Méta-morphosis – rapport 2

Owl's team

Pierre Bourdon <bourdo_p@epita.fr>
Tiphaine Petit <petit_t@epita.fr>
David Stavaux <stavau_d@epita.fr>

Table des matières

Notez qu'une version numérique de ce document est disponible au format PDF sur le site web de notre projet : `http://meta-morphosis.ath.cx/`

1 Présentation générale

1.1 Le projet

Notre projet, Méta-morphosis, fait partie de la grande catégorie des compilateurs. Cependant, au lieu de traduire un langage de programmation vers un fichier objet comme le font souvent les compilateurs, nous nous intéressons à un autre type de compilation : la compilation binaire vers binaire.

Cette méthode de compilation a des problématiques terriblement différentes de la compilation classique partant d'un langage de haut niveau et descendant vers le bas niveau : en effet, nous partons d'un binaire de bas niveau, essayons de remonter vers un langage abstrait de plus haut niveau pour ensuite redescendre.

Il est légitime de se demander le but de cette compilation : en effet, quel est l'intérêt de récupérer un binaire lorsque l'on a déjà un binaire ? En fait, les binaires sources et destination ne sont pas forcèment compatibles. Dans notre cas, Méta-morphosis prend en entrée un binaire de PlayStation et renvoie en sortie un binaire pour X86. Cela permet ainsi théoriquement de profiter des jeux de PlayStation sur une machine classique sans passer par une émulation du processeur.

D'une complexité assez conséquente, ce projet nous mettra au défi de la compilation pendant encore une soutenance.

1.2 Le groupe

Notre groupe, la *Owl's team*, semble maudit. Une petite chronologie est de vigueur pour expliquer cette pseudo-malediction :

- Début de l'année : groupe de 4 personnes, constitué de Pierre, Florent, Tiphaine et David
- Conseil de discipline : Florent est exclut de l'école, le groupe est maintenant réduit à trois
- Récupération des rescapés : rescapé d'un groupe de 4 réduit à deux, Julien entre dans le groupe
- 24 heures chrono plus tard, Julien est exclu de l'école
- Deuxième soutenance : nous sommes donc toujours un groupe de 3 personnes.

Ce chaos ambient a provoqué de nombreux problèmes dans la gestion du groupe et dans la répartition des tâches : en effet, il a fallu revoir notre planning pour gérer le départ d'un de nos membres, et travailler à trois pour rattraper le travail du quatrième. Ainsi, le travail fourni pour cette soutenance

n'est pas aussi poussé que nous l'aurions voulu, mais reste dans les limites du convenable établies par le cahier des charges.

1.3 Évolution entre la première et la deuxième soutenance

De nombreuses évolutions ont eut lieu sur notre projet durant la période séparant la première et la deuxième soutenance. Alors que notre première soutenance était basée sur leitmotiv d'avoir un résultat visuel correct, nous avons préféré nous concentrer sur la vitesse d'exécution et les algorithmes de compilation pour cette deuxième soutenance.

De ce fait, toute la partie concernant les bibliothèques d'émulation n'a pas été modifiée, si ce n'est pour garder la compatibilité avec le nouveau code généré par notre traducteur. Aucun nouveau matériel n'est supporté, que ce soit la puce sonore ou le décodeur matériel, et le support de la 3D n'est donc pas encore présent.

Les évolutions sur le traducteur et son fonctionnement interne sont présentées dans la partie suivante. Nous verrons ensuite dans la troisième partie les améliorations prévues sur le long terme (troisième soutenance) pour Méta-morphosis.

2 Recompilation et traduction de code

Le traducteur est la partie la plus importante de notre projet. Brique essentielle permettant de traduire les binaires de l'architecture de la Playstation vers un binaire fonctionnant sur un système d'exploitation moderne et une architecture classique, il a un rôle clé dans la traduction d'un jeu de Playstation pour PC. Malheureusement, c'est également la partie la plus compliquée et la plus difficilement réalisable de ce projet : en effet, il doit reproduire exactement la sémantique du matériel de la Playstation dans le code traduit, et ceci au bit près : toute erreur est « fatale ».

