Qemu : Visite au cœur de l'émulateur

Yvan Roch

Les machines virtuelles, par leurs avantages et grâce aux progrès des processeurs, sont devenues un domaine très en vogue actuellement même si leurs débuts remontent à l'époque des Beatles. Elles couvrent un large secteur de l'informatique allant du plus ludique émulateur de Commode 64 jusqu'aux plus sérieux hyperviseurs au cœur d'énormes centres de calcul. Les techniques utilisées pour les faire fonctionner sont aussi variées, depuis l'émulation totale jusqu'à la paravirtualisation. Les logiciels existants sont nombreux et Qemu est l'un d'eux, bien connu dans le monde de l'embarqué. Mais, comment fonctionne-t-il ? Et comment émuler son propre matériel ?

1. Introduction

Annonçons-le dès maintenant, cet article ne traite ni de la virtualisation, ni de comment monter une usine à gaz pour utiliser un scanner dont les pilotes n'existent que pour Windows®. Nous parlerons de Qemu, de son fonctionnement interne en émulateur et de la manière de créer une nouvelle plate-forme émulée.

Qemu est bien connu dans l'univers x86. En effet, il y a encore peu de temps, il constituait une des rares solutions libres pour faire tourner des logiciels propriétaires sous GNU/Linux. Mais alors, pourquoi deux articles sur lui dans un contexte dédié à l'informatique embarquée ? Parce qu'il est un merveilleux outil de simulation et de mise au point dans le ce domaine. En voici quelques exemples :

– La formation. Il est beaucoup plus facile et moins coûteux de faire des travaux pratiques sur un système ARM9 simulé dans Qemu qu'avec un système réel.

– Lorsque le développement d'une carte physique n'est pas encore terminé, il est possible de développer les applications sur un système équivalant émulé dans Qemu.

– Lors d'un déplacement en clientèle, une démonstration est toujours plus aisée dans un émulateur.

Je laisse le lecteur curieux consulter les articles [1] et [2] ainsi que l'ouvrage [3]. L'auteur, Pierre Ficheux, y détaille l'utilisation de Qemu dans un contexte embarqué. Il traite de son utilisation simple, le démarrage d'un système ARM9 sous GNU/Linux, et enchaîne sur des sujets avancés allant jusqu'au débuggage du noyau Linux en utilisant le couple Qemu/GDB.

La preuve est donc faite de l'utilité de Qemu dans l'embarqué, d'ailleurs, il est à la base du simulateur de smartphone de la plate-forme de développement Android. VirtualBox s'inspire de sa recompilation dynamique que nous détaillerons plus bas et Xen utilise une émulation matérielle issue de Qemu. Autant de grands noms devraient donner l'envie d'en savoir plus.

Qemu a été créé par Fabrice Bellard, un Français, alors Cocorico !!!. C'est un développeur de talent et il utilise énormément le préprocesseur C ainsi que des fonctionnalités avancées de GCC. Mais comme un avantage comporte toujours sa contrepartie, il résulte que le code de Qemu est difficile à appréhender et peut sembler, au premier abord, confus. Le site officiel de Qemu en fait mention [4]. D'ailleurs, est-ce un hasard si, hormis des records de calcul de décimales de Pi, l'auteur a aussi remporté plusieurs fois le IOCCC [5], International Obfuscated C Code Contest, c'est-à-dire le Concours International de Code C Obscurci. La mise en route de la connaissance du code de Qemu est, certes, un peu ardue, mais accessible à quiconque en fait l'effort. Pour cela quelques outils nous seront bien utiles.

Qemu est un vaste logiciel. Il est capable de réaliser de nombreuses choses. Nous intéresserons uniquement à l'émulation d'un système complet. Plus particulièrement, dans la première partie, nous allons étudier l'émulation d'un système complet ARM9 GNU/Linux sur un système hôte x86 GNU/Linux bien que dans ce cas, les mécanismes génériques restent les mêmes, quels que soient les systèmes hôte et invité. La recompilation dynamique effectuée par le Tiny Code Generator est au cœur de l'émulation et elle sera

détaillée. Nous aborderons aussi la gestion du temps ainsi que les horloges de Qemu. Des parties majeures du logiciel comme la virtualisation complète KVM, l'émulation de la MMU, l'émulation des périphériques en mode blocs, l'émulation d'interfaces réseau, le serveur GDB et bien d'autres encore, tant la liste est longue, ne seront pas étudiées. Il faudrait un ouvrage entier pour décrire tout le fonctionnement interne de Qemu.

Dans la seconde partie, nous passerons à la pratique en implémentant une nouvelle plate-forme matérielle ARM9 dans Qemu. Notre choix s'est porté sur un système ARM9 Armadeus APF27. Seuls les composants strictement nécessaires au démarrage d'un noyau Linux seront créés, c'est-à-dire la CPU, la mémoire, le contrôleur d'interruption, un timer et un contrôleur de port série.

2. Introduction générale à Qemu : Élégance et complexité

Qemu est un logiciel élégant et complexe. Par contre, comme c'est souvent le cas dans le monde du logiciel libre, il a été écrit par des développeurs passionnés pour lesquels le défit technique compte plus que la rédaction de spécifications et de la documentation technique. Sans aucun jugement moral, force est de reconnaître que la documentation de l'architecture est quasiment inexistante et que le code comporte très peu de commentaires. Au risque de paraphraser Alan Cox [6], « Si le code est bon, vous pouvez le lire sans commentaire », la seule documentation solide est le code source lui-même. Il est à notre disposition, c'est déjà bien, de plus il, fonctionne.

2.1 Description de l'arborescence des sources et du système de compilation

La version de Qemu étudiée est la version 1.0 sortie le 1er décembre 2011. L'archive du code source peut être obtenue à l'URL **http://wiki.qemu.org/download/qemu-1.0.tar.gz**. Seuls les répertoires utiles à la bonne compréhension de l'article seront décrits. En voici une brève description :

– La partie centrale du code de Qemu se trouve directement à la racine de l'archive.

– Le répertoire **hw** contient le code des plates-formes et des périphériques matériels émulés.

– Les répertoires **target-*** contiennent le code nécessaire à l'émulation d'architecture de processeur *. Cela va de l'ARM au s390.

– Le répertoire **tcg** contient le Tiny Code Generator, son code commun et celui spécifique aux systèmes hôte.

Le système de compilation utilise un script **configure**. Voici la configuration utilisée dans le cadre de cet article :

```
./configure --prefix=/somewhere --target-list=arm-softmmu --disable-user --enable-sdl --extra-cflags=-save-temps --enable-debug
```

Passons en revue ces différents paramètres :

– **--target-list=arm-softmmu** active l'émulation d'un système ARM complet

– **--disable-user** désactive le support en mode utilisateur. Sans cette option, il est possible de lancer l'émulation d'un programme en mode utilisateur sans un système émulé complet.

– **--enable-sdl** active le support de la bibliothèque SDL qui permet, par exemple, d'avoir l'émulation d'un frame buffer.

– **--extra-cflags=-save-temps** est une option pour les explorateurs que nous sommes. Elle sera détaillée en temps voulu.

– **--enable-debug** active la génération des symboles de débuggage pour pouvoir lancer confortablement Qemu sous le contrôle de GDB.

La compilation nécessite d'avoir une station de développement GNU/Linux digne de ce nom, c'est-à-dire dotée de tous les outils de développement classiques GNU. La configuration génère un répertoire **arm-softmmu** dans lequel se retrouvent les produits de la compilation.

2.2 Quelques conseils et outils avant de partir en exploration

Comme nous l'avons déjà dit, le code de Qemu est complexe. Mais la tâche de Qemu est tout aussi complexe. Il est vrai cependant qu'il peut tomber dans le domaine du compliqué et de l'obscur. Pour ne pas perdre de nombreuses heures et se décourager, mieux vaut se doter de quelques outils avant de partir en exploration. Le premier est un outil de navigation dans le code source. Un tel outil permet d'effectuer des

recherches efficaces dans un code source de grande taille. Il permet en particulier de trouver :

– toutes les références à un symbole,

– Les fonctions appelées par une fonction,

– Les fonctions appelant une fonction particulière,

– Des chaînes de caractères correspondant à des expressions régulières,

– Les fichiers inclus par d'autres fichiers.

Suivant les sensibilités de chacun, deux outils semblent convenir : Cscope [7] et l'indexeur de code C/C++ d'Eclipse [8].

Un outil de navigation dans le code source est nécessaire, mais il n'est pas suffisant. Prenons un exemple simple qui, pour être concret, est issu de la deuxième partie de l'article. Une des dernières lignes du fichier définissant une nouvelle carte à émuler se termine par ce qui pourrait ressembler à un appel de la fonction **`machine_init()`**. Voici la dernière ligne du fichier **`apf27.c`** :

```
[...]
machine_init(apf27_machine_init)
```

L'absence de point-virgule pourra être notée et se comprendra un peu plus bas. **`apf27_machine_init()`** est une fonction définie dans le même fichier **`apf27.c`**. Naïvement **`machine_init()`** pourrait être pris pour une fonction qui prend en argument un pointeur de fonction. Lors du lancement de Qemu, **`apf27_machine_init()`** est effectivement appelée. Après quelque temps passé à comprendre comment cela était possible et à analyser le code source, un fait était clair : il n'y a aucun appel direct à **`apf27_machine_init()`** ni à **`machine_init()`**. Mais **`machine_init()`** est en réalité une macro dont voici la définition :

```
#define machine_init(function) module_init(function, MODULE_INIT_MACHINE)
#define module_init(function, type)                                     \
static void __attribute__((constructor)) do_qemu_init_ ## function(void) {  \
        register_module_init(function, type);                           \
}
```

Non trivial, n'est-ce pas… Voici de l'expansion de la macro :

```
static void __attribute__((constructor)) do_qemu_init_apf27_machine_init(void) {
        register_module_init(apf27_machine_init, MODULE_INIT_MACHINE);
}
```

Ce que l'on pourrait prendre pour un étrange appel de fonction et en réalité la définition d'une fonction. D'où la non-nécessité du point-virgule. Mais alors comment cette nouvelle fonction **`do_qemu_init_apf27_machine_init()`** est-elle appelée ? L'attribut GCC **`__attribute__((constructor))`** indique à GCC que cette fonction doit être appelée automatiquement avant **`main()`**. Elle est donc lancée sans aucun appel explicite.

Par ailleurs, remarquons l'utilisation de l'opérateur **`##`** de concaténation du préprocesseur C. Lorsque des symboles sont définis avec cet opérateur, il est quasiment impossible de les retrouver directement dans le code source (fichiers **`*.c`** ou **`*.h`**) avec des outils de recherche textuelle tels que **`grep`**. Même l'indexeur d'Eclipse, qui fait pourtant de l'expansion dynamique de macros, s'y perd. Cela se comprend, car pour résoudre de tels symboles, il faudrait exécuter le préprocesseur depuis le début. Mais ce début dépend du système de construction basé sur **`make`**, qui dépend lui-même de nombreuses variables qui peuvent être elles aussi dynamiques. Donc pour résoudre ces symboles, il faut exécuter la compilation.

En effet, il sera impossible de retrouver l'expression **`do_qemu_init_apf27_machine_init(void)`** puisqu'elle n'existe pas textuellement pas sous cette forme dans les fichiers **`*.c`** et **`*.h`**.

Lors du processus de compilation, le préprocesseur lit les fichiers **`*.c`** et **`*.h`**, étend les macros et stocke le résultat dans des fichiers temporaires **`*.i`** qui sont ensuite passés en argument au compilateur. GCC les efface à la fin de la compilation. C'est dans ces fichiers que les symboles définis avec **`##`** sont identifiables sous leurs formes finales. Dans cet exemple relativement simple, il est possible de s'en sortir manuellement, mais il existe des cas où un symbole utilisé dans le code n'est défini explicitement nulle part ailleurs (voir la note sur les symboles d'opcodes du Tiny Code Generator). L'option **`-save-temps`** de GCC est alors d'un grand secours en conservant les fichiers d'expansion **`*.i`**. Il suffira alors de fouiller dans ces fichiers **`*.i`** pour trouver l'expansion des macros.

Dans notre exemple, lors du processus de compilation du fichier **`afp27.c`**, le préprocesseur fait l'expansion de toutes les macros dans un fichier **`apf27.i`** qui sera ensuite passé en argument au compilateur. Il serait normalement effacé à la fin de la compilation, mais, l'option **`-save-temps`** permet de le conserver. La solution consiste alors à aller regarder directement dans ce fichier **`apf27.i`** pour savoir comment cette macro obscure a été étendue.

Cette option peut être appliquée à tous les fichiers en passant l'option **`--extra-cflags=-save-temps`** au script de configuration **`configure`**. Il suffira alors de lancer un **`grep -r THE_LOST_SYMBOL *`** à la racine des sources de Qemu pour retrouver rapidement ce qui aurait pris autrement plusieurs heures.

