Machines virtuelles

Brique ASC

Samuel Tardieu
sam@rfc1149.net

École Nationale Supérieure des Télécommunications

Machines virtuelles

La compilation peut avoir des désavantages :

- Le résultat n'est en général pas portable
- Le résultat ne suit pas les évolutions des architectures :
  - Un programme compilé pour un Pentium 4 ne tournera pas sur un 386 (instructions manquantes)
  - Un programme compilé pour un 386 ne sera pas optimisé pour Pentium 4 (pas d'instructions MMX, ordonnancement)

Principes

Une machine virtuelle :

- possède un jeu d'instructions virtuel, prédéterminé ou dynamique
- est émulée sur une ou plusieurs plates-formes (par le biais d'un moteur d'exécution)
- peut posséder des instructions simples ou complexes

Quelques machines virtuelles

- Forth
- Pascal UCSD
- NeWS (serveur X11 PostScript)
- Python, Objective CAML, Erlang
- Parrot (pour Perl 6)
- Java, .NET

Processus de compilation

Programme source

Bytecode

Processus de compilation

Programme source

compilation

Bytecode → Machine virtuelle

Processus de compilation

Programme source

compilation

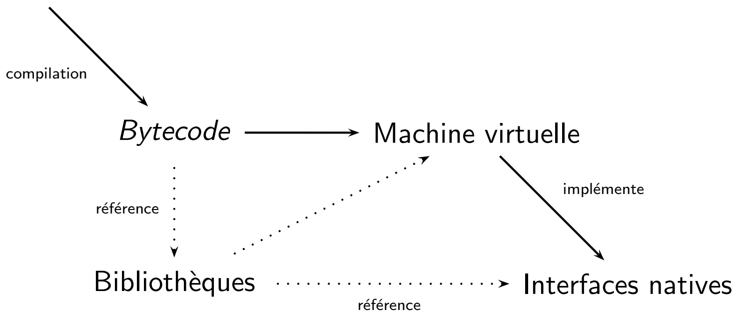
Bytecode

Machine virtuelle

référence

Bibliothèques

Processus de compilation



Langages à pile

Les langages à pile :

- peuvent être destinés à l'humain (exemple : Forth) ou à la machine
- sont plus efficaces lors d'un appel
- permettent une représentation compacte
- sont utilisés dans certaines machines virtuelles (exemple : Java)

Exemple Java

Le code suivant :

```
static int t (int x, int y, int z) {
   return x+y*z;
}
```

se compile en

```
   0:iload_0     // push x
   1:iload_1     // push y
   2:iload_2     // push z
   3:imul        // y*z
   4:iadd        // x+y*z
   5:ireturn     // return result
```

Remarques

- La pile n'est utilisée que pour les constructions internes
- Entre les méthodes, une pile de retour (arguments et variables locales, comme en C) est utilisée
- La pile de données est vide à l'entrée d'une méthode
- Ces piles peuvent être implémentées avec une paire "frame pointer"/"stack pointer"

Interprétation virtuelle

Il existe plusieurs techniques d'interprétation, dont :

- *direct threaded code*
- *indirect threaded code*
- *token threaded code*
- *subroutine threaded code*
- compilation au vol

Direct threaded code

- Un pointeur d'instruction (IP) pointe à l'endroit contenant la prochaine instruction à exécuter
- À la fin de chaque mot, `next` est exécuté :
  - Le pointeur d'instruction est incrémenté
  - On saute à la valeur précédente du pointeur d'instruction
- L'interpréteur peut être rappelé pour un mot non natif

Coeur du Direct TC

```
; next permet de sauter au mot suivant
next:
        movl ip,%eax
        incl ip
        jmp *%eax

; Exemple: execution de a b c
.section ".data"
abc:    .long a b c
ip:     .long abc
```

Interpréteur Direct TC imbriqué

```
; sauvegarde de ip et du sommet de la pile de retour
do_list:
        movl (%esp),%eax
        movl ip,%edx
        movl %edx,(%esp)    ; Sauvegarde ancien IP
        movl %eax,ip        ; Installation nouveau IP
        jmp next
; restauration de ip depuis la pile
exit:
        popl %eax
        movl %eax,ip
        jmp next
```

Exemple de Direct TC

Les mots Forth

```
: square dup * ;
: cube dup square * ;
```

se traduit par

```
square:
        call do_list
        .long dup,times,exit
cube:
        call do_list
        .long dup,square,times,exit
```

Code natif et Direct TC

Voici des implémentations natives de `dup` et `times` (la pile Forth est indexée par `%ebx`) :

```
dup:    movl (%ebx),%eax
        addl -4,%ebx
        movl %eax,(%ebx)
        jmp next

times:  movl (%ebx),%eax
        addl 4,%ebx
        imull (%ebx),%eax
        movl %eax,(%ebx)
        jmp  next
```

Caractéristiques du Direct TC

- Tous les mots sont directement exécutables, ce qui facilite l'interface avec d'autres langages
- Cela coûte un saut au début de chaque mot
- Utilisation non-optimale des caches par le mélange de code et de données

Toutefois, le *direct threaded code* est un des moyens d'exécution les plus performants et les plus simples à implémenter.