La version de ce programme que nous avons actuellement est en réalité presque complète sur le plan de la traduction des instructions : des équivalents à toutes les instructions compréhensibles par le processeur de la Playstation sont fournis. Sur le plan de l'exactitude, il est difficile de juger de son état sans le tester en situation réelle : en effet, très peu de documentation est disponible sur l'architecture de la Playstation.

Avant d'exposer l'avancement et l'évolution de ce traducteur, nous allons d'abord faire le point sur les technologies utilisées.

2.1 Technologies utilisées

Le traducteur est un programme dont le travail peut être découpé en plusieurs sous-tâches indépendantes qui sont les suivantes :

- La lecture du format de l'exécutable et des segments contenus dans le fichier ;
- Le décodage des instructions contenus dans le segment de code (aussi appelé « text segment ») ;
- La traduction des instructions du processeur de la Playstation ;
- La compilation de la traduction en un binaire pour le système d'exploitation ciblé.

Toutes ces tâches ne sont pas réalisées par notre code. En effet, certaines étapes sont simplement trop complexes pour que nous prenions la peine de le faire alors que des alternatives libres et gratuites existent déjà.

Commençons par la première des 4 étapes évoquées plus haut. Les exécutables de la Playstation sont contenus dans un format très simpliste, constitué d'un header spécifiant les adresses de départ et la taille des segments et de ces segments placés bout à bout à la suite du header. La lecture d'un exécutable est donc très simple et est réalisée par notre code, dans le module `Psx_file`, programmé en langage OCaml.

Le code machine pour les processeurs de type R3000A (pour rappel : le processeur de la Playstation) a été pensé intelligemment et de manière à ce que le décodage des instructions soit le plus simple possible. En effet, il existe moins d'une centaine d'instructions et uniquement 4 type d'instructions [1]. Ainsi, programmer un décodeur d'instructions est très simple et a été une des premières tâches réalisées pour ce projet. Il se trouve sous la forme d'un module : `R3000_decoder`, implémenté en OCAML pour la majeure partie et en C pour les fonctions de manipulation des bits.

La tâche suivante est de traduire chacune de ces instructions, pas directement en code natif mais dans un langage de plus haut niveau et plus portable, nous évitant de refaire ce travail pour chaque architecture cible. C'est la première de ces 4 tâches qui n'est pas en intégralité réalisée par notre projet : en effet, nous nous appuyons sur une bibliothèque nommée LLVM (Low Level Virtual Machine) qui est un framework pour compilateur permettant de générer procéduralement du code assembleur de haut niveau (LLVMIR – LLVM Intermediate Representation). Ainsi, même si notre code réalise le travail de traitement des instructions en créant du code assembleur via LLVM, c'est tout de même l'API exportée par LLVM qui se charge de la vérification de la cohérence de l'ensemble. Le principal module s'occupant de cela est `R3000_instrs` et est programmé en OCAML.

Enfin, il faut traduire cette représentation intermédiaire en code machine et la relier à la bibliothèque de support (`libpsx`) et au BIOS recodé (`libbios`). Cette dernière partie est entièrement réalisée par des outils tiers : en effet, nous utilisons la toolchain de LLVM (`llc` notamment) pour optimiser et compiler la représentation intermédiaire en fichier objet (`fichier.o`), puis nous utilisons `gcc` [2] pour réaliser le travail de link du fichier objet avec les bibliothèques dynamiques.

2.2 Évolution

Tout comme Rome ne s'est pas construite en un jour, il a fallu plusieurs essais et plusieurs reprises du code du traducteur pour arriver au résultat plutôt satisfaisant que nous avons à l'heure d'aujourd'hui. Je vais ici détailler chronologiquement toutes les améliorations qui ont été réalisées sur le traducteur depuis la soutenance précédente.