Après coup cet exemple peut paraître simple, nous verrons des cas bien plus retors plus loin. Il met cependant en exergue trois caractéristiques du code de Qemu :

– Le préprocesseur C est très utilisé.

– Les spécificités de GCC sont très utilisées.

– Le code peut être complexe, voire compliqué.

Un troisième outil s'avérera très utile : Valgrind [9]. Valgrind est un ensemble d'outils de débuggage et d'optimisation. L'un d'entre eux, Callgrind, permet d'établir des graphes d'appels de fonctions. Lorsque qu'un programme comme Qemu comporte des milliers d'appels de fonctions, il peut être intéressant d'avoir une vision globale sous la forme d'un graphe d'appel pour y voir plus clair. Les fichiers générés par Callgrind peuvent ensuite être visualisés avec KCachegrind [10]. KCachegrind nécessite Graphviz [11] pour la visualisation des graphes. Voici un exemple d'utilisation de Callgrind :

```
valgrind --tool=callgrind /usr/local/qemu-imx/bin/qemu-system-arm -M apf27 -kernel apf27-linux.bin -append "console=ttymxc0" -nographic -initrd apf27-rootfs.cpio
```

Callgrind génère un fichier **`callgrind.out.PID`** où **`PID`** est le numéro de processus de Callgrind. Pour obtenir les noms de fonctions dans le fichier de traces, Qemu doit être compilé avec l'option de débuggage. Il faudra noter que le lancement d'un programme sous le contrôle de Callgrind ralenti énormément son exécution, dans un rapport de 20 à 100 selon la documentation.

Au risque d'énoncer une banalité, la simple fonction **`printf()`** s'avère aussi très efficace pour savoir si une partie du code est exécutée ou connaître la valeur d'une variable. Judicieusement placée et après une recompilation elle constitue un outil puissant en comparaison de sa facilité de mise en œuvre. Par exemple, la statistique, présentée plus bas, donnant le nombre d'instructions par bloc de base, a simplement été produite avec **`printf()`**, **`grep`** et **`awk`**.

Pour finir avec l'équipement du randonneur, lorsque tous les outils précédents ne suffisent pas, il reste l'arme lourde, GDB. Il nécessite aussi que l'option de débuggage soit activée lors de la compilation. C'est grâce à lui, par exemple, que le mystérieux tableau **`code_gen_prologue[]`**, dévoilé ensuite, a livré ses secrets.

2.3 Architecture générale de l'exécution de Qemu

Nous allons maintenant plonger dans le code de Qemu. La quantité de code parcourue est assez importante et plutôt que de copier/coller de large portion de code dont je n'ai pas la paternité, j'ai préféré ne pas gaspiller de place. Aussi, je conseille au lecteur curieux de télécharger l'archive de Qemu, de la décompresser et de suivre les différents fichiers cités avec son éditeur de texte favori. Leurs chemins sont exprimés à partir de la racine de l'archive.

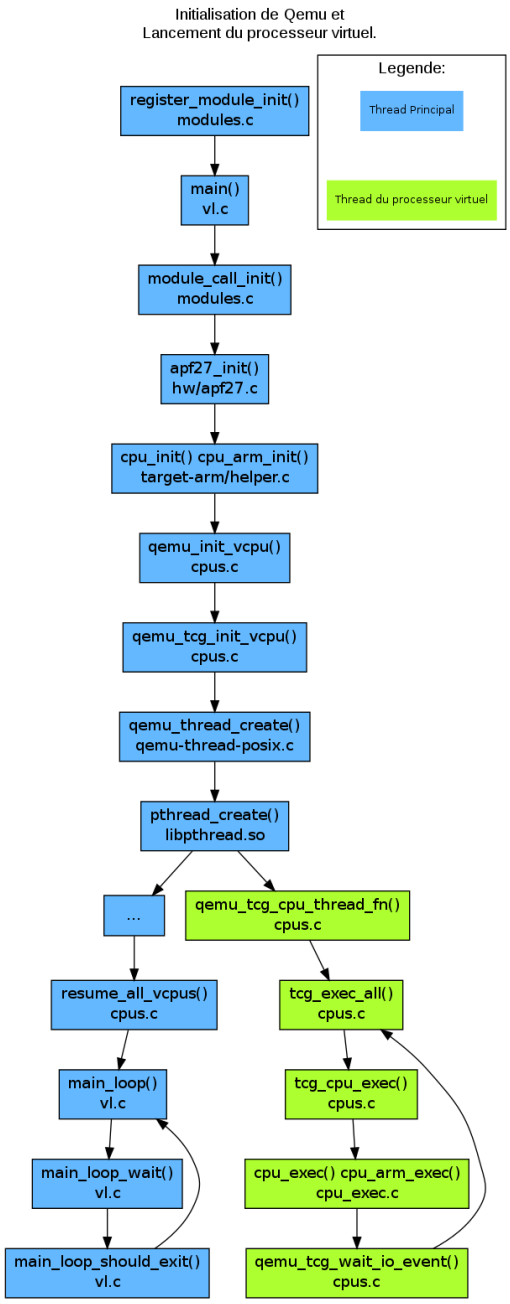

Fig. 1 : Graphe d'appels simplifié de l'initialisation de Qemu et du lancement du processeur virtuel.

Les graphes d'appels présentés ont été réalisé avec le trio Callgrind/KCachegrind/Graphviz. Pour des raisons de clarté, ils ne représentent pas exactement la réalité de l'exécution de Qemu. D'une part, de nombreux appels de fonctions non essentiels ont été supprimés. D'autre part un arc ne représente pas obligatoirement un appel direct mais un ordre d'appel. Par exemple ***`cpu_arm_exec()`*** *n'appelle jamais* ***`qemu_tcg_wait_io_event()`*** *directement, mais ces fonctions sont appelées dans cet ordre.*

La figure 1 présente, d'un point de vue très macroscopique, la séquence de lancement du processeur virtuel depuis le début jusqu'à la boucle d'exécution du code invité. Tout commence avant **`main()`**, avec un appel à **`register_module_init()`** pour chaque module matériel disponible. Cette phase sera détaillée en seconde partie. Seulement après **`main()`** procède au lancement de toutes les initialisations nécessaires au bon fonctionnement de Qemu. Entre autres **`module_call_init()`** initialise l'ensemble de modules matériels via leurs fonctions d'initialisation. **`apf27_init()`** du fichier **`hw/apf27.c`** est l'une d'elles. Le lecteur studieux et attentif aura remarqué que ce fichier n'est pas présent dans l'archive officielle de Qemu ! Il provient du patch présenté en seconde partie dans laquelle sera abordée l'implémentation de l'émulation d'une carte Armadeus APF27. Étant la fonction d'initialisation de la carte émulée, il lui revient d'initialiser le processeur virtuel qu'elle utilisera via **`cpu_init()`** qui est définie, pour l'architecture ARM, par **`cpu_arm_init()`**. Cette définition permet d'appeler la même fonction **`cpu_init()`** quel que soit l'architecture du processeur. Le code spécifique au processeur virtuel effectue l'initialisation de celui-ci, puis **`qemu_tcg_init_vcpu()`** crée un thread dédié à l'exécution du code invité en appelant **`qemu_thread_create()`**. Elle lance, par l'intermédiaire de **`pthread_create()`** de la bibliothèque pthread, un nouveau thread constitué de la fonction

qemu_tcg_cpu_thread_fn(). Cette dernière configure les signaux de ce thread et se met en attente de la fin de l'initialisation générale de Qemu. Lorsque Qemu est complètement opérationnel, **resume_all_vcpus()** démarre le processeur virtuel. Ensuite le thread principal entre dans une boucle d'attente qui gère les actions annexes de l'émulation comme les opérations d'entrées/sorties.

Maintenant que tout est prêt, l'émulation du code invité peut commencer au sein du thread dévolu au processeur virtuel. Les fonctions **tcg_exec_all()**, **tcg_cpu_exec()** et **cpu_exec()** sont exécutées temps que le processeur virtuel est actif. L'inactivité est principalement déterminée par les champs **halted**, **interrupt_request**. **stop** et **stopped** de la structure **CPUState** décrite plus loin. Si **halted** vaut 1 et que **interrupt_request** vaut 0, le processeur est inactif. **halted** est mis à 1 lors de l'émulation de l'exécution de l'instruction ARM **wfi** qui met le processeur en mode d'attente d'interruption avec arrêt de l'horloge. **interrupt_request** est mis à 1 lorsqu'une interruption est déclenchée. **stop** est mis à 1 lorsque qu'une demande d'arrêt du processeur virtuel est en cours et **stopped** l'est lorsqu'il est arrêté. C'est le cas lorsque la commande **stop** du moniteur de Qemu est invoquée. La boucle est constituée par un simple **while (1)** dans **qemu_tcg_cpu_thread_fn()**. **cpu_exec()** comporte, elle aussi, une boucle infinie qui exécute à chaque itération une portion de code invité sur le processeur hôte. Elle est interrompue si :

– Une interruption matérielle est simulée par **qemu_irq_raise()**, c'est le cas le plus courant.

– Lors d'une exception si le débuggage est activé.

– Lors d'une exception de demande d'arrêt du processeur virtuel, cela ne se produit pas en fonctionnement normal.

– Lorsque le champ **exit_request** de la structure **CPUState** est non nul, ce qui se produit par exemple lors d'un signal **SIG_IPI** (**main-loop.h**) est envoyé au thread du processeur virtuel. Le handler du signal **cpu_signal()** positionne **exit_request** à 1. Cela peut être provoqué par un appel à **qemu_mutex_lock_iothread()** lorsque le thread principal a besoin de l'exclusivité de l'exécution.

– Lorsqu'il n'y a plus d'instruction à traiter. Ce cas ne devrait jamais survenir avec Linux, car il existe toujours des instructions à venir. Cela se produit cependant lorsque Qemu fonctionne en mode utilisateur et que le programme se termine.

Après exécution de **tcg_exec_all()**, **qemu_tcg_wait_io_event()** est appelé et le thread du processeur virtuel est mis en attente par un appel de **pthread_cond_wait()** si celui-ci n'est pas actif. Il est ensuite relancé par un appel à **pthread_cond_broadcast()** dans **qemu_cpu_kick()**, elle-même appelée par **tcg_handle_interrupt()** de **exec.c**. qui est principalement déclenché par les interruptions du timer.

Ce survol de l'exécution générale de Qemu fait apparaître que la fonction **cpu_exec()** est au cœur de l'exécution du code émulé. C'est en son sein que nous poursuivrons nos investigations.

3. Le cœur de l'émulateur : TCG ou Tiny Code Generator

Qemu utilise la recompilation dynamique pour émuler une architecture invitée. Cette technique permet d'exécuter du code binaire prévu pour un certain processeur sur un autre processeur qui a des opcodes complètement différents. Le canevas général est le suivant : lors de l'exécution du code invité celui-ci est lu et désassemblé. Il est ensuite traduit en opcodes équivalents du processeur hôte. Pour finir, ce code hôte généré dynamiquement, est exécuté.

Cependant, comme Qemu émule de nombreuses architectures invitées sur de nombreuses architectures hôtes, une phase intermédiaire a été ajoutée à la recompilation dynamique. Au lieu de traduire directement le code invité vers le code hôte, il est traduit en un bytecode intermédiaire puis ce dernier est traduit en code hôte. L'intérêt de passer par un bytecode intermédiaire est de faciliter le support de nouvelles architectures émulées et aussi celui de nouvelles architectures hôtes. En effet si la traduction était faite directement depuis le code de l'architecture émulé vers le code de l'architecture hôte, il serait nécessaire, pour chaque architecture émulée d'écrire son traducteur de code pour chaque hôte spécifique. Si M architectures émulées sont supportées sur N architectures hôtes, il faudrait écrire M*N traducteurs spécifiques. C'est fastidieux et inutile. En introduisant l'utilisation d'un bytecode intermédiaire :

– Le support d'une architecture émulée consiste en l'écriture d'un traducteur unique du code de l'architecture émulée vers le bytecode intermédiaire.

– Le support d'une architecture hôte consiste en l'écriture d'un traducteur unique du bytecode intermédiaire vers le code de l'architecture hôte.

Ainsi, au lieu de M*N traducteurs, il n'est plus nécessaire que d'en écrire M+N. Voici donc les différentes phases de l'exécution du code invité dans Qemu :

– 1. Lecture d'un bloc de base du code invité.

– 2. Désassemblage de ce bloc de base et traduction en un bloc de bytecode TCG.

– 3. Traduction du bloc de bytecode TCG en un bloc d'opcodes natifs du processeur hôte.

– 4. Exécution de ce bloc natif sur le processeur hôte.