Indirect Threaded Code

L'*indirect threaded code* ressemble au direct threaded code avec un niveau d'indirection supplémentaire : l'adresse où sauter est stockée plutôt que le saut lui-même.
Exemple :

```
square:
        .long do_list,dup,times,exit
```

Les mots ne sont pas exécutables, mais le code *peut* être plus compact.

ITC : boucle d'interprétation

La boucle d'interprétation contient un niveau d'indirection supplémentaire :

```
next:
        movl ip,%eax
        incl ip
        movl (%eax),%eax  ; <--- ici
        jmp *%eax
```

Token threaded code

- Plutôt que des adresses, on utilise des identificateurs (*tokens*).
- Une table est utilisée pour associer chaque *token* à l'adresse à exécuter.
- Si ces identificateurs sont fixés *a priori* :
  - on obtient du code portable ;
  - il est possible de faire un "gros switch".

Boucle d'interprétation

```
void next (unsigned int *ip) {
  for (;;) {
    switch (*ip++) {
       case OP_PLUS:
         PUSH (POP() + POP());
         break;
       case METHOD_CALL:
         next (FIND_METHOD (*ip++));
         break;
       case EXIT:
         return;
       ...
```

Subroutine threaded code

Dans le *subroutine threaded code*, des appels sont insérés directement, c'est la voie vers la compilation native :

```
square:
        call dup
        call times
        return
```

La première optimisation est immédiate

```
square:
        call dup
        jmp times
```

Subroutine threaded code

Dans le *subroutine threaded code*, des appels sont insérés directement, c'est la voie vers la compilation native :

```
square:
        call dup
        call times
        return
```

La première optimisation est immédiate

```
square:
        call dup
        jmp times
```

STC et compilation native

La compilation native est similaire au *subroutine threaded code*, sauf qu'il est possible de remplacer des appels par du code *inline*, comme ici :

```
: foo drop bar ;
```

devient

```
foo:
        addl 4,%ebp   \ Drop
        jmp bar
```

Compilation au vol

La compilation au vol (Just In Time, ou JIT) :

- transforme du bytecode en code natif au chargement ou à la première exécution ;
- peut produire plusieurs alternatives natives d'un même code pour s'accomoder d'architectures différentes ;
- a un surcoût fixe important ;
- peut être plus efficace en « régime de croisière » qu'un interpréteur.

Exemple de la machine Java

Le code Java pour la JVM (Java Virtual Machine) :

- est *token threaded* (avec des opcodes fixés à l'avance) ;
- peut être compilés au vol ;
- a des *tokens* (opcodes) de très haut niveau (appel de méthode, exceptions, etc.)
- fait appel à un grand nombre de bibliothèques externes.

JVM et langages

D'autres langages que Java ciblent la machine virtuelle Java :

- Ada 95
- Python
- Forth
- Scheme, Basic, Logo, ...

Vérifications

Au moment du chargement (en général dynamique), la JVM vérifie :

- la bonne construction du fichier d'entrée
- l'absence de débordement de pile (chaque méthode indique la place dont elle a besoin)

Sur certaines JVM, il est possible de précharger tous les modules utilisés potentiellement par une application Java, pour éviter une mauvaise surprise à l'exécution.

Précompilation portable

Il est possible de partager du code machine s'exécutant, sans compilation au vol, à la vitesse du code natif.

- Il suffit d'inclure les deux (bytecode et code natif pour une ou plusieurs plates-formes) dans la même distribution.
- Au moment de l'exécution, choisir soit le code natif (plus rapide) s'il est disponible, soit le code à interpréter (plus facile à déboguer).

Le compilateur du langage Erlang et sa machine virtuelle ont adopté cette pratique.

Interface avec le système

- Tout ne peut pas se faire en byte code.
- La machine virtuelle s'interface avec le système.
- En général, la VM permet d'appeler le système directement (appels systèmes ou appels de bibliothèques).
- Écrire des parties en code natif permet d'accélérer certaines routines critiques.

Java vs .NET

- Java :
  - est contrôlé par Sun Microsystems ;
  - offre un bytecode portable ;
  - n'offre qu'une interface objet
- .NET :
  - est contrôlé par Microsoft ;
  - offre un bytecode portable ;
  - permet d'accéder aux données des autres objets directement.

Émulation et VM

Certaines machines virtuelles émulent de vraies architecture :

- émulateur 68LC040 d'Apple pour ses systèmes à base de PowerPC
- émulateur DragonBall de Palm Inc pour ses assistants personnels à base d'ARM
- émulateur PlayStation pour Unix