Cependant, même avec ce que nous avons actuellement, il reste de nombreux problèmes inhérents à l'approche que nous avons choisi. Ces problèmes seront exposés dans la partie suivante.

1. À côté de celà, on s'approche dangereusement des 750 instructions sur nos processeurs X86 et des 30 types d'instructions.
2. En réalité, tout linker pourrait fonctionner, celui qui est utilisé est celui qui a servi à compiler le traducteur.

2.2.1 Soutenance 1 – un fonctionnement naif

L'algorithme de traduction de code utilisé lors des démonstrations de notre première soutenance était en réalité plus que naïf. Parfois plus lent qu'une interprétation directe des instructions, sa structure a cependant servi de base à l'implémentation de toutes les modifications que nous avons réalisés pour cette deuxième soutenance.

Son fonctionnement était très simple : pour chaque instruction présent dans le segment de code du binaire PlayStation, on ajoute une case dans deux tableaux globaux et exportés dans le code généré. Le premier tableau, `instrs`, contient la fonction à appeller pour exécuter l'instruction, et l'autre, `instrs_params`, contient les paramètres de l'instruction.

Du fait du fonctionnement de cet algorithme, la seule phase réalisée à la compilation est le décodage des instructions. Lors de l'exécution, on appelle l'instruction à l'adresse actuelle avec les paramètres du tableau `instrs_params`. Du fait du grand nombre d'indirections et de sauts, l'exécution d'un programme est donc lente mais la compilation très rapide. Le code généré est également très petit, ne contenant que deux tableaux qui seront linkés avec la bibliothèque dynamique d'émulation.

Un autre problème de cette méthode de fonctionnement est que les fonctions de gestion des instructions se trouvent toutes dans une bibliothèque partagée, avec donc deux indirections à chaque saut (une recherche dans la PLT et un saut) et des résolutions de symboles à la première exécution.

Voici, par exemple, un équivalent C du code généré pour un exemple très simple :

```
const void* instrs = {
    psx_instr_add,
    psx_instr_j,
    psx_instr_sll
};

const uint32_t instrs_params = {
    4, 5, 0, 0,
    4242, 0, 0, 0,
    0, 0, 0, 0
};
```

2.2.2 Mise en place d'une traduction plus complète

La première étape pour améliorer les performances du code généré par notre projet est de casser une des indirections, et ce simplement en déplacant du code de la bibliothèque dynamique d'émulation vers le code généré. En effet, si toutes les fonctions doivent accèder à un tableau qui n'est pas dans le même module de compilation, une indirection via la PLT est nécessaire.

Pour résoudre ce problème de performances, nous avons simplement généré une fonction `psx_execute_at`, qui prend un index et exécute l'instruction à cet index en lui donnant les paramètres nécessaires. Ainsi, les accès sont directs et locaux au module, ce qui évite une indirection et un saut qui étaient nécessaires avant.

En comparaison du code précédent, voilà un équivalent C du code généré après cette transformation :

```
void psx_execute_at(int index)
{
    const uint32_t* params = &instrs_params[index * 4];
    instrs[i](params[0], params[1], params[2], params[3]);
}
```

Ainsi, cette amélioration, bien que simple, a permis d'améliorer de manière conséquente mais pourtant aisée à mettre en place les performances générales de notre projet.

2.2.3 Implémentation de l'indirect threading

L'*indirect threading* est une technique courante dans le monde de la programmation d'interpréteurs, plus généralement utilisée dans le cas d'un format de bytecode ayant de nombreux types d'instructions. Même si ici nous ne sommes pas exactement dans ce cas, il existe de nombreuses similarités entre la programmation d'interpréteurs et la mise en place d'une fonction exécutant l'instruction à un emplacement mémoire donné.