Avant d'aller plus loin, arrêtons-nous sur la notion de bloc de base. Qu'est-ce qu'un bloc de base ? Pourquoi l'utiliser comme unité de traduction ? Dans un programme, un bloc de base est une portion de code qui comporte un seul point d'entrée et un seul point de sortie. Cela signifie qu'il ne contient pas de code qui soit la cible d'une instruction de saut et que seule la dernière instruction peut lancer l'exécution d'un autre bloc de base. Il en découle que, lorsque la première instruction d'un bloc de base est exécutée, toutes les autres instructions de ce bloc de base le seront, une seule fois, en respectant leur ordre. Ce découpage en blocs de base facilite l'analyse du code par les compilateurs et les émulateurs comme Qemu. Les instructions de saut non conditionnel représentent une des situations permettant à ceux-ci de détecter la fin d'un bloc de base (nous verrons les autres plus bas). En effet il serait inutile de continuer à traduire du code invité lorsqu'on ne sait pas si celui-ci sera exécuté.

Fabrice Bellard a publié, en 2005, un papier [12] sur le fonctionnement interne du générateur de code dynamique de Qemu. Malheureusement ce document est obsolète et ne correspond plus aux mécanismes présents de la version actuelle. En effet avant la version 0.10, Qemu utilisait l'outil dyngen pour la phase 3 de la traduction dynamique. Grossièrement, cette technique consistait à recopier des blocs de code machine du processeur hôte générés par GCC. Chaque bloc copié correspondait à une instruction de bytecode intermédiaire. Le code recopié pouvait éventuellement être modifié avec les adresses des paramètres de l'instruction. La suite des blocs copiés donnait le code exécutable sur le processeur hôte. Cette technique comportait un inconvénient majeur : elle utilisait des options avancées et spécifiques à une version précise de GCC et Qemu devait être alors compilé avec cette version particulière. Depuis, même si le canevas de génération de code reste le même, la traduction utilise maintenant le TCG ou Tiny Code Generator.

Comme je l'ai mentionné plus haut, nous nous focaliserons sur l'émulation d'un processeur ARM9 sur un hôte x86. Ce cas correspond à une situation courante dans le monde de l'embarqué, mais grâce à son architecture générique, ce qui suit est facilement transposable à un autre couple invité, hôte. Le TCG comporte deux parties. La première est spécifique au processeur cible et elle réalise les phases 1 et 2 présentées plus haut. La seconde est spécifique au processeur hôte et elle prend en charge la phase 3. Elles sont reliées par du code commun à toutes les architectures. Le TCG comporte donc trois types de code :

– Du code générique à tous les couples invité, hôte : **`translate-all.c`**, **`tcg/tcg.c`**

– Du code spécifique à l'invité ARM9 : **`target-arm/translate.c`**

– Du code spécifique à l'hôte : **`tcg/i386/tcg-target.c`**

Concernant les architectures émulées, il en existe 14 dans Qemu, situées dans les répertoires **`target-XXX/`**, où **`XXX`** représente l'architecture allant du gentil **`m68k`** au monstrueux **`s390x`**. De même les architectures hôtes supportées, au nombre de 9, sont situées dans les répertoires **`tcg/YYY/`**. Elles sont appelées en terminologie Qemu des « TCG Target » bien qu'il n'y ait aucun rapport avec les « cibles » émulées. Ce terme signifie simplement que le TCG est capable de générer du code natif pour ces processeurs.

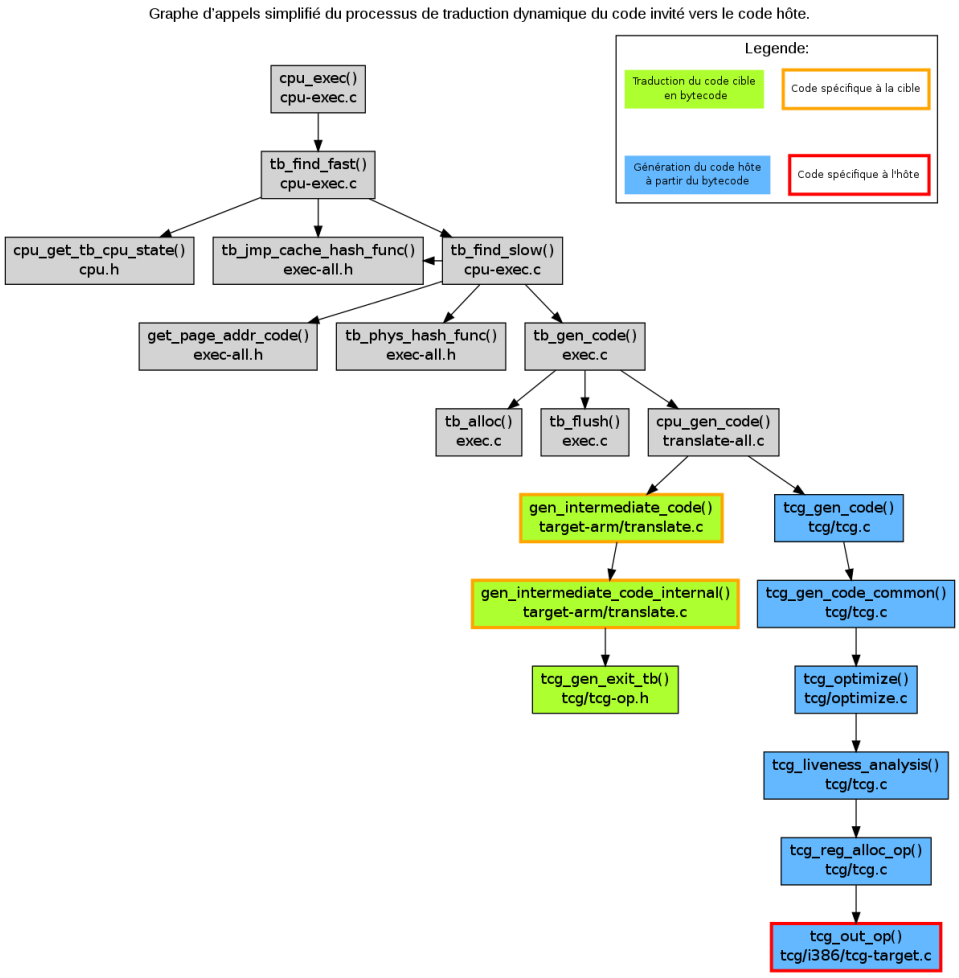

Fig. 2 : Graphe d'appels simplifié du processus de traduction dynamique du code invité vers le code hôte.

Le graphe d'appel simplifié présenté sur la figure 2 représente le processus global de traduction dynamique de code. Il commence dans la fonction **`cpu_exec()`** présentée précédemment. Pour faciliter la compréhension, nous sauterons dans un premier temps les premières étapes, pour y revenir ensuite. Allons donc directement dans la fonction **`cpu_gen_code()`** du fichier **`translate-all.c`**. C'est elle qui lance les phases 1 à 3 de la traduction. En voici le prototype :

```
int cpu_gen_code(CPUState *env, TranslationBlock *tb, int *gen_code_size_ptr);
```

Le premier argument de type **`CPUState`** est une structure complexe qui contient toutes les informations nécessaires concernant le processeur virtuel émulé ainsi que son état. Elle comporte une partie commune à tous les processeurs émulés, définie par la constante **`CPU_COMMON`** située dans le fichier **`cpu-defs.h`**. Chaque architecture complète cette base avec ses besoins spécifiques. Dans le cas de l'ARM, **`CPUState`** est habilement définie dans **`target-arm/cpu.h`** par :

```
#define CPUState struct CPUARMState
```

pour continuer avec :

```
typedef struct CPUARMState {
...
CPU_COMMON
...
} CPUARMState;
```

`CPUState` n'est explicitement définie nulle part. Cette technique, utilisant le préprocesseur, permet d'employer des noms de symboles génériques, structures ou fonctions, dans le code commun et de les définir dans le code spécifique à l'architecture émulée. Les informations présentent dans **`CPUState`** vont des registres du processeur virtuel, aux drapeaux d'interruptions en passant par l'état d'exécution.

Le deuxième argument de type **`TranslationBlock`**, définie dans **`exec-all.h`**, est une structure qui représente un « Translation Block » ou TB dans la terminologie Qemu. Pour l'instant, pour pouvons

considérer qu'un TB correspond à un bloc de base, même si une nuance, dévoilée plus bas, existe. Cette structure renferme toutes les informations concernant un bloc de base, c'est-à-dire un pointeur sur le code invité de ce bloc de base, sa taille, son contexte, un pointeur sur le code hôte traduit, ainsi que d'autres données facilitant l'exécution du code recompilé. Cette structure est déjà allouée et initialisée par le système de cache que nous verrons plus bas. Le troisième argument est un pointeur sur la taille du code hôte généré.

Une autre structure importante est **`TCGContext`** définie dans **`tcg/tcg.h`**. Elle contient les données du contexte courant du TCG. Les principales sont ses variables internes utilisées dans un ou plusieurs blocs de base, les pointeurs sur le tampon contenant le code hôte généré et sur l'instruction courante du code hôte généré. Un TB correspond à une fonction TCG. Lorsque le code invité sera complètement traduit en code hôte natif, ces fonctions TCG, images des blocs de base du code invité, seront exécutées. Les variables internes du TCG sont utilisées pour remplacer les variables réelles du code invité. Une variable **`TCGContext`** globale unique **`tcg_ctx`** est utilisée, déclarée dans **`translate-all.c`**.

3.1 La partie invité

Après avoir réinitialisé le contexte du TCG via l'appel à la fonction **`tcg_func_start()`**, **`cpu_gen_code()`** lance la lecture, le désassemble et la traduction du code invité via la fonction **`gen_intermediate_code()`**. Avec cette dernière, nous entrons dans le code spécifique à l'invité contenu dans le fichier **`target-arm/translate.c`**, voici son prototype :

```
void gen_intermediate_code(CPUState *env, TranslationBlock *tb) ;
```

Elle prend en argument les variables **`CPUState`** et **`TranslationBlock`** obtenues précédemment. Elle appelle la fonction **`gen_intermediate_code_internal()`** qui prend les mêmes arguments. Cette fonction est le cœur de la traduction du code invité. Elle lit et traduit le code invité en bytecode TCG, jusqu'à ce qu'elle détermine la fin du bloc de base courant puis elle retourne. Elle est principalement constituée d'une boucle **`do { … } while`** qui désassemble les opcodes ARM et qui s'arrête sur l'une des conditions suivantes :

– Une instruction de saut conditionnelle ou une exception sont trouvées (**`DISAS_JUMP`** ou **`DISAS_TB_JUMP`**).

– Une instruction qui change l'état du processeur virtuel est trouvée (**`DISAS_UPDATE`**).

– La taille maximale du tampon de bytecode TCG est atteinte.

– Le mode pas à pas est activé.

– Un changement de page mémoire survient dans le code invité.

– Le nombre maximal d'instructions par bloc de base est atteint.

Ces conditions déterminent la fin d'un bloc de base du code invité. Lorsqu'une fin de bloc de base est trouvée, les instructions **`INDEX_op_exit_tb`** et **`INDEX_op_end`** de bytecode TCG sont ajoutées. La première sera utile pour assurer le retour de la fonction TCG de code hôte générée correspondant à ce bloc de base de code invité. La seconde arrête la génération du code hôte pour ce bloc de base. Dans les deux premiers cas, les constantes citées, définies dans **`exec-all.h`**, sont affectées au membre **`is_jmp`** du **`DisasContext dc`**. Cette structure permet de gérer le contexte de désassemblage du code machine ARM.

Le désassemblage en lui-même est assuré par les fonctions **`disas_thumb_insn()`** ou **`disas_arm_insn()`** suivant que le processeur virtuel ARM utilise le jeu d'instruction Thumb ou non. Ces énormes fonctions, plus de 1000 lignes, sont très complexes et elles font appel à de nombreuses autres fonctions. Ensembles, elles traduisent le code ARM en bytecode TCG. Pour bien les comprendre, une connaissance approfondie des processeurs ARM et de leurs jeux d'instructions est nécessaire. Elles constituent un exemple de l'essentiel des développements à effectuer pour le support d'une nouvelle architecture invitée dans Qemu. Comme nous sommes au cœur du système, prenons le temps d'examiner un exemple. Celui-ci sera simple puisqu'il concerne le désassemblage d'une instruction ARM de non logique. Voici une ligne de code ARM désassemblée issue d'un programme :

```
e1e03003      mvn r3, r3
```

L'instruction mnémonique assembleur ARM **`mvn`** signifie « Move Not ». C'est le moyen, sur un processeur ARM, d'effectuer un non logique. L'instruction présentée calculera le non logique du registre **`r3`** et stockera le résultat dans **`r3`**. **`e1e03003`** est le code hexadécimal correspondant. Pour faciliter la compréhension du désassembleur de Qemu, analysons ce code hexadécimal. En binaire, il devient **`1110 00 0 1111 0 0000 0011 000000000011`**. Les bits ont été regroupés en parties signifiantes :

– bits 28 à 31 **`1110`** : Condition d'exécution de l'instruction : toujours (Page A3-4).

