une palette de couleurs par défaut est définie, mais un usager peut décider d'importer sa propre palette par le biais d'instructions spécifiques à ce cas d'utilisation.

3 Interpréteur

3.1 Définition et motivation

Un interpréteur est un programme qui exécute directement des instructions écrites dans un langage de programmation sans les compiler au préalable. Au sens strict, il n'investira pas d'efforts dans l'optimisation de l'exécutable qu'il exécute que ce soit au niveau des instructions elles-mêmes ou au niveau de la manière qu'il exécute ces instructions. Bien entendu, un bon nombre d'interpréteurs[REF] vont au-delà de cette définition pour offrir une meilleure expérience d'utilisation. Les sous-sections 3.2, 3.3 et 3.4 font d'ailleurs état des techniques et des architectures utilisés dans le milieu des interpréteurs conçus à des fins d'émulation de systèmes.

Un aspect important qu'il faut souligné lorsqu'on décrit un interpréteur est la forme que vont prendre les instructions à exécuter. L'approche usuelle est d'utiliser une représentation linéaire. Parmi les exemples de cette approche, on compte le *bytecode* utilisé par la JVM et les instructions en langage d'assemblage que l'on peut extraire d'un exécutable de jeu pour vieilles consoles de jeux. Une autre approche qui peut être envisagée est l'utilisation d'un arbre syntaxique abstrait (ASA) comme représentation. La différence entre un interpréteur et un compilateur avec une telle représentation est que l'interpréteur ne n'optimisera pas l'ASA. Il supposera plutôt que l'ASA est optimal. Dans une optique plus concrète, l'interprétation avec ce type de représentation s'apparentera au passage d'un visiteur dans un arbre.

Une des forces des interpréteurs pour l'émulation est qu'ils imitent très bien le comportement d'un processeur. Ainsi, la façon dont fonctionne un processeur est qu'il va chercher l'instruction pointée par le compteur d'exécution et il l'exécute. Ce simple algorithme peut facilement être écrit au sein d'un émulateur dans un langage de programmation haut niveau. Ceci permet donc un développement

rapide où le programmeur sera en mesure d'obtenir des rétroactions dans des courts délais. Aussi, un tel système sera facilement testable. En effet, il suffira, tout du moins au début, de vérifier que chaque émulation d'instruction résulte en le comportement attendu par les spécifications de l'architecture émulée.

Au niveau des désavantages, celui qui nuit le plus aux interpréteurs est sa lenteur. De fait, un interpréteur va prendre beaucoup plus de cycle que la machine émulée pour exécuter une instruction étant donné tout le traitement qu'il doit effectuer avant l'exécution. En des termes plus concrets, cette situation peut se traduire par le fait qu'une instruction nécessitant un cycle pour s'exécuter nativement sur l'architecture émulée en prendra dix en s'exécutant au travers de l'émulateur.

Les sous-sections suivantes présentent une approche naïve (3.2) et ce qui est considéré comme étant à la fine pointe de la technologie en ce qui concerne l'implémentation d'interpréteurs.

3.2 Switch Dispatch

L'architecture la plus intuitive et la plus facile à mettre en place pour un interpréteur est celle dénommée *switch dispatch*. Comme son nom l'indique, il s'agit tout simplement de modéliser l'exécution d'une instruction par une conditionnelle à multiples branches. Évidemment, on ne veut pas utiliser cette architecture pour un jeu d'instructions trop grand. Par contre, pour les jeux d'instructions restreints, cette technique permet d'obtenir des résultats très rapidement. Qui plus est, tout du moins en C++, un compilateur faisant face à une telle conditionnelle, souvent massive, pourra l'optimiser en la transformant en table de sauts.

3.3 Threaded Code

Une approche moins intuitive, mais plus rapide, à la mise en place d'un interpréteur est d'utiliser ce qu'on nomme du *threaded code*. Ce qui est désigné par *threaded code* est, en gros, un code de bas niveau consistant essentiellement en une suite d'appels de sous-routines.

Pour y arriver, tout du moins en C++, il faudra utiliser une extension non standard du langage, les *labels as values*, présente uniquement en GNU C++. Cette extension permet d'utiliser des variables avec une instruction *goto* pour effectuer des sauts. L'interprétation aura donc la forme suivante dans ce contexte:

```
void interpret()
{
    goto *pc;
ADD:
    /*z=x+y*/
    goto *(++pc);
//  ...
UNKNOWN:
    /*PANIC*/
}
```

La première chose que l'on peut remarquer dans ce code est que le décodage des instructions ne s'effectue pas de façon centralisée. Bien que cela rend le code moins esthétique, il reste tout de même qu'on gagne en vitesse en agissant de la sorte. Ainsi, moins de travail sera effectué lors de l'émulation d'une instruction de cette manière. De fait, une conditionnelle à branches multiples va toujours devoir faire un test pour s'assurer si elle doit prendre la branche par défaut ou non. On comprend donc qu'en enlevant un test l'interpréteur va s'en trouver accélérer.

