Émulateur

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 prgrammation 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 seront essentielles dans un tel projet. Malheureusement, les resources en ligne sur les techniques d'émulation sont parcimonieuses et il se peut donc qu'une personne est bien de la misère à débuter un projet d'émulation pour cette raison.

Le présent article à 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 abordée 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.
TODO

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 diponibles pour mettre en mémoire une ROM. La pile débute à l'adresse 0xFDF0 et occupe 512 bytes. Finalement, les deux ports d'entrées-sorties occupe l'espace allant de l'adresse 0xFFF0 à l'adresse 0xFFFF. En effet, les entrées-sorties dans le modèle architecturale de Chip16 produisent une écriture à des adresses mémoire définies à même les spécification de la machine virtuelle. Détail important à noter, la mémoire est ordonancer 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 registres de 8 bits servant de registre de drapeaux.

Au niveau graphique, Chip16 affiche à une résolution de 320x240 en utilisant des couleurs indexé 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. 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. **TODO: Finish sprite description**. La seconde notion primordiale au niveau graphique est celle de la palette de couleurs. **TODO: Palette**

3 Interpréteur

3.1 Définition et motivation

3.2 Switch Dispatch

3.3 Direct Threading

3.4 Context Threading

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 language 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 dynamique 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 auto-modifiant.

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 ce manifestera surtout lorsque l'on tente d'émuler des consoles de jeux vidéos 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 mise 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 peut 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.**TODO?**

4.2 Émission d'instructions x86

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 traditionnelle. 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 appliqués 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.

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. Un algorithme populaire **TODO**

L'approche globale à l'analyse d'activité va considérer que certains registres virtuels sont déjà actifss au début d'une sous-partie de programme et que certains vont être actifs dans une sous-partie subséquente. **TODO**

4.4 Allocation de registres

TODO : Heuristiques

TODO : Linkage

5 Conclusion

6 Références

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

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*