– bits 26,27 **`00`** : Toujours à zéro (Page A4-82).

– bit 25 **`0`** : I, si I est positionné adressage immédiat, donc adressage par registre direct (Page A4-82).

– bits 21 à 24 **`1111`** : Opcode pour **`mvn`** (Page A4-82).

– bit 20 **0** : S, si S vaut zéro, le registre d'état CPSR (Current Program Status Register) ne sera pas mis à jour (Page A1-5 et A4-82)

– bits 16 à 19 **0000** : Toujours à zéro (Page A4-82).

– bits 12 à 15 **0011** : Registre de destination, valeur 3 donc registre **r3** (Page A4-82).

– bits 0 à 11 **000000000011** : Opérande décalé, valeur 3 donc l'opérande est le registre **r3** sans décalage (Page A5-8).

Les pages citées renvoient au manuel de référence de l'architecture ARM [13]. Ce document est téléchargeable gratuitement sur le site officiel de la documentation ARM en ayant créé un compte au préalable. Voici maintenant une portion du code de la fonction **disas_arm_insn()** simplifiée au maximum pour les besoins didactiques (… représentent des lignes supprimées et les déclarations de variables sont minimales) :

```
static void disas_arm_insn(CPUState * env, DisasContext *s)
{
        unsigned int insn, op1, rm, rd;
…
        TCGv tmp2;
…
        insn = ldl_code(s->pc);
        s->pc += 4;
…
        else if (((insn & 0x0e000000) == 0 &&
                        (insn & 0x00000090) != 0x90) ||
                        ((insn & 0x0e000000) == (1 << 25))) {
                int set_cc, logic_cc;

                op1 = (insn >> 21) & 0xf;
                set_cc = (insn >> 20) & 1;
                logic_cc = table_logic_cc[op1] & set_cc;
                if (insn & (1 << 25)) {
…
                } else {
                        rm = (insn) & 0xf;
                        tmp2 = load_reg(s, rm);
…
                }
…
                rd = (insn >> 12) & 0xf;
                switch(op1) {
…
                case 0x0f:
                        tcg_gen_not_i32(tmp2, tmp2);
                        if (logic_cc) {
                                gen_logic_CC(tmp2);
                        }
                        store_reg_bx(env, s, rd, tmp2);
                        break;
                }
…
}
```

La variable **insn** contient le code hexadécimal complet de l'instruction sur 32 bits à savoir dans notre cas **0xe1e03003**. **op1** contient l'opcode de l'instruction sur 4 bits, **0x0f** ici. **rm** et **rd** représentent respectivement les numéros de registres opérande et destination de l'instruction. **tmp2** est une variable temporaire. Elle est de type **TCGv** qui est en fait un **int**. La valeur de l'instruction courante est d'abord chargée dans **insn** via la macro **ldl_code()**. Elle vaut alors **0xe1e03003**. Ensuite, comme cette instruction a une taille de 32 bits, le compteur de programme du contexte de désassemblage est incrémenté de 4 octets. Le **if** suivant sert à détecter une instruction ARM de traitement de données. Comme c'est notre cas, **(insn >> 21) & 0xf** isole les bits 21 à 24 correspondant à l'opcode de l'instruction, **op1** vaut alors **0xf**, l'opcode de l'instruction **mvn**. Le bit 20 est ensuite testé pour savoir s'il faut mettre à jour le registre d'état du processeur virtuel, dans notre cas non. Dans le cas contraire, le drapeau **logic_cc** serait positionné en fonction de **table_logic_cc**. Celui-ci est un simple tableau constant, indexé par les opcodes, qui indique si l'opcode concerné met à jour le registre d'état du processeur. Puis le bit 25 est testé pour identifier le mode d'adressage immédiat. Notre instruction ne l'utilise pas, car nous sommes dans un cas d'adressage direct par registre. Le numéro du registre de l'opérande est isolé par **(insn) & 0xf** et **rm** prend la valeur 3. Le registre de l'opérande est **r3**. La fonction **load_reg(s, rm)** provoque deux opérations. Elle crée une nouvelle variable interne du TCG

puis elle charge le registre référencé par **rm** dans cette nouvelle variable. Cela à pour conséquence de générer une première instruction de bytecode TCG : **INDEX_op_mov_i32** via la fonction **tcg_gen_mov_i32()** indirectement appelée par **load_reg()**. Pour finir la valeur du registre est stocké dans **tmp2**. Le numéro du registre de destination est identifié par **(insn >> 12) & 0xf** et il est stocké dans **rd** qui prend la valeur 3 correspondant au registre **r3**. Suit un **switch** qui teste les différents opcodes. Celui de l'instruction **mvn** est **0xf**. Une deuxième instruction de bytecode TCG est générée par l'appel de la fonction **tcg_gen_not_i32()** : **INDEX_op_not_i32**. Si la mise à jour du registre d'état du processeur virtuel avait été requise via le drapeau **logic_cc**, la fonction **gen_logic_CC()** générerait une autre instruction de bytecode TCG, mais ce n'est pas le cas. Pour finir notre désassemblage, la fonction **store_reg_bx()** génère une troisième instruction de bytecode TCG : **INDEX_op_mov_i32** et stocke la valeur de la variable **tmp2** dans le registre référencé par **rd**.

Les constantes **INDEX_op_mov_i32**, **INDEX_op_not_i32** sont en réalité les opcodes du bytecode TCG. Ils sont stockés dans le tableau global **gen_opc_buf[OPC_BUF_SIZE]**. Leurs paramètres sont stockés dans le tableau global **gen_opparam_buf[OPPARAM_BUF_SIZE]**. Ces deux tableaux sont définis dans **translate-all.c**. Ce sont les fonctions de génération d'instruction de bytecode TCG telles que **tcg_gen_mov_i32()** et **tcg_gen_not_i32()** (**tcg/tcg-opc.h**) qui les alimentent en maintenant les pointeurs courants d'opcodes et de paramètres que sont **gen_opc_ptr** et **gen_opparam_ptr** (**tcg/tcg.c**). La suite du processus de recompilation dynamique utilisera ces tableaux pour générer le code hôte natif.

*Les constantes d'opcodes du bytecode TCG telles que **INDEX_op_end**, **INDEX_op_mov_i32**, **INDEX_op_not_i32** ne sont explicitement définies nulle part dans le code de Qemu bien quelles soient beaucoup utilisées. Un grand magicien du préprocesseur C en a décidé ainsi. Elles sont définies par l'intermédiaire de plusieurs macros. La première, présente dans **tcg/tcg.h**, est la suivante :*

```
typedef enum TCGOpcode {
#define DEF(name, oargs, iargs, cargs, flags) INDEX_op_ ## name,
#include "tcg-opc.h"
#undef DEF
        NB_OPS,
} TCGOpcode;
```

*Ensuite, chaque opcode est défini dans **tcg/tcg-opc.h**, voici l'exemple de l'opcode **INDEX_op_not_i32** :*

```
DEF(not_i32, 1, 1, 0, IMPL(TCG_TARGET_HAS_not_i32))
```

*On peut noter l'utilisation de l'opérateur de concaténation du préprocesseur C **##** dans **INDEX_op_ ## name**. Lors de l'expansion, on obtient une magnifique énumération C, avec en prime une constante **NB_OPS**, générée automatiquement, contenant le nombre total opcodes :*

```
typedef enum TCGOpcode {
…
INDEX_op_movi_i32,
…
INDEX_op_not_i32,
…
        NB_OPS,
} TCGOpcode;
```

*Ce résultat est observable en passant l'option **--extra-cflags=-save-temps** au script de configuration **configure**. Tous les fichiers incluant **tcg/tcg.h** ou **tcg/tcg-opc.h** comporteront cette expansion. C'est le cas de **target-arm/translate.c** dont l'expansion après la compilation pour un système ARM complet (option **--target-list=arm-softmmu** du script de configuration **configure**) est contenue dans le fichier **arm-softmmu/translate.i**. Cette écriture est, certes, fort élégante, mais la lecture et l'analyse du code sont l'occasion d'intenses séances d'arrachage de cheveux… Heureusement, les miens sont bien implantés !!!*

Notre petite ligne d'assembleur ARM a été traduite en bytecode TCG équivalent :

```
mov_i32 tmp9,r3
not_i32 tmp9,tmp9
mov_i32 r3,tmp9
```

Le bytecode TCG ci-dessus est issu d'une trace d'exécution que nous aborderons ensuite. La variable interne du TCG s'appelle **tmp9**. Nous pouvons remarquer qu'une simple ligne d'assembleur ARM a été convertie en trois instructions de bytecode TCG. Cela laisse augurer les piètres performances de l'émulation. D'aucuns diraient, « pourquoi ne pas la convertir directement en opcode x86, puisque c'est le système hôte, un **not %ebx** aurait bien suffi !!! ». Tout d'abord, en procédant ainsi, il aurait fallu développer 14 × 9 = 126 traducteurs distincts (nombre d'architectures invités × nombre d'architectures hôtes) au lieu de 14 + 9 = 23 traducteurs en utilisant un bytecode intermédiaire. Par ailleurs, le cas présenté est extrêmement simple et l'ensemble des cas réels nécessiterait une analyse bien plus poussée. Ainsi, le gain de performance d'un nombre réduit d'instructions serait contrebalancé par une analyse plus gourmande en ressources. Cette technique confère en pratique de bonne performance à Qemu et c'est le prix à payer pour une émulation complète. Cependant, plusieurs optimisations, que nous découvrirons bientôt, sont mises en œuvre.

3.2 La partie hôte

À ce stade du processus de traduction dynamique, nous disposons des tableaux **gen_opc_buf[]** et **gen_opparam_buf[]** contenant respectivement les opcodes du bytecode TCG et leurs paramètres. Ceux-ci représentent, d'un point de vue logique, l'équivalent d'un bloc de base du code invité. La phase 3 du processus consiste en la traduction du bytecode intermédiaire en code natif de l'hôte, qui est dans notre cas un x86. Retournons dans la fonction **cpu_gen_code()** du fichier **translate-all.c**. Lorsque **gen_intermediate_code()** retourne, le mécanisme de chaînage des blocs de base est tout d'abord initialisé

via l'affectation des membres **tb_next_offset**, **tb_jmp_offset** et **tb_next** des structures **TranslationBlock *tb** et **TCGContext *s** précédemment évoquées. Ensuite la fonction **tcg_gen_code()**, définie dans **tcg/tcg.c**, est appelée. En voici son prototype :

```
int tcg_gen_code(TCGContext *s, uint8_t *gen_code_buf);
```

Elle prend en argument un pointeur sur le contexte courant du TCG ainsi qu'un pointeur **gen_code_buf** qui pointe sur un tampon qui recevra le code hôte natif généré par la suite. Elle retourne un entier qui contient la taille du code généré. L'allocation et la gestion de la mémoire nécessaire au code hôte généré seront détaillées plus loin lorsque le système de cache sera étudié. Pour l'instant, il suffit de savoir que **gen_code_buf** et le membre **tc_ptr** du TB courant pointe sur de la mémoire disponible et exécutable pouvant accueillir du code hôte. Cette fonction lance **tcg_gen_code_common()** qui a des arguments presque identiques. Après l'optimisation du bytecode TCG, présentée plus bas, elle est principalement constituée d'une boucle infinie **for(;;)**. Cette dernière parcours le tableau **gen_opc_buf[]** et s'arrête lorsque l'opcode TCG **INDEX_op_end** est trouvé. Il correspond à la fin d'un bloc de base.

La boucle de traitement est un **switch** qui énumère tous les opcodes TCG possibles, éventuellement en appelant la fonction **tcg_reg_alloc_op()**. Suivant l'opcode traité, elles appellent des fonctions présentes dans **tcg/i386/tcg-target.c** qui génèrent les opcodes x86 et particulièrement **tcg_out_op()**. Cette partie du code est particulièrement complexe. En effet, dans certains cas, il peut avoir plusieurs allers-retours entre du code générique de **tcg/tcg.c** et du code spécifique de l'hôte de **tcg/i386/tcg-target.c**.

