Rapport
Projet de compilation avancée

Développement d'un interpréteur pour le langage UM de la machine universelle
Développement d'un compilateur du langage S-UM vers le langage UM

Amel ARKOUB 3301571
Ling-Chun SO 3414546

04 avril 2018

Sommaire

0

Introduction

0.1 Présentation

Le but de ce projet est de fournir dans un premier temps, une implantation de la machine universelle (`www.boundvariable.org/um-spec.txt`) issue d'un concours de programmation, *ACM International Conference on Functional Programming* (ICFP) de 2006 (`www.boundvariable.org/task.shtml`). Nous souhaiterions aussi pouvoir programmer dans le langage compris par la machine universelle, cependant celui-ci est un langage binaire, ce qui n'est pas chose aisée. Ainsi dans un second temps nous allons écrire un compilateur qui compile le langage S-UM *Specification of Universal Machine* vers le langage binaire UM.

0.2 Description des fichiers

Voici les différents dossiers et ce qu'ils contiennent:

- ./um/ → Ce dossier contient l'implantation de la machine universelle.
- ./sum/ → Ce dossier contient l'implantation du compilateur S-UM.
- ./tests/ → Ce dossier contient les différents tests du compilateur S-UM.
- ./rapport.pdf → Ce fichier correspond à ce rapport.
- ./README.md → Ce fichier contient les instructions pour compiler et tester le projet.
- ./rapport/ → Ce dossier contient les sources de ce rapport en LateX.
- ./archives/ → Ce dossier contient essentiellement des fichiers à ne pas considérer, plus spécifiquement, se trouve un début d'analyseur/parseur en C pour le langage S-UM mais aussi une implantation de la machine universelle à partir de liste cependant pour des raisons de performance elle à été délaissé pour une implantation plus performante.

0.3 Etat du projet

Le projet dans l'ensemble terminé cependant certains points sont à préciser:

- la machine universelle (UM): Fonctionnel
- le compilateur S-UM: Fonctionnel
  Tous les traits du langage sont supportés cependant il est important de préciser:

- séquence d'instructions: Fonctionnel
- affectation de variable: Fonctionnel
- print: Améliorable
  En effet, le print fonctionne pour les chaînes de caractères et ainsi que les constantes, cependant elle n'affiche que l'évaluation modulo 256. Par exemple, *print 96+1* affichera en sortie "a", 97 étant la lettre "a" en ASCII.
- scan: Améliorable
  Le scan ne prend qu'un caractère ASCII.
- alternative: Fonctionnel
- entier: Fonctionnel
- chaine de caractère: Fonctionnel
- expression arithmétique: Fonctionnel
- expression relationnelles: Fonctionnel
- expression logiques binaires: Fonctionnel
- expression logiques unaires: Fonctionnel

1

Machine Universelle

1.1 Structure

Au niveau de l'implantation de la machine universelle, étant donné que chaque *platter* est codé sur 32 bits, nous utilisons un entier non signé 32 bits:

```
typedef uint32_t uint32;
```

Nous avons choisit comme structure pour contenir les instructions du programme celle-ci:

```
typedef struct array{
  uint32 size;
  uint32 *platter;
} array;
```

C'est une structure permettant de contenir un tableau d'instructions et la taille en mémoire (la taille est nécessaire lors de l'instruction LOAD PROGRAM).

Afin de pouvoir garder en mémoire tout les indices réutilisable nous avons une structure de liste pour les indices, dans le cas où la liste est vide on retourne une variable globale et on l'incrémente.

```
extern uint32 indexcpt;
typedef struct freeindex{
 uint32 index;
 struct freeindex *next;
} freeindex;
```

1.2 Fonctions

Voici la description des fonctions de la machine universelle:

- array* loadFile(const char* filename) → Renvoie un tableau contenant toutes les instructions lues dans le fichier *filename* en binaire, il faut ensuite inversé l'endianess (le lancement de la machine virtuelle avec le fichier *ressources/sandmarkz.umz* indique si l'endianess est incorrect par l'affichage sur le shell *endianess*).
- freeindex* initFreeIndex() → Initialise la file d'indice utilisable.
- void addFreeIndex(freeindex** fi, uint32 index) → Ajoute dans la file d'indice fi, l'indice index.

