Arquitectura de Computadores II

L.E.S.I. – 3º ano

Módulo nº 5

Avaliação e Optimização do Desempenho

Técnicas independentes da máquina

1. Introdução

No final deste módulo os alunos deverão ser capazes de:

- interpretar medições do desempenho baseadas em número de ciclos por elemento – CPE;
- descrever e aplicar a técnica dos k-melhores resultados para obtenção de métricas fiáveis;
- avaliar o desempenho de um programa;
- decidir quais as técnicas de optimização independentes da máquina a aplicar a um algoritmo genérico;

2. Material de Apoio

A bibliografia relevante para este módulo são as secções 5.1 a 5.6, 9.3 e 9.4 do livro "Computer Systems: a Programmer's Perspective", de Randal E. Bryant e David O'Hallaron.

Para realizar os exercícios propostos abaixo deve descarregar o código disponibilizado na página da disciplina em http://gec.di.uminho.pt/lesi/ac2/praticas/ac2Mod5.tgz

3. Medição e Avaliação do Desempenho

Para avaliar o desempenho de um programa numa determinada máquina é necessário medi-lo. Para o medir exige-se uma métrica e uma metodologia para obter valores para esta métrica.

➢ Métrica

Uma métrica apropriada para programas que efectuam computações repetitivas sobre vários elementos de dados é o número de ciclos por elemento (**CPE**). Do ponto de vista de um programador é mais elucidativo exprimir o desempenho em termos de ciclos do relógio do que em termos de tempo. Apesar desta métrica não ser independente nem da arquitectura nem da organização do processador, é relativamente independente da frequência do relógio e transmite uma ideia mais clara de como o programa estará ser executado pelo sistema.

➢ Medição dos ciclos

Para obter o CPE para um determinado algoritmo e conjunto de dados é necessário medir o número de ciclos do processador gastos no processamento de todos os elementos de dados. Este valor é depois dividido pelo número de elementos, resultando no CPE.

A arquitectura IA32 disponibiliza uma instrução (`rdtsc`) que devolve o contador de ciclos do processador como uma quantidade inteira de 64 bits nos registos `%edx` e `%eax`. Obtendo os valores deste contador antes e após a sequência de código que se pretende medir, podemos calcular o número de ciclos decorridos calculando a diferença entre os dois valores.

O código disponibilizado em clock.c exporta 2 funções que permitem realizar estas operações:

`void start_counter (void)` – deve ser invocada antes da sequência de código a medir;

`double get_counter (void)` – deve ser invocada após a sequência de código a medir e devolve o número de ciclos decorridos desde a invocação da função anterior.

➢ Problemas associados à medição do tempo de execução

A medição do número de ciclos do processador sofre de vários problemas.

Todos os ciclos do relógio decorridos entre a invocação das 2 funções são contados. Numa máquina multi-processo isto significa que o tempo decorrido a executar outros processos e os tempos associados a comutações de contexto também são incluídos. O intervalo de tempo a medir não deve portanto ser muito longo para evitar este fenómeno – preferencialmente deverá ser inferior à fatia de tempo (*time slice*) atribuída pelo Sistema Operativo a cada processo.

Mesmo o tempo de execução de um pequeno fragmento de código sofre várias perturbações devido a factores como: interrupções e excepções, presença em *cache* das instruções e dados acedidos pelo código, etc.

A metodologia de medição do tempo de execução deve minimizar a probabilidade de ocorrência destas interferências para que a métrica seja o mais fiável possível.

➢ Metodologia de medição

Pretende-se que a forma como a métrica é obtida minimize os efeitos da *cache* e das interrupções no valor final reportado.

Uma decisão inicial é se se pretende fazer as medições de tempo com a *cache* fria ou não, Por *cache* fria entende-se que o código e os dados necessários para a execução do programa não se encontram em *cache* na altura da medição, logo o tempo necessário para os aceder na memória central também é contabilizado. Optou-se aqui por não incluir este tempo. Para tal a função cujo tempo pretendemos medir é executada uma vez antes das medições começarem, para que a *cache* seja preenchida com a informação relevante.

Para medir o tempo de execução vamos usar a técnica das k-melhores medições. O raciocínio que sustenta esta técnica é que os erros de medição introduzidos pela comutação de contexto, interrupções e *cache* resultam sempre num aumento do tempo de execução dos programa. Assim medimos o tempo de execução da função várias vezes e guardamos os K melhores valores. Se a diferença entre o menor e maior destes valores fôr inferior a uma determinada tolerância, t, dizemos que a medição convergiu e o valor considerado é o menor destas K medições. Esta técnica tem 3 parâmetros:

K – numero de medições que devem convergir;

t – tolerância percentual;

M – número máximo de repetições de execução da função em questão, antes que desistamos da convergência.