Pour continuer notre exemple précédant du non logique, voyons comment l'instruction de bytecode TCG

```
not_i32 tmp9,tmp9
```

est traduite en instructions binaires x86. Dans **tcg_reg_alloc_op()** la génération des instructions x86 est lancée par :

```
tcg_out_op(s, opc, new_args, const_args);
```

s de type **TCGContext** pointe vers le contexte courant du TCG. **opc** contient l'opcode de bytecode TCG qui vaut dans notre cas **INDEX_op_not_i32**. **new_args** est un tableau de **TCGArg** qui contient les paramètres de l'instruction **opc**. Ils proviennent d'une recopie des éléments de **gen_opparam_buf[]** correspondant à l'instruction courante. **const_args** est un tableau d'entiers de la même taille que **new_args** indiquant si l'élément **new_args[n]** est une constante (**const_args[n]=1**) ou un registre (**const_args[n]=0**). Passons à **tcg_out_op()** :

```
#define OPC_GRP3_Ev        (0xf7)
#define EXT3_NOT   2
static inline void tcg_out_op(TCGContext *s, TCGOpcode opc,
                const TCGArg *args, const int *const_args)
{
        int rexw = 0;
…
        switch(opc) {
…
        OP_32_64(not):
                tcg_out_modrm(s, OPC_GRP3_Ev + rexw, EXT3_NOT, args[0]);
        break;
…
        }
}
```

La macro **OP_32_64(not):** est simplement étendue en **case INDEX_op_not_i32:**. La constante **OPC_GRP3_Ev** est le premier octet l'opcode x86 pour désigner les instructions unaires du groupe 3 dont l'instruction **NOT** fait partie (cet opcode **0xf7** est partagé avec les instructions x86 **TEST**, **NEG**, **MUL**, **IMUL**, **DIV** et **IDIV**). Il nécessite d'être suivi de l'octet ModR/M pour spécifier l'instruction et le registre opérande. Pour l'instruction **NOT**, les bits 3 à 5 de l'octet ModR/M doivent valoir **010b** qui correspondent à la valeur 2 de la constante **EXT3_NOT**. Les détails des instructions x86 pourront être trouvés dans la documentation officielle Intel® 64 and IA-32 Architectures Software Developer's Manual [14] pages 2-5 Vol. 2A et 4-162 Vol. 2B.

args[0] correspond à l'opérande de l'instruction de bytecode TCG c'est-à-dire **tmp9**. Il contient le numéro d'index dans le tableau des variables du TCG **s->temps[]**. Cet opérande est donc **s->temps[args[0]]**. Admettons que **args[0]** vaille 3. L'appel effectué est donc **tcg_out_modrm(s, 0xf7, 2, 3);**. Poursuivons avec cette fonction :

```
static void tcg_out_modrm(TCGContext *s, int opc, int r, int rm)
{
        tcg_out_opc(s, opc, r, rm, 0);
        tcg_out8(s, 0xc0 | (LOWREGMASK(r) << 3) | LOWREGMASK(rm));
}
```

soit après l'expansion des macros :

```
static void tcg_out_modrm(TCGContext *s, int opc, int r, int rm)
{
  (tcg_out_opc)(s, opc);
  tcg_out8(s, 0xc0 | ((r) << 3) | (rm));
}
```

car **tcg_out_opc** est modifié par :

```
#define tcg_out_opc(s, opc, r, rm, x)  (tcg_out_opc)(s, opc)
```

pour être définie par ailleurs :

```
#define P_EXT         0x100          /* 0x0f opcode prefix */
#define P_DATA16      0x200          /* 0x66 opcode prefix */
static void tcg_out_opc(TCGContext *s, int opc)
{
        if (opc & P_DATA16) {
                tcg_out8(s, 0x66);
        }
        if (opc & P_EXT) {
                tcg_out8(s, 0x0f);
        }
        tcg_out8(s, opc);
}
```

Pour continuer dans **tcg/tcg.c** :

```
static inline void tcg_out8(TCGContext *s, uint8_t v)
{
  *s->code_ptr++ = v;
}
```

Analysons maintenant ces opérations. **tcg_out_modrm(s, 0xf7, 2, 3);** va donc effectué l'appel suivant :

```
tcg_out_opc(s, 0xf7);
```

Qemu étiquette les instructions avec préfixe par les constantes **P_DATA16** ou **P_EXT**. Ce n'est pas le cas de l'opcode **0xf7**. L'appel à **tcg_out8(TCGContext *s, 0xf7)** produira le premier octet **0xf7** de code binaire x86 dans le tampon **s->code_ptr**.

Puis l'appel à **tcg_out8(s, 0xc0 | (2 << 3) | 3);** produit le second octet de l'opcode, appelé l'octet ModR/M. Détaillons le ([14] pages 2-5 Vol. 2A et suivantes) :

– **0xc0 = 11b**, constituant les bits 7 et 8 du champ mod, signifie que les bits 3 à 5 désignerons soit un registre opérande, soit un complément d'opcode d'instruction unaire qui est notre cas.

– **(2 << 3)** constitue les bits 3 à 5, soit **010b**. C'est le complément d'opcode, qui, associé à **0xf7** produit, l'instruction x86 **NOT**.

– **3** constitue les bits 0 à 2, soit **011b**. C'est le numéro de registre de l'opérande soit registre x86 **EBX**.

Le résultat du ou logique fourni **11010011b**, soit **0xd3**. L'appel à **tcg_out8(TCGContext *s, 0xd3)** produira le second octet **0xd3** de code binaire x86 dans le tampon **s->code_ptr**.

Les octets **0xf7** et **0xd3** sont dans le tampon de code x86 qui sera exécuté dans la phase 4. Après désassemblage du code x86 généré, nous obtenons :

```
f7 d3   not    %ebx
```

L'instruction de bytecode TCG **not_i32** a bien été traduite en une instruction binaire x86 équivalente. Nous voici arrivé au terme des opérations de désassemblage du code invité et de génération dynamique du code hôte binaire. Les deux derniers paragraphes ont pu être quelque peu indigestes. Pourtant, le cas étudié est l'un des plus simples. Il a cependant mis en avant la complexité de la recompilation dynamique qui nécessite une connaissance pointue des architectures hôte et invitée.

3.3 Les optimisations

Avant d'étudier la phase 4 d'exécution du code hôte, arrêtons-nous quelques instants sur les optimisations de l'émulation. Nous avons effectivement vu que la recompilation dynamique génère beaucoup plus d'instructions que le code initial n'en contient. Il en résulte une grande perte de performances. Pour conserver une vitesse d'exécution acceptable, plusieurs mécanismes d'optimisation sont mis en œuvre dans Qemu.

Allocation de la mémoire et gestion système de cache :

La mémoire dédiée au code hôte généré et celle des « Translation Block », TB, est allouée dans la fonction **code_gen_alloc()** de **exec.c** appelée au lancement de Qemu. Elle est pointée par le pointeur global **code_gen_buffer**. L'allocation pour le code est réalisée via un appel à **mmap()** avec une autorisation en exécution (**PROT_EXEC**) pour que le code hôte généré puisse s'exécuter. Par défaut, la taille allouée pour le code hôte s'élève au quart de la mémoire du système émulé, c'est-à-dire 32Mo dans notre cas. Le nombre de TB alloués en découle en faisant l'hypothèse que la taille moyenne du code hôte associé à un TB est de 128 octets, soit par défaut 262144 TB. Ce nombre de TB est généralement bien supérieur aux 32768 entrées de l'index de cache et cela permet de n'avoir à vider le cache que rarement. Le système de cache utilise l'intégralité de cette mémoire. Lorsqu'un TB et son code hôte associé sont créés, ils ne sont jamais supprimés tant qu'il reste de la mémoire disponible. Le cache est intégralement vidé lorsque, soit le nombre maximal de TB a été atteint, soit que toute la mémoire de réservée au code hôte est utilisée. L'espace libre courant de **code_gen_buffer** est pointé par **code_gen_ptr**. La valeur de ce pointeur est copiée dans le membre **tc_ptr** de chaque TB lors de son initialisation.

Lors de l'exécution d'un programme, il est courant que le même bloc de code soit exécuté de nombreuse fois, c'est notamment le cas dans une boucle **while{ … }**. Sans système de cache, les quatre phases du TCG se répéteraient alors pour produire au final le même code hôte exécutable. C'est inutile et contre-performant. Qemu intègre donc un cache de code hôte qui contient les TB et leur code associé. À l'issue des trois premières phases du TCG, le TB et son code hôte exécutable sont référencés dans les index du cache. Ainsi, lorsqu'une portion de code invité doit être exécutée, le cache est inspecté pour voir si ce code n'est pas déjà traduit et disponible. Si tel est le cas, les phases 1 à 3 sont court-circuitées et le code hôte correspondant est lancé directement. Sinon, les 4 phases du TCG sont lancées. Dans la version actuelle de Qemu, le cache comporte deux index :

– Le membre **struct TranslationBlock *tb_jmp_cache[]** de la structure **CPUARMState** de taille 4096 (**target-arm/cpu.h** et **cpu-defs.h** via la macro **CPU_COMMON**).

– **TranslationBlock *tb_phys_hash[]** de taille 32768 (**exec.c**).

Le premier index contient les pointeurs sur les 4096 TB les plus récemment exécutés tandis que le second contient les pointeurs sur les 32768 TB les plus récemment générés.

Pour présenter le fonctionnement du cache, revenons au lancement du code invité sur l'hôte. La fonction **cpu_exec()** lance le processus de recompilation dynamique via la fonction **tb_find_fast()** de **cpu-exec.c**. Celle-ci récupère, via la fonction **cpu_get_tb_cpu_state()** de **target-arm/cpu.h**, l'état courant d'exécution du processeur virtuel émulé. Les éléments pris en compte sont le registre **pc** du compteur de programme et le registre **flags** d'état du processeur. Le registre **pc** est passé dans une fonction de hachage qui fournit un index dans le **tb_jmp_cache[]**. Si cet emplacement contient un pointeur valide sur une structure **TranslationBlock**, et que les membres **pc** et **flags** de ce TB sont égaux à ceux de l'état courant, le TB sera exécuté directement sans lancer les phases 1 à 3 de la traduction dynamique. Sinon, **tb_find_slow()** est lancée. Un procédé similaire au précédent, prenant en compte aussi les adresses physiques de l'espace mémoire du processeur virtuel, scrute l'index **tb_phys_hash[]**. Si un TB valide est trouvé, son pointeur est stocké dans l'index court **tb_jmp_cache[]** et il est directement exécuté. Sinon le processus de recompilation dynamique est lancé depuis le début via **tb_gen_code()**. Cette identification d'un TB en considérant uniquement ses membres **pc** et **flags** pourrait être sujette à erreur. En effet, du code différent situé à la même adresse mémoire **pc**, dans le même état du processeur **flags** pourrait avoir pris la place du code du TB placé dans le cache. Mais cette situation ne peut pas se produire, car dans la gestion de la mémoire du système invité faite par Qemu (que nous n'étudierons pas ici), lorsque du code est chargé à des adresses qui ont déjà été utilisées par un TB précédent, cet ancien TB est invalidé dans le cache.

Chaînage direct des blocs de base :

Jusqu'à présent, pour faciliter la compréhension, nous avons admis qu'un « Translation Block », TB, correspondait à un bloc de base. La documentation du TCG de Qemu fait la différence entre un TB et un bloc de base (basic block). Dans le cas le plus général, un TB correspond exactement à un bloc de base. Cependant, pour des raisons de performance et dans les cas les plus courants, plusieurs blocs de base peuvent s'enchaîner directement. C'est notamment le cas lorsque la suite du code se trouve dans la même page mémoire ou encore lorsque l'adresse d'une instruction de saut se trouve dans la même page mémoire. Le TCG chaîne alors les différents blocs de base et ils forment ainsi un seul et même TB.

Le chaînage des blocs de base est prévu lors de la phase 2 (traduction du code invité en bytecode TCG) via l'instruction de bytecode TCG **INDEX_op_goto_tb**. Cette instruction est générée lorsque le cas le permet par la fonction **gen_goto_tb()** de **target-arm/translate.c**.

Puis qui, dans la phase 3 (traduction du bytecode TCG en code hôte), pour un « TCG Target » i386,

INDEX_op_goto_tb est traduite dans la fonction **tcg_out_op()** de **tcg/i386/tcg-target.c** sous la forme d'une instruction x86 **JMP 0x00000000** (opcode **0xe9**, opérande **0x00000000**). C'est une instruction de saut proche relatif qui ajoute au registre **EIP** (le registre x86 de pointeur d'instructions) la valeur de l'opérande. Donc, dans notre cas, **EIP** est inchangé et l'instruction ne fait donc rien.

Ensuite, dans la phase 4, après avoir été exécuté, le TB précédant est examiné dans la fonction **cpu_exec()** pour savoir s'il peut être chaîné avec le TB suivant. Si c'est le cas, il est modifié dynamiquement via la fonction **tb_add_jump()** de **exec-all.h**. Cette modification consiste à remplacer l'opérande de l'instruction **JMP**, vue ci-dessus, valant initialement **0x00000000**, par la valeur du décalage de l'adresse du début du code du TB suivant.