La seconde raison pour laquelle cette approche est plus rapide est liée à la manière dont les prédicteurs de branchements fonctionnent. Il va sans dire que prédire le bon branchement que va prendre un programme à l'avance est crucial pour atteindre des bonnes performances en termes de vitesse d'exécution, surtout quand une mauvaise prédiction entraîne de terribles conséquences sur le pipeline du processeur. Dans cette perspective, il serait donc avantageux de fournir le maximum d'information au processeur pour qu'il effectue les meilleures prédictions possible. C'est ce que fait l'approche par *threaded code*. Effectivement, la décentralisation du décodage fait en sorte que la prédiction se fait dans un contexte précis, soit celui de l'instruction courante. À l'inverse, un

décodage centralisé signifie que chaque prédiction est faite de manière générale et il donc plus facile de faire une mauvaise prédiction dans ce cas.

4 Recompilateur dynamique

4.1 Définition et motivation

La meilleure définition de ce qu'est un recompilateur dynamique passe par la description du travail qu'il accomplit. Tout d'abord, il lit une partie d'un fichier exécutable. Par partie, on peut signifier une instruction du langage assembleur dans lequel l'exécutable a été écrit[REF], un bloc de base[REF] ou bien une trace[REF]. Par la suite, la partie de l'exécutable lue est analysée pour y extraire de l'information sur l'activité des registres virtuels. Il est aussi possible d'extraire d'autres informations pour aider à optimiser le nouveau code assembleur à produire, mais ceci dépasse le cadre du présent article. Finalement, à l'aide des informations d'activités recueillies à la phase précédente, le recompilateur dynamique émet des instructions dans le langage assembleur compris par la machine hôte. Généralement, un recompilateur dynamique est couplé à un interpréteur qui aura pour but d'exécuter le code assembleur natif généré.

L'aspect dynamique du recompilateur est nécessaire dans le contexte de l'émulation. Ainsi, le code produit par un recompilateur statique risque fortement de ne pas être correct. La raison derrière cette affirmation est qu'il est impossible, tout du moins avec l'état des connaissances au moment de la rédaction de cet article, pour un recompilateur de déduire certains comportements dynamiques du code à recompiler. De fait, le code original d'un exécutable peut contenir des branchements indirects dont la cible ne peut être déduite de façon statique. De plus, compte tenu qu'il n'existe pas de distinction entre le code et les données au niveau auquel opère un recompilateur, il n'est pas exclu que le code de l'exécutable que l'on désire recompiler contienne du code automodifiant.

La raison majeure pour laquelle on peut désirer mettre en place un recompilateur dynamique est qu'il permet des gains de vitesse énorme au niveau de la vitesse d'exécution d'un émulateur. Ce désir se manifestera surtout lorsque l'on tente d'émuler des consoles de jeux vidéo appartenant à la 6e (GameCube, PlayStation2, XBox) et à la 7e (Wii, PlayStation3, XBox 360) génération. En effet, ces systèmes ont des architectures assez particulières leur permettant des performances remarquables au niveau de la vitesse d'exécution. La seule manière de s'approcher de ces performances est d'utiliser un recompilateur dynamique au sein d'un émulateur.

Les raisons pour lesquelles on voudrait éviter de programmer un tel système sont par contre nombreuses. La première est que programmer un recompilateur est une tâche très ardue. Il s'agit, grosso modo, de mettre en place un "backend" de compilateur. On rappelle au passage que les trois grandes étapes mises en place par cette partie d'un compilateur, la sélection d'instructions, l'ordonnancement d'instructions et l'allocation de registres, sont toutes des problèmes NP-complet. Qui plus est, ces trois problèmes interagissent ensemble ce qui complique encore plus la chose.

La seconde raison pouvant rendre un recompilateur dynamique peu attrayant est la difficulté de tester un tel système. De fait, l'analyse des binaires produits de façon dynamique pour déterminer de leur qualité requiert une expertise dans le langage assembleur ciblé. De plus, suivre le fil de l'exécution d'un recompilateur est difficile compte tenu des sauts qu'il effectue entre l'interprétation et l'exécution de code assembleur. Et bien que certains environnements de développement permettent de suivre un tel fil d'exécution, le problème de comprendre et réglé des problèmes au niveau de l'assembleur reste entier.