Le principe est assez concis à expliquer : à l'intérieur d'une fonction, on définit différents blocs qui gèrent différents types d'instructions (dans un interpréteur, les différentes instructions du bytecode, par exemple). Ces blocs sont repérés par leur adresse. Au début de la fonction, on récupère l'adresse du bloc qui sert à gérer l'instruction que l'on doit exécuter, puis on y saute (via, en C, un *computed goto*, ou en assembleur LLVM, une instruction `indirectbr`).

Dans notre fonction d'exécution, nous avons utilisé ce principe d'indirect threading : lorsque notre fonction nommée `psx_exec_one` est appelée, elle

saute à l'adresse du bloc qui gère non pas le type d'instruction assembleur mais l'adresse virtuelle de l'instruction exécutée. Par exemple, pour ce code assembleur R3000 :

```
sll r1, r1, r1  # pc = 10
addu r1, r2, r1 # pc = 14
```

La fonction `psx_exec_one` sera la suivante (transcrite en C) :

```
void psx_exec_one(void)
{
    void** instrs = { at_10, at_14 };
    goto &&instrs[pc];

at_10:
    psx_instr_sll(1, 1, 1, 0); goto end;
at_14:
    psx_instr_addu(1, 2, 1, 0); goto end;
end:
}
```

Ainsi, un travail plus conséquent est réalisé par notre traducteur : en effet, alors que le code précédemment généré était plutôt léger, la taille du code maintenant généré atteint parfois les plusieurs méga-octets pour des programmes 10x plus petits. En effet, pour une seule instruction assembleur R3000, on génère beaucoup plus d'assembleur X86. Cependant, le gain de vitesse est plus qu'important : là où le précédent traducteur ne faisait en gros que le décodage des instructions, on va là plus dans les détails tout en permettant plus d'améliorations pour la suite des événements.

2.2.4 Traduction des instructions en code natif

Parmi les améliorations directement permises par l'étape précédente vient la plus importante de toute la période entre notre première et deuxième soutenance : la génération directe d'assembleur à la place d'appels de fonctions C émulant les instructions.

Travail simplement titanesque, il s'agissait de reproduire les 80 instructions de l'assembleur MIPS R3000 et d'en faire des fonctions OCaml générant de l'assembleur LLVM. Pour la plupart, les instructions R3000 sont traduisibles en moins de 3 ou 4 instructions LLVM sans compter les chargements et sauvegardes de registres. Ces instructions LLVM sont en majorité traduites directement en une instruction X86 chacune.

Ce travail, malgré sa complexité et l'impossibilité de faire une erreur sans casser l'ensemble des programmes compilés par le projet, a été correctement terminé, et des tests unitaires ont permis de tester le comportement et de le comparer entre avant la modification, après la modification mais également avec un émulateur bien connu, PCSX. Ainsi, nous sommes à près sûrs de la validité de notre traducteur d'instructions.

En pseudo-code, une instruction `add` est traduite comme ceci :

```
Charger contenu du reg 1 dans un reg machine
Charger contenu du reg 2 dans un reg machine
Calculer reg 1 + reg 2 dans un reg machine
Sauver le résultat dans le reg 3
```

De plus, une grande quantité de code est factorisable entre les différentes instructions, ce qui fait que le code réalisant la traduction des instructions est particulièrement court.

2.2.5 Optimisations simples mais efficaces

Plusieurs petites optimisations ont également été mises en place afin d'optimiser aussi bien la vitesse de traduction que la vitesse d'exécution du programme résultant. Celles notables sont listées ici :