Lorsque l'exécution d'un TB sera lancée via ma macro **tcg_qemu_tb_exec()**, que nous détaillerons plus bas, ce TB sera exécuté ainsi que les TB qui lui sont chaînés. Cette technique de chaînage permet l'exécution ininterrompue de grande portion de code hôte natif au sein de la même itération de la boucle **cpu_exec()** sans retourné dans le code de l'émulateur. Les performances de l'émulation s'en trouvent ainsi grandement améliorées. En effet, selon des mesures faites expérimentalement sur le boot d'un noyau Linux, un TB contient au minimum une instruction du processeur émulé avec plus de 100 instructions au maximum (106 exactement). Mais un TB contient en moyenne de cinq instructions. S'il y a aussi peu d'instructions dans un TB, cela vient de l'énorme utilisation d'instructions conditionnelles telles que **if … then … else**. Comme chaque lancement TB correspondant à un appel de fonction sur le processeur hôte, on comprend alors le grand intérêt du chaînage de TB.

Optimisations du bytecode TCG :

À la fin de la phase 2 de la recompilation dynamique, lorsque l'intégralité d'un bloc de base a été traduit en bytecode TCG, ce dernier bénéficie d'optimisations. Elles sont réalisées par les fonctions **tcg_optimize()** de **tcg/optimize.c** et **tcg_liveness_analysis()** de **tcg/tcg.c** lancées dans **tcg_gen_code_common()**.

La première optimisation, **tcg_optimize()**, supprime les instructions TCG qui ne modifient pas le résultat du traitement. En voici un exemple :

```
and_i32 t0, t0, t0
```

Cette instruction effectue un et logique entre le registre **t0** et lui-même et stocke le résultat dans **t0**. Le résultat est bien évidement **t0**, qui restera inchangé. Ce genre d'instruction est simplement supprimée.

La seconde optimisation, **tcg_liveness_analysis()**, supprime les mouvements entre les variables mortes ainsi que les instructions dont le résultat est inutile. Dans le bytecode TCG suivant :

```
add_i32 t0, t1, t2
add_i32 t0, t0, t3
mov_i32 t0, t3
```

La seule instruction qui sera conservée est **mov_i32 t0, t3**. Les deux premières affectent **t0** qui est ensuite écrasé par la troisième, elles sont donc inutiles.

3.4 L'exécution du code hôte

À ce stade du processus de traduction dynamique, nous disposons de blocs de code hôte natif prêts à être exécutés. L'exécution est lancée via la macro **tcg_qemu_tb_exec()** dans la fonction **cpu_exec()** :

```
uint8_t *tc_ptr;
TranslationBlock *tb;
…
tb = tb_find_fast(env);
…
tc_ptr = tb->tc_ptr;
next_tb = tcg_qemu_tb_exec(env, tc_ptr);
```

env pointe sur une structure **CPUState**, déjà décrite plus haut, qui contient l'état courant du processeur virtuel. **tb** contient le TB à exécuter. **tc_ptr** pointe sur le bloc de code hôte exécutable de ce TB. La valeur retournée **next_tb** contient un pointeur qui permettra de chainé ce TB avec son successeur (voir le chaînage direct ci-dessus). Pour un hôte x86, la macro **tcg_qemu_tb_exec()** est étendue en :

```
next_tb = ((long __attribute((regparm(3))) (*)(void *, void *))code_gen_prologue)(env, tc_ptr);
```

Amusant, non ? Ne nous décourageons pas devant cette syntaxe cryptique et procédons par étapes, en la décomposant, pour bien la comprendre :

```
next_tb = (code_gen_prologue)(env, tc_ptr);
```

C'est un appel de fonction dont le code est pointé par le pointeur **code_gen_prologue**, qui est déclaré dans

exec.c de la manière suivante :

```
uint8_t code_gen_prologue[1024] ;
```

code_gen_prologue[1024] est donc un tableau de 1024 **uint8_t** (**unsigned char**) alloué statiquement et **code_gen_prologue** est équivalent à un pointeur de type **unsigned char***. Nous reviendrons plus tard sur ce curieux tableau.

```
(long (*)(void *, void *))code_gen_prologue
```

code_gen_prologue étant de type ***unsigned char**, il nécessite une conversion explicite de type (cast) avant d'être utilisé comme pointeur de fonction. La ligne ci-dessus le converti en pointeur de fonction qui accepte comme arguments deux pointeurs de type **void*** et qui retourne un **long**.

```
__attribute((regparm(3)))
```

Pour finir, l'attribut **regparm(3)** de GCC [15] fait que les 3 premiers paramètres de la fonction sont passés dans **EAX**, **EDX**, and **ECX** au lieu d'être placés sur la pile. Il résulte, lors de l'appel, que **env** se retrouve dans **EAX** et **tc_ptr** dans **EDX**.

Reprenons, notre ligne de C est en fait un appel de fonction dont le code est contenu dans le tableau **code_gen_prologue[]**. En raison des contraintes de typage fort du langage C, **code_gen_prologue** est converti en pointeur de fonction prenant deux arguments de type **void*** et retournant un **long**. Les deux arguments de l'appel de fonction seront transmis via EAX et EDX en raison de l'attribut de **regparm(3)**.

Cette fonction, qui n'est définie explicitement nulle part, est appelée à chaque exécution d'un TB. Son code est généré dynamiquement lors de l'initialisation de Qemu via la fonction **tcg_target_qemu_prologue()** de **tcg/i386/tcg-target.c**. Le tableau **code_gen_prologue[]** est présent dans la section **.bss** comme le sont les variables non initialisées allouées statiquement. Pour des raisons évidentes de sécurité, la mémoire de cette section n'est pas exécutable par défaut. La zone mémoire de **code_gen_prologue[]** est rendue exécutable grâce à l'appel **mprotect()** doté de l'autorisation **PROT_EXEC**. Cette opération est réalisé au moment de l'allocation de la mémoire pour le TCG dans la fonction déjà citée **code_gen_alloc()**. La fonction **tcg_target_qemu_prologue()** utilise les fonctions du TCG target i386 pour générer le code hôte binaire du prologue et de l'épilogue du lancement des TB. Voici le code généré pour un hôte x86 :

```
0xb8634360 <+0>:     push   %ebp
0xb8634361 <+1>:     push   %ebx
0xb8634362 <+2>:     push   %esi
0xb8634363 <+3>:     push   %edi
0xb8634364 <+4>:     add    $0xfffffd74,%esp
0xb863436a <+10>:    mov    %eax,%ebp
0xb863436c <+12>:    jmp    *%edx
0xb863436e <+14>:    add    $0x28c,%esp
0xb8634374 <+20>:    pop    %edi
0xb8634375 <+21>:    pop    %esi
0xb8634376 <+22>:    pop    %ebx
0xb8634377 <+23>:    pop    %ebp
0xb8634378 <+24>:    ret
```

Ce code assembleur a été obtenu en exécutant Qemu sous le contrôle de GDB et en lançant la commande GDB **disas code_gen_prologue**. Tout d'abord, l'adresse **0xb8634360** correspond à l'emplacement mémoire du tableau **code_gen_prologue**, c'est-à-dire l'adresse de la fonction appelée par la macro **tcg_qemu_tb_exec()**. Elle est déterminée au moment du chargement du fichier binaire exécutable de Qemu et elle peut donc varier entre deux lancements successifs de Qemu. Par contre elle reste fixe tout au long d'une même exécution. Les quatre premières instruction **push** constituent la sauvegarde sur la pile des registres **%ebp**, **%ebx**, **%esi** et **%edi** selon les conventions standard d'appel de GCC [16].

L'instruction **add $0xfffffd74,%esp** alloue un espace de 652 octets sur la pile en retirant 652 à **%esp** (**$0xfffffd74** est la représentation en complément à deux de -652). Cette constante est déterminée par différentes constantes génériques du TCG et d'autres spécifiques à l'hôte i386. L'espace ainsi alloué, appelé « stack frame » sera utilisé, entre autres, pour stocker les variables internes du TCG.

mov %eax,%ebp affecte à **%ebp** l'adresse de la structure **CPUState env**, qui est contenue dans **%eax** en raison de l'attribut **regparm(3)**. Cette structure capitale pourra donc être accédée directement par le registre de pointeur de base.

jmp *%edx saute à l'adresse pointée par **%edx** qui contient **tc_ptr** en raison de l'attribut **regparm(3)**. Cette adresse est le début du code du TB à exécuter. Remarquons que le contrôle de l'exécution est donné au TB via une instruction **jmp** et pas par une instruction **call**. Il sera de la responsabilité du code du TB de reprendre l'exécution à l'instruction suivante lorsqu'il se termine qui, dans notre cas, se situe à l'adresse **0xb863436e**. Il faut donc que cette adresse soit connue par chaque TB lorsqu'ils sont générés dynamiquement. Cette pirouette est réalisée de la manière suivante :

– Lorsque la fonction **tcg_target_qemu_prologue()** génère le prologue que nous analysons, après avoir

produit l'instruction **jmp** ci-dessus, elle affecte à la variable globale **tb_ret_addr** la valeur l'adresse courante du code hôte généré contenue dans le membre **code_ptr** de la structure **TCGContext**, qui vaut dans notre exemple **0xb863436e**.

– Lors du désassemblage du code ARM invité, effectué à la phase 2, lorsque qu'une fin de bloc de base est détectée, l'instruction de bytecode TCG **INDEX_op_exit_tb** est générée via la fonction **tcg_gen_exit_tb()**. Ensuite lorsque cette même instruction est traduite en code hôte x86 à la phase 3, une instruction **jmp tb_ret_addr** est insérée comme dernière instruction du bloc de code hôte, c'est-à-dire **jmp 0xb863436e** dans le cas présent.

L'exécution reprend donc après la dernière instruction du prologue. Cette première instruction de l'épilogue est **add $0x28c,%esp** qui ajoute 652 à **%esp**. Le « stack frame » alloué précédemment se trouve ainsi libéré. Les quatre instructions **pop** restaurent les registres **%ebp**, **%ebx**, **%esi** et **%edi**, pour finir avec un ret qui rend la main à la fonction **cpu_exec()**.

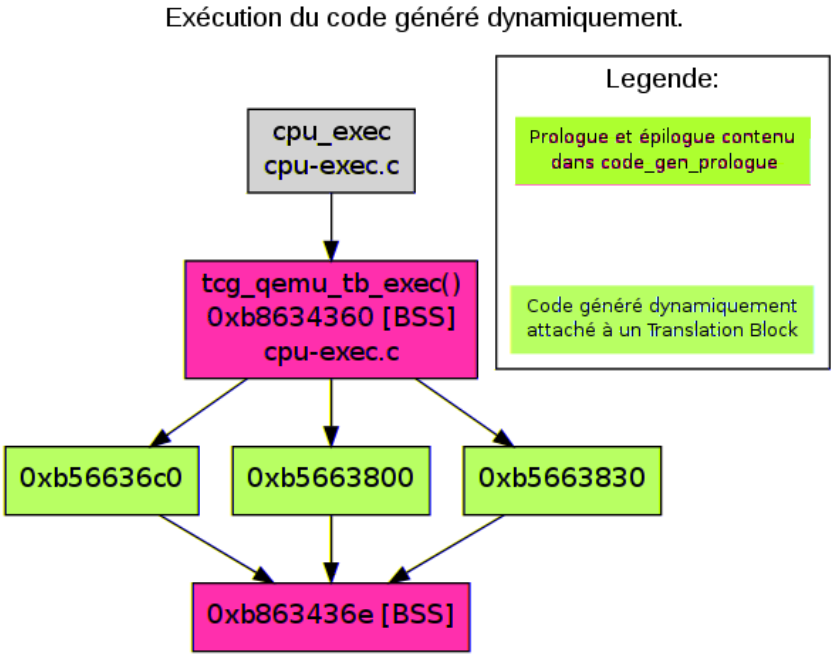

*Fig. 3 : Graphe d'appels simplifié de l'exécution du code hôte généré dynamiquement. Les trois TB aux adresses **0xb56636c0**, **0xb5663800** et **0xb5663830** sont exécutés via le prologue à l'adresse **0xb8634360** correspondant au tableau **code_gen_prologue[]**. Ils se terminent par l'épilogue à l'adresse **0xb863436e**. La mention BSS indique que le code du prologue et de l'épilogue n'est pas situé dans la section **.text**, mais dans la section **.bss**.*

Pour terminer ce petit exposé sur l'émulation de code par recompilation dynamique, reprenons l'exemple de l'instruction ARM de non logique **mvn** dans un contexte plus global. Imaginons qu'un programme C définisse la fonction triviale suivante :

```
int not (int a) {
        return ~a;
}
int main (int argc, char **argv){
        return not(argc);
}
```

Ce programme sera compilé avec une chaîne de compilation croisée GCC pour ARM.

Qemu dispose d'une fonctionnalité de journalisation des différentes actions qu'il mène en adjoignant les options **-d in_asm,op,out_asm** à sa ligne de commande. Le fichier de log par défaut **/tmp/qemu.log** contiendra, pour chaque TB et suivant les options :

– **in_asm** : listage du code invité désassemblé.