4.2 Émission d'instructions x86

Le but principal d'un recompilateur dynamiqueest de produire du code assembleur pour l'architecture hôte. Étant donné que le jeu d'instructions le plus répandu au moment de la rédaction de cet article est le jeu x86 d'Intel[REF], cette sous-section s'attardera à la génération d'instructions pour ce jeu en particulier.

Au niveau du recompilateur, plus particulièrement au niveau de l'émetteur d'instructions x86, le travail est des plus simples. En bref, le recompilateur ne doit qu'émettre des instructions selon une mise en correspondance décidée au moment de l'implémentation. Il doit aussi s'attarder au

mode d'adressage utilisé dans l'instruction courante de la machine cible pour le reproduire dans l'instruction hôte. De plus, bien que le choix des registres ne relève pas de l'émetteur, ce dernier peut choisir des instructions hôtes ayant été optimisées lorsqu'elles sont mise en oeuvre avec des registres hôtes particuliers.

La simplicité de cette traduction découle du fait que l'architecture cible est une architecture RISC. Ses opérations sont donc relativement basiques et peuvent donc facilement être mises en correspondance avec les opérations de base offerte par x86, une architecture CISC.

4.3 Analyse d'activité

L'analyse d'activité dans un compilateur se définit comme un problème d'analyse de flux de données. Le but recherché dans cette analyse est la découverte des endroits dans un programme où une variable est active, c'est-à-dire où elle potentiellement lue avant sa prochaine écriture.

L'analyse d'activité dans un recompilateur, dynamique ou statique, diffère de celle effectuée dans un compilateur traditionnel. Tel que mentionné précédemment, l'approche classique consiste à évaluer l'activité des variables symboliques contenues dans un sous-ensemble de la représentation intermédiaire résultant de transformations appliquées au programme original. Or, dans le cas d'un recompilateur, l'analyse d'activité se concentrera plutôt sur l'utilisation des registres virtuelles dans une partie de l'exécutable traité. Ce problème d'analyse peut être autant approché à partir d'une perspective locale que globale.

Au niveau algorithmique, la communauté de la compilation à depuis longtemps formulé un algorithme pour effectuer une analyse d'activité sur une sous-partie d'un programme. Cet algorithme est explicité ci-dessous.

Data:
$n \rightarrow$ identificateur d'une instruction
in $\rightarrow$ variables actives avant l'exécution d'une instruction
out $\rightarrow$ variables actives après l'exécution d'une instruction
succ $\rightarrow$ identificateurs des instructions pouvant suivre une instruction donnée
use $\rightarrow$ variables utilisées dans une instruction
def $\rightarrow$ variables définies dans une instruction
Initialisation:
foreach *n* **do**
    in[n] $\leftarrow \emptyset$;
    out[n] $\leftarrow \emptyset$;
end
Calcul:
while *in'*=*in* $\wedge$ *out'*=*out* **do**
    foreach *n* **do**
        in'[*n*] $\leftarrow$ *in*[*n*];
        out'[*n*] $\leftarrow$ *out*[*n*];
        in[*n*] $\leftarrow$ use[n] $\cup$ (*out*[*n*] - *def*[*n*]);
        out[*n*] $\leftarrow \cup_{s \in succ[n]}$ *in*[*s*];
    end
end

Algorithm 1: Analys d'activité globale

La bonne image mentale à se faire de cet algorithme est celle d'une analyse de flux dans un graphe de flot de contrôle. Ainsi, le graphe modélisé est en fait la partie du programme sur laquelle l'analyse d'activité s'effectue. L'avantage de cet algorithme est qu'il est facilement généralisable pour plusieurs niveaux d'abstractions. De fait, cet algorithme est autant applicable sur un bloc de base que sur une trace. Il pourrait même être utilisé pour faire de l'analyse et de l'optimisation interprocédurale.

Une approche locale à l'analyse d'activité des registres virtuels consiste à limiter l'analyse du flux de données à une sous-partie bien définie du programme, soit un bloc de base[REF] ou une trace[REF]. Ceci sous-entends donc qu'il est pris pour acquis que la totalité des registres physiques sont disponibles au début de la sous-partie traitée et qu'ils sont tous libérés à la sortie. Ceci empêchera

nécessairement de lier des sous-parties de programme ensemble, mais rendra l'analyse beaucoup plus simple. Le lecteur attentif aura remarqué que l'algorithme ci-dessus illustre l'approche locale à l'analyse d'activité du fait que *in* et *out* sont vides au départ.