- Surement la plus importante étant donné qu'elle a divisé la vitesse de traduction par 15. Les appels de fonction de support dans les gestionnaires d'exceptions étaient gérés via l'instruction LLVM `invoke`. Cependant, ce que nous avons découvert à nos dépends est que cette instruction, en plus de réaliser sa fonction d'invocation de sous-programme, s'occupe de la gestion des points de retour d'exception. Un timing des passes d'optimisation de LLVM (via l'option `-time-passes`) nous a montré que la passe de retrait des exceptions prenait plus de 95% du temps de compilation. Changer toutes les instructions `invoke` en instructions `call` nous a permis de réduire énormément le temps de traduction d'un programme.
- La taille du code que nous générons est plus que conséquente : parfois 10 fois plus importante que la taille du binaire d'origine, il nous est primordial de la faire diminuer le plus possible. Pour cela, une heuristique très naïve mais pourtant plutôt efficace a été mise en place dans cette deuxième version de notre projet : lorsqu'une instruction ne dépendant pas de son emplacement en mémoire est dupliquée, son code n'est généré qu'une seule et unique fois. Cela permet de réduire la taille du code généré d'un facteur presque 10, et rend donc la traduction praticable sur de plus gros binaires.

2.3 Problèmes de l'approche actuelle

Malgré les grandes avancées qui ont été réalisées sur cette partie du projet, le traducteur comporte néanmoins encore de nombreux problèmes majeurs qui le rendent non utilisable en pratique [3]. Le code généré est de plus lent et ne fait aucune supposition sur le contenu des registres avant une instruction.

Une grande partie des problèmes est dû au fait que notre code généré pour remplacer le binaire est stateless : chaque instruction est exécutée une par une sans connaitre le résultat des exécutions d'instructions précédentes, ce qui empêche notamment de réaliser de la propagation de constantes ou un mappage de la mémoire. Ce problème est en cours de résolution par la substitution de notre fonction `psx_exec_one` par une autre fonction, `psx_exec_branch`, qui exécute elle toutes les instructions jusqu'à un saut. Cela implique de remplacer la gestion des sauts (qui est faite en C dans la bibliothèque d'émulation) par de l'assembleur LLVM, mais permettra de nombreuses optimisations futures, aussi bien au niveau de la vitese d'exécution que de la vitesse de traduction.

Également, notre traducteur ne gère pas encore le chargement de données depuis un média externe (port série, lecteur CD-ROM, etc.) qui est nécessaire pour gérer de plus gros jeux. En effet, tous les jeux commerciaux placent leurs données sur le système de fichier du CD-ROM, hors du binaire. Il faudra donc implémenter une bibliothèque de lecture des CD-ROM pour la compilation, afin d'accèder au contenu du disque depuis le traducteur.

3. Comprendre : pour autre chose que des programmes de démonstration d'un SDK

3 Évolutions futures

Malgré l'approche de la date de la soutenance finale, notre projet a bien entendu prévu de nombreuses améliorations à réaliser pour cette dernière deadline. Bien que la perte d'un membre nous obilge à revoir la plupart de nos plannings et de nos prévisions sur la suite des événements, nous avons clarifié les évolutions à mettre en place pour la dernière soutenance de ce projet d'InfoSpé.

Les améliorations majeures prévues pour notre projet sont décrites dans cette partie.

3.1 Remplacement de la bibliothèque d'émulation

La bibliothèque d'émulation est, comme exposé dans le paragraphe 2.1 de ce rapport, une pièce importante du puzzle qu'est notre projet. Son but est de convertir des appels haut niveau PlayStation en appels plus bas niveau pour PC. Elle s'occupe notamment de la gestion des timers, de la DMA, des accès mémoire et de la communication avec les différents composants (GPU, SPU, GTE, DDE).

Le problème de notre bibliothèque d'émulation est qu'elle est actuellement incomplète, et qu'elle ne sera surement jamais complète d'ici la troisième soutenance : représentant un travail bien plus élevé que ce nous nous imaginions, et nécessitant d'énormes connaissances sur la console et son architecture matérielle, elle est cependant nécessaire au fonctionnement de tout programme sur la console.

Nous avons donc décidé qu'il serait intéressant de, plutôt que recoder toute cette partie du projet, réutiliser une bibliothèque d'émulation comme celle de l'émulateur PCSX serait une idée pratiquable et nous permettant de nous concentrer sur notre but premier : programmer un compilateur de binaire vers binaire, plutôt que de travailler sur une partie du projet qui nous semble de plus en plus infaisable au fur et à mesure de l'avancement du projet.