– **op** : listage du bytecode TCG pour chaque ligne d'assembleur du code invité.

– **out_asm** : listage du code assembleur hôte généré.

Ces options sont très verbeuses et un boot complet d'un système GNU/Linux ARM avec un login initial génère un fichier de log de plus de 200 Mo contenant plus de 7 millions de lignes. Il faut avoir un peu de flair pour identifier dans le fichier de log les lignes correspondant à l'exécution d'un programme précis. C'est

l'intérêt de l'instruction ARM **mvn** qui est relativement peu utilisée. L'extrait de log présenté, dont de larges portions ont été supprimées, correspond à l'exécution de la fonction **not()** :

```
----------------
IN:
0x00008468:  e52db004    push {fp}         ; (str fp, [sp, #-4]!)
0x0000846c:  e28db000    add  fp, sp, #0    ; 0x0
0x00008470:  e24dd00c    sub  sp, sp, #12   ; 0xc
0x00008474:  e50b0008    str  r0, [fp, #-8]
0x00008478:  e51b3008    ldr  r3, [fp, #-8]
0x0000847c:  e1e03003    mvn  r3, r3
0x00008480:  e1a00003    mov  r0, r3
0x00008484:  e28bd000    add  sp, fp, #0    ; 0x0
0x00008488:  e8bd0800    pop  {fp}
0x0000848c:  e12fff1e    bx   lr

OP:
---- 0x8468
  mov_i32 tmp8,r13
  movi_i32 tmp9,$0xfffffffc
  add_i32 tmp8,tmp8,tmp9
  mov_i32 tmp9,r11
  qemu_st32 tmp9,tmp8,$0x1
  mov_i32 r13,tmp8
...
  ---- 0x847c
 mov_i32 tmp9,r3
 not_i32 tmp9,tmp9
 mov_i32 r3,tmp9
...
  ---- 0x848c
  mov_i32 tmp9,r14
  movi_i32 tmp8,$0xfffffffe
  and_i32 pc,tmp9,tmp8
  movi_i32 tmp8,$0x1
  and_i32 tmp9,tmp9,tmp8
  st_i32 tmp9,env,$0xd0
  exit_tb $0x0

OUT: [size=306]
0xb56636c0:  mov   0x34(%ebp),%ebx
...
0xb5663779:  mov   %eax,%ebx
0xb566377b:  not   %ebx
0xb566377d:  mov   %ebx,%esi
...
0xb56637ed:  jmp   0xb863436e

----------------
```

La première section **IN:** correspond à l'intégralité de la fonction **not()** en assembleur ARM. La première colonne est l'adresse de l'instruction dans l'espace d'adressage virtuel du processeur invité. La deuxième est l'opcode ARM de l'instruction. La suite est le désassemblage de l'instruction. Nous remarquerons, à l'adresse **0x0000847c**, l'instruction **mvn r3, r3** qui a déjà été le sujet de notre intérêt.

La deuxième section **OP:** contient le bytecode TCG. Chaque bloc de bytecode TCG est préfixé de l'adresse de l'instruction assembleur ARM correspondante. Seuls les traductions de la première ligne, de la dernière ligne et de l'instruction **mvn r3, r3** sont présentées. Pour cette dernière, dans la section **0x847c**, nous retrouvons la même traduction que celle qui a été déduite au paragraphe 3.1. Dans la traduction de la dernière instruction ARM **bx lr**, qui constitue le retour de fonction, nous pouvons noter que la dernière instruction de bytecode TCG est **exit_tb $0x0**. Elle correspond à l'opcode **INDEX_op_exit_tb** qui permet le retour du code du TB vers l'épilogue de la fonction de lancement des TB. L'utilisation des variables du TCG, telles que **tmp8** et **tmp9**, dépend de la complexité du code à traduire. Plus le code est complexe, plus il y en aura.

La dernière section **OUT:** contient le code assembleur hôte x86. Sa taille est de 306 octets et 92 instructions x86 pour le TB entier. Seules quelques lignes sont représentées, car l'ensemble du TB en comporte 92. La première colonne fournit l'adresse de l'instruction dans l'espace mémoire du processus hôte Qemu. La première adresse **0xb56636c0** est le début du code hôte du TB. C'est la valeur contenue dans le pointeur **tc_ptr** transférée ensuite dans le registre **%edx** lorsque l'instruction **jmp *%edx** lance le TB. Les lignes

suivantes concernent l'instruction **not_i32** déjà présentée au paragraphe 3.2. D'autres registres du processeur x86 auraient pu être utilisés à la place de **%eax**, **%ebx** et **%esi** suivant leur occupation courante. La dernière instruction du TB est **jmp 0xb863436e** qui correspond à l'adresse de l'épilogue de la fonction de lancement des TB, contenue dans variable globale **tb_ret_addr**. L'exécution de ce TB à l'adresse **0xb56636c0** est illustré sur la figure 3.

4. La gestion du temps sous Qemu

Après avoir disséqué la traduction du code invité vers le code hôte, nous pouvons nous poser une question : Qu'en est t-il du temps d'exécution du code émulé, et comment ce temps sera-t-il perçu par le système invité ? En examinant l'exemple précédent, nous pouvons constater que 10 instructions ARM ont été traduites en 92 instructions x86 auxquelles il faut ajouter les 13 instructions de la fonction de lancement des TB. Cela nous amène à 105 instructions hôte pour 10 instructions invité. Il y a donc fort peu de chance, quel que soit le système hôte, que les temps d'exécution correspondent à ceux d'un système réel.

Mais avant d'aller plus loin, commençons cette partie par quelques précisions terminologiques. Comme le sujet de ce chapitre est le temps voici les termes qui seront utilisés. Le temps du système hôte correspond au temps du système sur lequel s'exécute Qemu. Il concorde avec temps de la machine physique, qui, si elle fonctionne correctement, équivaut approximativement à notre temps humain. Le temps du système émulé ou système invité désigne le temps au sein du système émulé par Qemu, c'est-à-dire le temps « vu » par le système Linux émulé. Ces deux temps sont distincts et peuvent ne pas s'écouler à la même vitesse même si la théorie de la relativité ou une quelconque faille temporelle de Star Trek n'en soient responsables. La plaisanterie facile étant faite, nous insistons sur le fait que le terme « horloge temps réel » n'a rien à voir avec le terme « système temps réel ». Pour éviter toute confusion, rappelons qu'une horloge temps réel (ou RTC pour real time clock) fourni le temps courant avec une référence humaine contrairement à une horloge simple (comme celle qui cadence un processeur) qui bat simplement à intervalles réguliers. Par ailleurs un système temps réel est un système qui a des contraintes temporelles fortes et ce sujet sort complètement du cadre de cet article.

4.1 Les horloges de Qemu

Qemu dispose de trois horloges internes principales. Elles sont définies dans le fichier **qemu-timer.c** et sont parmi les premières entités à être initialisées lors du lancement. En voici la description :

– **host_clock** : Cette horloge fournit le temps du système hôte. Elle utilise la fonction **gettimeofday()** de la bibliothèque C standard. Elle donne le nombre de microsecondes écoulées depuis l'époque Unix (1er janvier 1970 à 00:00:00 (UTC)). Elle est sujette à des ajustements et des sauts dans le passé ou le futur. Elle est utilisée lorsqu'une horloge temps réel est nécessaire.

– **rt_clock** : Malgré son nom peu adéquat, cette horloge ne fournit pas le « temps réel » en accord avec le rappel de sa définitions donnée plus haut. Elle utilise la fonction **clock_gettime()** sur l'horloge **CLOCK_MONOTONIC** . Sous Linux, **CLOCK_MONOTONIC** fournit le nombre de nanosecondes écoulées depuis le boot du système et cela de manière monotone, c'est-à-dire de manière toujours croissante. Elle n'est donc pas sujette aux sauts. Elle ne s'arrête pas pendant l'exécution de Qemu et il l'utilise comme référence pour la partie hôte de l'émulateur. La confusion des noms de variables est donc totale.

– **vm_clock** : Qemu l'utilise comme référence pour le système invité. Son nom « virtual machine clock » est plus signifiant que les autres. C'est l'horloge du processeur virtuel. Comme **rt_clock**, sa résolution est la nanoseconde. Par défaut elle utilise aussi la fonction **clock_gettime()** sur l'horloge **CLOCK_MONOTONIC**, mais elle est mise a zéro lors du démarrage de la machine virtuelle. Elle croit au même rythme que **rt_clock**, mais elle est arrêtée lorsque la machine virtuelle, c'est-à-dire le système invité, l'est aussi. Lorsque l'option **-icount** de Qemu est spécifiée elle utilise le compteur d'instructions du processeur virtuel pour s'incrémenter. Nous détaillerons cet aspect plus loin.

Ces horloges sont des structures **QEMUClock** définie dans **qemu-timer.c**. Elle contient son type (**host_clock**, **rt_clock** ou **vm_clock**) ainsi qu'une liste chaînée de structures **QEMUTimer**. Ces derniers permettent de déclencher une fonction de call back lorsqu'un délai, sur une certaine horloge, est arrivé à expiration.

4.2 Comprendre icount

Lorsque Qemu est lancé sans option de contrôle des horloges, il exécute le code invité à sa vitesse maximale et les horloges **rt_clock** et **vm_clock** progressent au même rythme. Le temps du système hôte et

celui du système invité s'écoule approximativement à la même vitesse, c'est-à-dire celle que nous ressentons humainement. Si les performances de l'émulation sont maximales, cela peut cependant avoir un inconvénient. Imaginons que deux développeurs travaillent sur un projet commun en utilisant Qemu dans une phase de prototypage. S'ils n'ont pas des stations de travail ayant exactement la même puissance, les programmes qu'ils développent ne tourneront pas à la même vitesse dans leurs émulateurs. Comme dans le cas que nous étudierons plus bas, l'un pourra avoir une vision optimiste tandis que pour l'autre l'exécution sera plus lente que sur le système réel. Qemu offre une solution à ce genre de problème.

Il peut être invoqué avec l'option **`-icount N`**. Cette option, dont la signification est « instructions count », permet de cadencer l'horloge **`vm_clock`** du système émulé avec le compteur d'instructions du processeur virtuel plutôt qu'avec **`clock_gettime()`**. L'ingéniosité de cette technique vient du fait que, quel que soit la vitesse du système hôte, **`vm_clock`** progressera manière proportionnelle au nombre d'instructions de code invité exécutées. Autrement dit, un même programme invité exécuté sur deux hôtes différents prendra autant de temps d'un point de vue du système invité. Comment est-ce possible ? Eh bien le temps sur le système invité sera distordu durant l'exécution. Il s'écoulera soit plus vite, soit plus lentement. Cependant, lorsque le système invité est complément inactif et que le processeur virtuel est dans un état de sommeil, **`vm_clock`** ne progresserait plus du tout. Pour palier le glissement temporel dans ce cas, l'horloge **`vm_clock`** est cadencée indirectement par **`rt_clock`**.

Maintenant que le concept l'horloge à cadence variable est introduit entrons un peu dans ses arcanes. L'horloge virtuelle **`vm_clock`** a une résolution d'une nanoseconde, sa fréquence virtuelle est donc de 1 GHz. Cela implique, lorsque le paramètre **`-icount`** n'est pas utilisé, qu'un incrément de **`I`** de **`vm_clock`** correspond à une durée de **`I`** nanosecondes dans le monde réel. Quand Qemu est lancé avec l'option **`-icount N`**, pour chaque instruction du processeur virtuel exécutée, **`vm_clock`** sera incrémentée de **`2^N`**. Ainsi, pour chaque instruction, avec :

– **`-icount 0`**, **`vm_clock`** sera incrémentée de 2^0 = 1 nanoseconde. Le processeur virtuel a une cadence virtuelle de 1 GHz.

– **`-icount 1`**, **`vm_clock`** sera incrémentée de 2^1 = 2 nanosecondes. Le processeur virtuel a une cadence virtuelle de 500 MHz.

– **`-icount 4`**, **`vm_clock`** sera incrémentée de 2^4 = 16 nanosecondes. Le processeur virtuel a une cadence virtuelle de 62.5 MHz.

Cela sera valable quelque soit la vitesse de l'hôte, et uniquement lorsque le processeur virtuel sera actif. La documentation de Qemu [17] n'est pas très claire à ce sujet : « The virtual cpu will execute one instruction every 2^N ns of virtual time. » . Cela laisse penser que le processeur virtuel exécute une instruction à chaque 2^N nanosecondes du temps virtuel, alors que dans la réalité, c'est l'horloge virtuelle qui est incrémentée de 2^N nanosecondes pour l'exécution de chaque instruction du processeur virtuel. Dans la pratique cela revient à la même chose, mais la compréhension du principe n'est pas facilitée par cette formulation.