L'approche globale à l'analyse d'activité va considérer que certains registres virtuels sont déjà actifs au début d'une sous-partie de programme et que certains vont être actifs dans une sous-partie subséquente. Pour ce faire, il suffit juste de faire passer de l'information entre les sous-parties d'un programme au moyen de *in* et *out*. Ainsi, tout en respectant leurs définitions, on peut allonger l'activité considérée avant et après l'exécution d'une instruction pour englober les sous-parties directement avant et après celle présentement traitée.

4.4 Allocation de registres

Une fois l'analyse d'activité effectuée, le recompilateur possède alors suffisamment d'information pour procéder à l'allocation de registres, c'est-à-dire, associer les registres virtuels de l'architecture émulée avec les registres natifs de l'architecture hôte.

Dans le cas que l'analyse d'activité était uniquement locale, une approche à prendre pour l'allocation de registres natifs est d'associer certains de ces registres avec les registres virtuels les plus utilisés. On n'associera par contre pas l'ensemble de ces registres natifs, préférant plutôt en garder libres pour un ensemble de registres virtuels moins utilisés. Dans le cas de l'émulateur implémenté pour cet article, le nombre de registres natifs libres est de 2. Ce choix a été motivé par la présence d'instructions à trois opérandes dans le jeu d'instructions de l'architecture cible. De cette façon, même si les trois registres virtuels impliqués dans l'instruction courante ne sont associés à des registres natifs au préalable, l'instruction pourra tout de même être exécutée avec un chargement dans un registre suivi d'instructions *in situ*.

Advenant que l'analyse d'activité ait été effectuée de façon globale, il est alors possible de lier les blocs de base recompilés ensemble. Au niveau de l'allocation de registres, cela implique qu'on ne pourra plus utiliser de registres natifs pour des mises en correspondance éphémères avec les registres virtuels les moins utilisés. Par conséquent, on peut s'attendre à des ralentissements, dans le sens d'un gain de vitesse moins élevé sur un interpréteur, lorsque le programme doit aller chercher des données en mémoire. Tout ceci découle du fait que le passage en mode natif est plus long et donc que l'allocation est pensée pour un plus long terme où on ne veut pas souffrir les pénalités associées avec des manipulations trop nombreuses de la pile. Un exposé sur les différents types d'allocateurs de registres globaux va au-delà de la portée de cet article, mais le lecteur curieux est appelé à consulter [REF LSA] pour un aperçu d'une approche énormément utilisée dans le milieu des compilateurs juste à temps.

5 Conclusion

Cet article à décrit une architecture virtuelle minimaliste créée pour faciliter l'acquisition des connaissances nécessaires au développement d'un émulateur. Quelques-unes de ces techniques ont ensuite été décrites. Cette description a été divisée en deux parties, la première exposant des techniques issues du milieu de l'interprétation et la seconde posant les bases théoriques pour la mise en place d'un recompilateur dynamique.

Bien évidemment, l'émulation regroupe beaucoup plus de techniques et d'architectures qu'il n'a été présenté dans cet article. Il serait intéressant, dans un but d'atteindre rapidement la fine pointe de cette technologie, d'utiliser la suite d'outils offerte par LLVM[REF] pour développer un recompilateur dynamique. Aussi, le sujet de l'optimisation du code produit pour l'architecture hôte n'a pas du tout été abordé et il serait pertinent d'y revenir dans un article subséquent.

6 Références

CHIP-8, http://en.wikipedia.org/wiki/Chip8

TODO: Ref x86

TODO: Ref JVM

TODO: Ref LLVM

TODO?: Ref Dolphin

TODO: Ref Linear scan allocation

TODO: Wiki Threaded Code

Chip16 v1.1 specifications

A. V. Aho, R. Sethi, J. D. Ullman, M. S. Lam, *Compilers: Principles, Techniques and Tools, Second Edition*

K. Cooper and L. Toczon, *Engineering a Compiler*

M. Zaleski, A. D. Brown, K. Stoodley, *YETI: a gradually Extensible Trace Interpreter*

N. Topham and D. Jones, *High Speed CPU Simulation using JIT Binary Translation*

M. Berndl, B. Vitale, M. Zaleski, A. D. Brown, *Context Threading: A flexible and efficient dispatch technique for virtual machine interpreters*

M. Anton Ertl and D. Gregg, *The Structure and Performance of Efficient Interpreters*

V. Chipounov and G. Candea, *Dynamically Translating x86 to LLVM using QEMU*

G. S. M. de Paula and H. D. H. O. Gomes, *Analysis and extension of PCSX2, a PlayStation2 emulator*

M. Probst, A. Krall, B. Scholz, *Register Liveness Analysis for Optimizing Dynamic Binary Translation*