Arquitectura de Computadores I - Módulo 8

Medição do Desempenho

1. Introdução

Pretende-se com esta aula prática que os alunos se familiarizem com o processo de *profiling* de programas e que avaliem o impacto no desempenho de um programa de algumas optimizações disponibilizadas pelo compilador.

2. Profiling

Os *profilers* são ferramentas de instrumentação e análise que recolhem informação sobre o desempenho do programa enquanto este executa. Este processo implica executar uma versão instrumentada do programa que determina:

- o tempo de execução de cada componente do programa (e.g. quanto tempo foi passado a executar uma determinada função);
- quantas vezes foi executada cada componente de um programa (e.g. quantas vezes foi invocada determinada função, ou foi executada determinada instrução).

Os sistemas UNIX fornecem um *profiler*, `gprof`, que gera precisamente estes dois tipos de informação:

- o tempo de execução de cada função é calculado baseado em contagem de intervalos, isto é, sempre que o processo é interrompido para mudança de contexto o tempo correspondente à *time slice* (entre 1 a 10 ms) é integralmente atribuído á função que estiver a executar no momento da interrupção; este método gera apenas uma aproximação, produzindo resultados mais fiáveis quando a duração do programa é bastante superior à duração da *time slice*;
- para cada função é calculado o número de vezes que esta é invocada.

Para utilizar o `gprof` o programa deve ser compilado com a opção `-pg`. Quando executado o programa grava a informação de *profile* no ficheiro `gmon.out`. O *profiler* lê esta informação e apresenta-a em dois formatos:

- **perfil plano** – para cada função é apresentada a percentagem do tempo de execução acumulado, o tempo de execução acumulado, o tempo de execução associado à função e o número de vezes que foi invocada sem contar com invocações recursivas;
- **grafo de invocações** – para cada função é apresentado o seu tempo de execução, o número de vezes que foi invocada por outras funções (incluindo invocações recursivas) e o número de funções que a própria invocou;

Considere o seguinte programa, escrito em C:

progf.c

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1000000
int a[SIZE];

int factorial (int n)
{
  if (n>0)
    return (n*factorial(n-1));
  else
    return (1);
}

int funcao (void)
{
  int sum, i;

  sum=0;
  for (i=1 ; i<SIZE ; i++)
  {
    sum += a[i];
    a[i] = factorial (a[i]);
  }
  return (sum);
}
```

```
main ()
{
  int i, res;

  for (i=0 ; i< SIZE ; i++)
    a[i] = (int) (100.0*rand ()/(RAND_MAX+1.0));

  res = funcao ();

  printf ("Resultado =%d\n",res);
}
```

Compile-o usando os comandos

```
gcc –O0 -pg progf.c –o progf.O0
gcc –O1 -pg progf.c –o progf.O1
```

Execute `progf.O0` e execute o comando

```
gprof –b progf.O0
```

Questão 1 – Qual o tempo de execução e percentagem do total de cada uma das funções? De acordo com a Lei de Amdahl em que função deve o programador concentrar os esforços de optimização?

Execute `progf.O1` e execute o comando

```
gprof –b progf.O1
```

Questão 2 – Qual o tempo de execução e percentagem do total de cada uma das funções? Qual o ganho relativamente à versão não optimizada?

Recodifique a função `factorial ()`, usando um algoritmo iterativo, conforme apresentado na seguinte tabela:

progf.c

```
int factorial (int n)
{
  int res=1;

  for ( ; n>1 ; n--) res *= n;
  return (res);
}
```

Compile, execute e faça o *profiling* da nova versão do programa usando os comandos