`vm_clock` est mise à jour via la fonction **`qemu_get_clock_ns()`** de **`qemu-timer.c`**. Lorsque le mode **`-icount`** est activé, cette dernière n'utilise plus la fonction **`cpu_get_clock()`** du fichier **`cpus.c`** basée sur l'heure de l'hôte, mais elle emploie la fonction **`cpu_get_icount()`** utilisant le compteur d'instructions du processeur virtuel :

```
int64_t cpu_get_icount(void)
{
        int64_t icount;
        CPUState *env = cpu_single_env;;
        icount = qemu_icount;
        if (env) {
                if (!can_do_io(env)) {
                        fprintf(stderr, "Bad clock read\n");
                }
                icount -= (env->icount_decr.u16.low + env->icount_extra);
        }
        return qemu_icount_bias + (icount << icount_time_shift);
}
```

`qemu_icount` est le compteur d'instructions du processeur virtuel. Il est mis à jour dans la fonction **`tcg_cpu_exec()`**, déjà présentée, avant chaque boucle d'exécution avec la valeur prévue en fin de boucle. Si la fonction **`cpu_get_icount()`** est appelée avant la fin de la boucle courante d'exécution du processeur virtuel, le nombre d'instructions restantes à exécuter dans cette boucle (contenu dans **`env->icount_decr.u16.low + env->icount_extra`**) lui est soustrait pour obtenir le compteur d'instructions courant. La valeur de **`vm_clock`** retournée comporte deux parties, **`qemu_icount_bias`** et **`(icount << icount_time_shift)`**. **`icount_time_shift`** contient la valeur **`N`** passée en paramètre de l'option **`-icount N`**. **`(icount << icount_time_shift)`**, qui est en fait une multiplication par **`2^icount_time_shift`**, justifie le fait que, comme le dit la documentation, une instruction du processeur virtuel est exécutée pour **`2^N ns`** de

l'horloge virtuelle. L'horloge **vm_clock** est donc fonction de **2^icount_time_shift** fois le nombre d'instructions du processeur virtuel et c'est ce mécanisme qui permet de la synchroniser sur le compteur d'instructions.

Cependant, si seul le compteur d'instructions du processeur virtuel incrémentait **vm_clock**, le temps du système virtuel serait extrêmement ralenti lors de l'inactivité du processeur. C'est la raison d'être de **qemu_icount_bias**. Cette variable globale, définie dans **cpus.c**, n'est pas mise à jour lorsque le processeur virtuel est occupé, mais seulement lorsqu'il est inactif. Cette compensation, ou distorsion, est réalisée dans les fonctions **qemu_clock_warp()** et **icount_warp_rt()** aux noms évocateurs définies dans **cpus.c**. **qemu_icount_bias** est calculé en fonction de **rt_clock**. Il existe un décalage entre **qemu_icount_bias** et **rt_clock** qui correspond au temps d'activité du processeur virtuel en temps sur le système hôte. Ainsi, lorsque le processeur virtuel est inactif, le temps sur le système virtuel s'écoule approximativement à la même vitesse que sur le système hôte. Il en va autrement lorsque le processeur virtuel est actif.

Pour appréhender ce curieux mécanisme, voici un tableau qui compare les temps d'exécution d'un même programme tournant sur un même système invité, sur des systèmes hôte de puissances différentes et avec ou sans l'utilisation de l'option **-icount**. Le programme utilisé pour le test est le suivant :

```
main(){
        int i;
        for (i=0;i<1000000000;i++);
}
```

C'est une simple boucle d'un milliard d'itérations. Il a été, bien évidemment, compilé sans option d'optimisation, sinon il retournerait immédiatement.

Type de processeur hôte et vitesse en BogoMIPS (provenant de **/proc/cpuinfo**)	Machine 1 :Intel(R) Core(TM)2 Quad CPU Q8200 @ 2.33GHz 4662 BogoMIPS (x4)	Machine 2 :Intel(R) Xeon(TM) CPU 1.70GHz 3361 BogoMIPS
Temps écoulé sur système hôte sans **-icount**	21s	39s
Temps écoulé sur le système invité sans **-icount**	0m20.297s (varie un peu d'un lancement à l'autre)	0m38.306s (varie un peu d'un lancement à l'autre)
BogoMIPS invité sans **-icount**	304.74 BogoMIPS	172.85 BogoMIPS
Temps écoulé sur système hôte avec **-icount 0**	22s	68s
Temps écoulé sur le système invité avec **-icount 0**	0m7.003s (toujours fixe)	0m7.003s (toujours fixe)
BogoMIPS invité avec **-icount 0**	999.42 BogoMIPS	999.42 BogoMIPS
Temps écoulé sur système hôte avec **-icount 1**	22s	68s
Temps écoulé sur le système invité avec **-icount 1**	0m14.010s (toujours fixe)	0m14.010s (toujours fixe)
BogoMIPS invité avec **-icount 1**	499.71 BogoMIPS	499.71 BogoMIPS
Temps écoule sur système hôte avec **-icount 4**	24s	74s
Temps écoulé sur le système invité avec **-icount 4**	1m52.567s	1m52.563s
BogoMIPS invité avec **-icount 4**	62.25 BogoMIPS	62.25 BogoMIPS

Au premier abord ce tableau peut sembler nébuleux, mais nous allons analyser chacun de ses résultats. Les mesures de temps sur le système hôte ont été réalisées avec un simple script qui détecte et mesure le temps consommé par le processus Qemu. Sa précision est de l'ordre de la seconde. Il est en effet difficile de mesurer la durée d'exécution d'un processus invité à partir du système hôte, car ce dernier n'a « conscience » que d'un seul processus, celui de l'émulateur. Cependant, cette précision suffira pour notre réflexion. Sur l'invité, la mesure a été beaucoup plus aisée avec une simple commande **time**.

La vitesse des systèmes hôte et invité est mesurée en BogoMIPS [18]. Pour rappel, cette invention de Linus Torvalds permet de mesurer empiriquement la vitesse d'exécution d'un processeur. Cette mesure n'est pas exacte d'où le début de son nom « Bogo » pour bogus, c'est-à-dire faux en français et « MIPS » signifiant millions d'instructions par seconde. Elle est basée sur une boucle de calcul lancée au démarrage du noyau Linux et peut être consultée dans **/proc/cpuinfo**. Les BogoMIPS donnent néanmoins une bonne idée de la

rapidité du processeur.

À titre d'exemple, le système réel utilisé comme support de cet article, une carte Armadeus APF27 affiche 199 BogoMIPS. Le programme de test prend 27.670s à s'exécuter sur ce système. Ce temps peut être pris comme référence pour comparer les performances de l'émulation.

Dans le premier cas où l'option **`-icount`** n'est pas utilisée, le nombre de BogoMIPS du processeur virtuel dépend de celui du processeur hôte. Ainsi, le processeur virtuel tournant sur la machine 1 est plus rapide que celui de la machine 2. Sur une même machine, les temps d'exécution réels (ceux mesurés sur l'hôte) et les temps d'exécution vus depuis le système cible sont similaires. Notons que le processeur virtuel de la machine 1 est plus rapide que sont pendant réel de l'APF27 contrairement au processeur virtuel de la machine 2 qui est plus lent. Le point important à noter est que les temps d'exécution vu du système cible diffèrent entre la machine 1 et la machine 2.

Dans les trois cas suivants, l'option **`-icount`** est activée avec les valeurs 0, 1 et 4. Les deux premiers constats importants sont :

– Le nombre de BogoMIPS du processeur virtuel ne dépend pas de celui de l'hôte mais uniquement de la valeur du paramètre **`-icount`**. La vitesse du processeur virtuel est donc la même sur la machine 1 et sur la machine 2 pour une même valeur de **`-icount`**.

– Il en découle que le temps d'exécution vu du système invité est le même sur les deux machines hôte. Ce temps lui aussi ne dépend que de la valeur de **`-icount`**.

Ensuite, le nombre de BogoMIPS est quasiment égal à la fréquence de **`vm_clock`**, c'est-à-dire:

– 999.42 BogoMIPS pour **`vm_clock`** à 1 GHz (-**`icount`** 0).

– 499.71 BogoMIPS pour **`vm_clock`** à 500 MHz (-**`icount`** 1).

– 62.25 BogoMIPS pour **`vm_clock`** à 62.5 GHz (-**`icount`** 4).

Ce petit écart provient de l'erreur induite par la boucle de calcul des BogoMIPS.

Le temps d'exécution réel augmente un peu sur la machine 1, mais il double presque sur la machine 2. Cela laisse penser que la charge de traitements supplémentaires induite par **`-icount`** n'est pas négligeable sur un système hôte peu puissant.

Le fait le plus « amusant » est certainement la distorsion temporelle, c'est-à-dire les différences de vitesses d'écoulement du temps sur le système invité :

– Lorsque **`-icount`** vaut 0 ou 1, le nombre de BogoMIPS du processeur virtuel est supérieur à celui qu'il aurait « naturellement », c'est-à-dire sans l'option **`-icount`**. Le processeur hôte ne peut pas émuler plus rapidement le processeur virtuel que ce nombre de BogoMIPS « naturel ». Il résulte que le temps du système invité s'écoule moins rapidement que le temps sur le système hôte. 7 secondes sur l'invité s'écoulent en 22 secondes réelles pour **`-icount 0`** sur la machine 1.

– Par contre, pour **`-icount 4`**, le nombre de BogoMIPS du processeur virtuel est inférieur à sa valeur « naturelle ». Comme il n'existe aucune temporisation dans l'émulation que fait Qemu, le processeur hôte traite les instructions du système invité plus vite que ne le ferait un système d'une telle puissance. Le résultat est une accélération de l'écoulement du temps sur le système invité où 1 minute 53 secondes s'écoulent en 24 secondes sur la machine 1.

Il est possible de s'en rendre compte visuellement en lançant **`while [ 1 ]; do date;sleep 1;done`** dans un shell sur le système invité lorsque le processeur virtuel est chargé. Le **`sleep 1`**, qui dure une seconde réelle dans notre monde réel, prendra, sur une machine moderne, soit plus d'une seconde avec **`-icount 0`**, soit moins d'une seconde avec **`-icount 4`**.

Pour finir, le paramètre **`-icount`** peut être renseigné avec **`auto`**. Dans ce cas Qemu va ajuster la valeur d'incrémentation de **`vm_clock`** pour suivre au plus prêt le temps du système hôte. Malheureusement, certains pilotes de périphérique Linux n'apprécient pas que le temps système varie lors de leurs phases d'initialisation. C'est notamment le cas du pilote de port série de l'i.MX27 utilisé sur la carte Armadeus APF27. Celui-ci ne rend jamais la main lors de son initialisation si le paramètre **`-icount auto`** est activé.

5. Conclusion

Nous voici arrivés à la fin de notre exploration du cœur de Qemu. J'espère n'avoir pas été trop indigeste dans mon exposé. Qemu est effectivement un logiciel complexe et sa compréhension nécessite quelques acrobaties mentales. La base est dévoilée, mais il reste encore beaucoup d'autres parties de l'émulateur à découvrir. Après cette petite introduction théorique, passons maintenant à la pratique : Comment implémenter un nouveau système invité dans Qemu ? Je vous retrouve donc le mois prochain, en seconde partie, pour vous présenter la mise en œuvre d'une carte Armadeus APF27 émulée avec Qemu.

Références

[1] Pierre Ficheux, « Introduction à BuildRoot », *GNU Linux Magazine France* Hors-série n°47 (mai 2010)

[2] Pierre Ficheux, « Mise au point à distance avec GDB et QEMU », *OpenSilicium* n°1 (janvier 2011)

[3] Pierre Ficheux, *Linux embarqué*, 3e édition, Éditions Eyrolles (2010)

[4] http://wiki.qemu.org/Documentation/GettingStartedDevelopers#Getting_to_know_the_code

[5] http://www.ioccc.org/winners.html#B

[6] Documentaire Nom de code Linux, minute 24, http://video.google.fr/videoplay?docid=-3699763257121592701

[7] Cscope, http://cscope.sourceforge.net/

[8] Eclipse C/C++ Indexer, http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.user/concepts/cdt_c_indexer.htm

[9] http://valgrind.org/

[10] http://kcachegrind.sourceforge.net/cgi-bin/show.cgi/KcacheGrindIndex

[11] http://www.graphviz.org/

[12] Fabrice Bellard, « QEMU, a Fast and Portable Dynamic Translator », http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard_html/index.html

[13] ARM Architecture Reference Manual, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0100i/index.html

[14] Intel® 64 and IA-32 Architectures Software Developer's Manual, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[15] GCC, Declaring Attributes of Functions, http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

[16] Linux Assembly HOWTO, Calling conventions, http://asm.sourceforge.net/howto/conventions.html

[17] QEMU Emulator User Documentation, http://qemu.weilnetz.de/qemu-doc.html#sec_005finvocation

[18] The FAQ about BogoMIPS, http://tldp.org/HOWTO/BogoMips/bogo-faq.html