4. Medição do desempenho de f()

Em http://gec.di.uminho.pt/lesi/ac2/praticas/ac2Mod5.tgz pode descarregar o código necessário para realizar este modulo. Pode descompactar este ficheiro escrevendo

```
tar xvzf ac2Mod5.tgz
```

O ficheiro `main.c` contem o código necessário para medir o tempo de execução da função `f()`. Esta função realiza uma operação aritmética (soma ou multiplicação) sobre um vector de elementos de um determinador tipos de dados (inteiro ou vírgula flutuante). O tamanho do vector varia e é determinado em `main()`. A função encontra-se definida no ficheiro `function1.c`:

```
/*
 * FUNCTION1: Abstract implementation
 *            No programmer optimization
 *
 */

#include "data.h"

#ifndef OPER_MULT
#define IDENT 0
#define OPER +
#else
#define IDENT 1
#define OPER *
#endif

void f (data_t *res) {
  int i=0;

  *res=IDENT;
  for (i=0 ; i < get_length () ; i++)
    *res = *res OPER get_elem(i);
}
```

➢ Optimizações introduzidas pelo compilador

Gere o executável correspondente a este programa para soma de inteiros sem utilizar qualquer nível de optimização para a compilação. Para tal certifique-se que o ficheiro `Makefile` tem

```
CFLAGS =
OPTIONS =
```

Para gerar o executável escreva

```
make
```

e depois execute-o:

```
./perf
```

Observe os CPEs para diversos tamanhos do vector. Consegue identificar alguma tendência? A que se deverá?

Preencha na tabela que se encontra no fim deste módulo o CPE correspondente à soma de inteiros sem optimizações, para o vector com 64K elementos.

Repita as medições para soma de números em vírgula flutuante. Para tal deve alterar a `Makefile`:

```
OPTIONS=-DUSE_FLOAT
```

Para forçar o `make` escreva:

```
touch *.c
make
./perf
```

Anote o CPE para 64K elementos na tabela.

Vamos agora usar as optimizações do compilador. Altera a `Makefile` de forma a que

```
CFLAGS= -O2
```

Recompile e execute o programa para inteiros e vírgula flutuante. Anote o CPE na tabela.

A que se deverá a diferença de CPE entre as versões com e sem optimização?

Gere o ficheiro *assembly* de `function1.c` para as duas versões e identifique as diferenças no código gerado. Para tal escreva:

```
gcc –S –c function1.c –o function1NoOpt.s
gcc –S –O2 –c function1.c –o function1.s
```

➢ Movimento de código

Analisando o código de `f()` na versão anterior é fácil verificar que uma razão para a ineficiência deste código é a invocação da função `get_length()` dentro do ciclo. Este código pode e deve ser movido para fora do ciclo.

Por que é que o compilador não o faz automaticamente?

Escreva uma nova versão de `f()` com esta optimização, normalmente designada por movimentação do código (*code motion*). Escreva esta versão num ficheiro chamado `function2.c` e altere a `Makefile` para que seja este o código usado. Anote os CPEs obtidos na tabela.

➢ Redução da invocação de procedimentos

O código de `f()` está escrito de forma estruturada. O acesso aos elementos do vector é feito através da função `get_elem()`, mantendo assim `f()` independente da implementação da estrutura de dados. No entanto, esta modularidade tem custos em termos de desempenho. A função `data_t * get_vec_addr()` devolve um apontador para o início do vector a tratar. Altere o código de `f()` de forma a invocar esta função antes do ciclo e aceda directamente aos elementos do vector. Escreva esta versão no ficheiro `function3.c` e altere a `Makefile`.

Preencha a tabela com os valores do CPE para um vector com 64K elementos.

A que se devem as diferenças observadas?

➢ Variáveis locais

A acumulação do resultado de `f()` é feita no parâmetro `data_t *res`, que é um apontador para uma variável qualquer. Isto obriga o compilador a escrever na memória o valor de `*res` em cada iteração do ciclo, impossibilitando-o de usar um registo e apenas escrever o resultado no fim do ciclo. Porque é que o compilador não pode fazer automaticamente esta optimização?

Escreva uma nova versão de `f()` no ficheiro `function4.c`, usando uma variável local para acumular o resultado e apenas actualizado `*res` no fim do ciclo. Anote os CPEs na tabela.

A que devem os valores obtidos?

Compare o código *assembly* de `function3.c` e `function4.c` identificando a diferença na forma como a acumulação do resultado é feita. Para isso escreva:

```
gcc –S –O2 –c function3.c –o function3.s
gcc –S –O2 –c function4.c –o function4.s
```

Versão	Inteiros	Virgula Flutuante
Sem optimização		
-O2		
Movimento de código		
Redução de procedimentos		
Variável local		