- uint32 getFreeIndex(freeindex** fi) → Récupère un indice utilisable, si la file est libre on retourne indexcpt++.
- void freeFreeIndex(freeindex** fi) → Désalloue la structure de file.
- array* initArray(uint32 size) → Alloue la structure array de taill size.
- void freeArray(array *arr) → Désalloue la structure array.

1.3 Interprétation du langage

Le canevas de l'interpretation du langage est le suivant:

```
uint32 registers[8] = {0};
while(1){
   word = zero[pt];
   op = word>>28;
   a = ((word>>6) & 0x7);
   b = ((word>>3) & 0x7);
   c = (word & 0x7);
   switch(op){
     case ..:
     case ..:
     ...
   }
   pt++;
}
```

On initialise un tableau de 8 entiers non signés qui correspondent aux registrers. Le coeur de l'interpretation correspond à un switch englobé dans un while(1) et nous récupérons l'opération, l'indice des registres par des décalage de bit de l'instruction 32 bits puis on incrémente le compteur pt.

1.4 Performance

Les performances d'un interpreteur n'est pas à négliger, du fait de notre implantation C par tableau compilé en -O3 nous avons des performances très satisfaisantes. En prenant le fichier sandmarkz.umz, nous obtenons un temps de 18.424 secondes alors qu'une implantation JAVA peut aller jusqu'à plusieurs minutes pour finir l'éxécution du programme.

1.5 Implantation de la machine universelle par liste

NOTE: Nous allons discuter de la première implantation de la machine universelle à partir de liste, celle-ci bien que fonctionnel est très lente et nous allons étudier ses performances. Il est donc à noter que cette implantation ci n'est pas à retenir pour une utilisation mais plutôt pour une analyse. Cependant si vous souhaitez voir le code, il se trouve dans le dossier ./archives/Universal_Machine_list.

La première version de la machine universelle se repose sur une structure de liste, bien que cette implantation fonctionne, elle possède le désavantage d'être très lente. Le fichier sandmarkz.umz ne se termine toujours pas après 10 heures d'éxécution. Ainsi afin de déterminé le

goulot d'étranglement du programme, nous avons utilisé le logiciel de profilage de code Gprof. Cet outil permet de récupérer les statistiques sur le temps et le nombre d'appel de fonctions dans une éxécution du programme. Afin d'avoir un fichier en sorti nommé *gmon.out*, il faut au préalable désactivé l'optimisation à la compilation et compiler avec le flag *-pg*. Cependant, il faut que le programme termine correctement pour que ce fichier soit généré, nous avons donc décidé de bind le signal *SIGUSR1* comme ceci:

```
#include <signal.h>
...
void sig_exit(){
 exit(0);
}
...
int main(int argc, char **argv){
 signal(SIGUSR1, sig_exit);
 ...
}
```

Une fois le fichier *gmon.out* généré (nous avons lancé le signal *SIGUSR1* à la fin du sandmark 100, il suffit d'appliquer la commande:

```
gprof universal_machine gmon.out > analyse.txt
```

Nous obtenons dans le fichier *analyse.txt*:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 99.52    146.74   146.74  5307119    27.65    27.65  getArray
  0.58    147.60     0.86                             main
  0.03    147.65     0.05    44683     1.01     1.01  removeArray
  0.01    147.66     0.01    87030     0.12     0.12  addArray
  0.00    147.66     0.00    87030     0.00     0.00  getFreeIndex
  0.00    147.66     0.00    44683     0.00     0.00  addFreeIndex
  0.00    147.66     0.00        1     0.00     0.00  initArrays
  0.00    147.66     0.00        1     0.00     0.00  initFreeIndex
  0.00    147.66     0.00        1     0.00     0.00  loadFile
```

Ainsi, sur les 147.66 secondes d'éxécutions, près de 99.52% du temps de calcul est utilisé pour éffectuer la fonction getArray. Nous atteignons aussi un nombre très élevés d'appel 5307119 pour un temps relativement cours. Les problèmes de performances étaient donc dû aux accès de tableaux, en raison de ce nombre important nous avons décidé d'utiliser une implantation de tableaux de tableaux plutôt qu'une implantation de liste de tableaux.

2

Compilateur S-UM