Ainsi, il est probable que le projet montré à la troisième soutenance soit capable de faire fonctionner de vrais jeux de PlayStation en se basant sur les fonctionnalités de PCSX tout en exécutant du code compilé par notre traducteur.

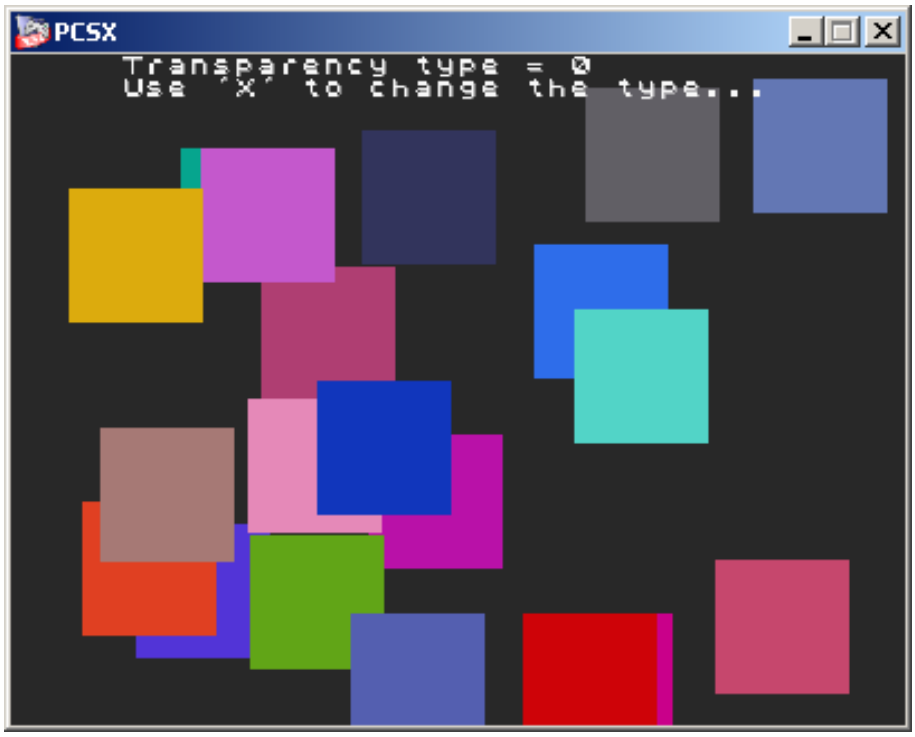

FIGURE 1 – PCSX faisant fonctionner un des programmes de démonstration.

3.2 Programmation d'une bibliothèque d'accès au CD-ROM

Comme il l'a été expliqué dans la partie 2.3 de ce rapport, la plupart des jeux de PlayStation utilisent un lecteur de CD-ROM pour récupérer leurs données sur un disque.

Le problème de cette approche dans notre compilateur est qu'il est impossible de savoir ce qui est code et ce qui est données parmi les octets récupérés via le lecteur de CD-ROM de la PlayStation. Ainsi, sans connaissance des données sur le lecteur, il nous est impossible de traduire l'intégralité du programme. Avoir une bibliothèque de lecture de CD-ROM comprenant le format ISO-9660 Mode 2 Form 2 utilisé par les CD-ROMs de PlayStation.

Cette bibliothèque sera plus que probablement programmée en C pour de meilleures performances, une plus grande proximité du système et pour la facilité qu'a ce langage à manipuler les octets bruts.

Il faut noter que gérer les programmes avec accès externes au système sera certainement la partie la plus difficile de notre projet : en effet, cela demande de réaliser un graphe complet des interdépendances des registres et des accès mémoire dans le programme, afin de pouvoir prévoir *ahead of time* là où seront placées les instructions et donc ce qu'il faut décoder.