```
gcc –O1 -pg progf.c –o progf.O1
./progf.O1
gprof –b progf.O1
```

Questão 3 – Qual o tempo de execução e percentagem do total de cada uma das funções? Qual o ganho relativamente à versão da questão 2?

3. Medição do Desempenho

Medição do número de ciclos

A arquitectura IA32, dispõe, a partir do lançamento do Pentium, de um contador de 64 bits, que é incrementado em cada ciclo do relógio. O contador é iniciado a 0 quando é feito o *reset* do processador, sendo depois consecutivamente incrementado. Quando o maior valor possível de representar é atingido (64 *bits* a 1) o contador volta a 0. Num contador de 64 bits e com um relógio de 1 GHz, isto acontecerá de 570 em 570 anos.

Este contador pode ser lido pela instrução "*RDTSC – Read Time Stamp Counter*", que coloca os 32 bits mais significativos no registo EDX e os 32 bits menos significativos no registo EAX. Se forem ignorados os 32 bits mais significativos, é possível trabalhar com um contador de 32 bits apenas, tendo no entanto em consideração que o contador dará a volta (*wrap around*), isto é, volta a 0, de 4,3 em 4,3 segundos, numa máquina com um relógio de 1 GHz.

O número de ciclos decorridos durante a execução de um programa pode ser calculado lendo o contador no início (T1) e no fim (T2) da execução do mesmo. A diferença T2-T1 é igual ao número de ciclos, devendo-se verificar se T2>T1 e que o programa não demorou mais do que o tempo de *wrap around*.

O número de ciclos medido será sempre uma aproximação, pois o programa pode ser interrompido e podem ocorrer mudanças de contexto. Para que a leitura seja o mais precisa possível, mediremos apenas programas cujo tempo de execução seja muito inferior ao tempo de *wrap around* do contador de 32 *bits* e inferior à *time slice* dos processos no LINUX. Tentaremos portanto não ultrapassar os 5 milisegundos, minimizando também a influência das rotinas de atendimento de interrupção. Para mais detalhes sobre temporização de processos e utilização dos 64 bits do contador ver secções 9.3 e 9.4 do livro "Computer Systems: A Programmer's Perspective"; Bryant and Hallaron.

Variações nas medições

As medições obtidas para diferentes execuções do mesmo programa podem variar bastante, devido à influência da hierarquia de memória, interrupções e outros componentes da máquina. O valor nunca será, no entanto, menor do que o número de ciclos mínimo necessário para executar o programa em circunstâncias ideias: número mínimo de *cache misses*, não ocorrência de interrupções, etc.

Sugere-se que, para cada versão do programa, este seja executado várias vezes e o valor mínimo reportado seja o considerado. Para mais detalhes e métodos de temporização mais sofisticados, ver a bibliografia atrás referida.

Cálculo do CPE (ciclos por elemento)

Para caracterizar o desempenho de determinada função, usaremos como métrica o número de ciclos por elemento a processar (CPE). Neste caso concreto, calcularemos o número de ciclos necessário para somar cada elemento inteiro de um *array*. Para calcular o **CPE** é necessário saber o número de ciclos necessários para executar a função e o número de elementos do *array*. Este último está codificado no próprio programa em C.

O número de ciclos pode ser lido usando a instrução `rdtsc`. O `gcc` disponibiliza uma directiva que permite incluir *assembly* no código em C. Para que o compilador respeite as variáveis a utilizar e não escreva valores por cima de registos que estejam a ser usados para outros cálculos, esta directiva especifica:

- A lista de instruções *assembly*; as referências a registos devem utilizar `%%` em vez de `%`, a referência a parâmetros de entrada/saída é feita usando `%` seguido do número de ordem da sua ocorrência nas lista de *input/output*;
- A lista de parâmetros de saída, isto é, endereços de memória (variáveis) que serão escritas pelas instruções em *assembly*; variáveis inteiras são indicadas usando o símbolo “=r”, seguido do nome da variável dentro de parêntesis;
- A lista de parâmetros de entrada, isto é, endereços de memória (variáveis) que serão lidas pelas instruções em *assembly*; variáveis inteiras são indicadas usando o símbolo “r”, seguido do nome da variável dentro de parêntesis;
- A lista de registos que são utilizados pelas instruções em *assembly*.

Para mais pormenores consulte o código abaixo e a secção 3.15 e 9.3.1 do livro “Computer Systems: A Programmer’s Perspective”; Bryant and Hallaron.

Uma vez que os programas instrumentados usando esta técnica devem ter um tempo de execução bastante pequeno, o programa da secção anterior foi reescrito, retirando a função `factorial()` e reduzindo o número de elementos do *array*, conforme apresentado na seguinte tabela:

prog.c

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1000
int a[SIZE];
int c;

int funcao (void)
{
  int sum, i;

  sum=0;
  for (i=1 ; i<SIZE ; i++)
  {
    sum += a[i];
  }
  return (sum);
}
```

```
main ()
{
  int i, res;
  double CPE;

  for (i=0 ; i< SIZE ; i++)
    a[i] = (int) (100.0*rand ()/(RAND_MAX+1.0));

  asm ("rdtsc; movl %%eax, %0"
       : "=r" (c)   /* Output */
       : /* no inputs */
       : "%eax","%edx"   /* used registers */
    );
  res = funcao ();
  asm ("rdtsc; subl %1, %%eax; movl %%eax, %0"
       : "=r" (c)   /* Output */
       : "r" (c)    /* Inputs */
       : "%eax","%edx"   /* used registers */
    );

  CPE = ((double) c)/SIZE;
  printf ("Resultado =%d\n ",res);
  printf ("Ciclos=%d; CPE=%lf\n", c, CPE);
}
```

Compile-o usando os comandos

gcc –O0 prog.c –o prog.O0
gcc –O1 prog.c –o prog.O1
gcc –O1 –funroll-all-loops prog.c –o prog.loop

Execute várias vezes cada uma das versões do programa e anote os valores mínimos de CPE.

Questão 4 – Que conclui relativamente à melhoria de desempenho conseguida com cada uma das optimizações?

Questão 5 – Analisando o código *assembly* de `função()` das versões sem optimização (`gcc –O0 –S prog.c`) e com o nível de optimização 1 (`gcc –O1 –S prog.c`), quais as optimizações introduzidas pelo compilador?

Questão 6 – Analisando o código *assembly* de `função()` da versão com *loop unrolling* (`gcc –O1 –funroll-all-loops –S prog.c`) consegue explicar a optimização introduzida pelo compilador?