Arquitectura de Computadores II

LESI - 3º Ano

Medição de desempenho

João Luís Ferreira Sobral
Departamento do Informática
Universidade do Minho

Abril 2002

1. Introdução

A medição de desempenho pretende quantificar a quantidade de trabalho que um computador consegue realizar por unidade de tempo. Assim, é possível comparar o desempenho de várias máquinas e seleccionar a mais adequada para uma determinada função.

O desempenho pode ser medido a vários níveis, nomeadamente, ao nível das aplicações, determinando o número de soluções de um determinado problema calculadas por unidade de tempo (utilizando programas reais ou sintéticos, ex. SPEC2000, Dhrystones), ao nível do ISA, determinando a quantidade instruções efectuadas por segundo (MIPS, MFLOPS) e ao nível do *datapath*, utilizando a frequência de relógio ou a largura dos barramentos do processador.

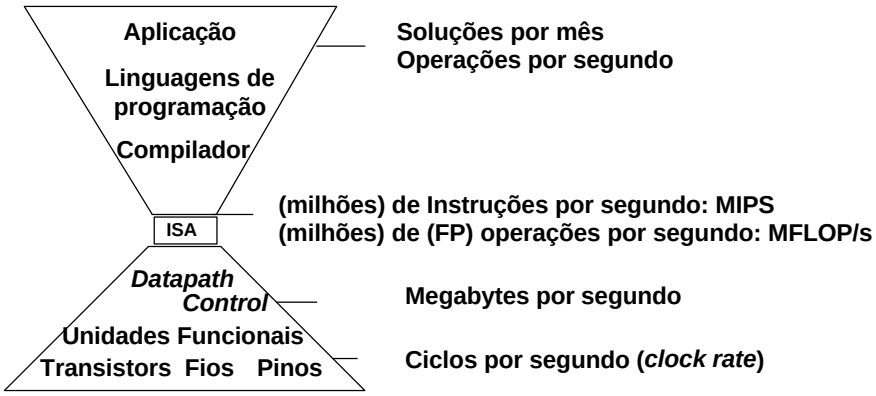

Figura 1 – Níveis de avaliação de desempenho

A medida mais indicada de desempenho é aquela que é obtida com aplicações reais, as quais serão efectivamente utilizadas na máquina que se pretende avaliar. Mesmo assim, as máquinas podem apresentar desempenhos diferentes consoante o tipo de aplicação. Por exemplo, as aplicações que envolvem gráficos dependem fortemente do desempenho da placa gráfica, enquanto as aplicações que envolvem bases de dados dependem mais do subsistema de disco. Por esta razão, actualmente o SPEC2000 é constituído por um conjunto de aplicações que pretendem medir o desempenho das máquinas em vários tipos de aplicações, permitindo seleccionar aquela que apresenta melhor desempenho no tipo de aplicações em que irá ser utilizada.

A utilização de programas sintéticos deve ser evitada, uma vez que frequentemente o desempenho obtido pouco tem a ver com o desempenho obtido com aplicações reais.

A medição do desempenho ao nível do ISA (em MIPS ou MFLOPS) apenas poderá ser utilizada quando se pretende comparar processadores que implementam um mesmo ISA. A utilização de medições obtidas a este nível para comparar processadores com ISA diferentes é errónea porque a quantidade de informação processada por cada instrução pode não ser igual nas duas arquitecturas.

A medida de desempenho menos adequada é a frequência de relógio, uma vez que a frequência de relógio pode não ter uma correspondência directa com o desempenho de aplicações reais. Recorde-se que o tempo de execução é dado por #I x CPI x Tcc.

As medições de desempenho realizadas nestes exercícios efectuam principalmente uma avaliação ao nível do ISA, não com o intuito de inferir qual o desempenho com aplicações reais mas com o intuito de determinar características do *datapath* do processador que terão impacto no tempo de execução de aplicações reais. Adicionalmente são realizados testes a subsistemas do computador.

2. Medição de tempo

A medição de tempo em computadores envolve a utilização de um relógio do sistema para medir o tempo de decorre entre o início e o fim de uma actividade. A precisão do relógio é a diferença entre o tempo medido pelo relógio do sistema e o tempo efectivamente decorrido. Por exemplo, o sistema pode indicar que decorreram 1003 segundos desde o início do programa mas na realidade podem apenas ter decorrido 1000 segundos. A resolução do relógio é a unidade de tempo mínima que decorre entre valores do relógio. Frequentemente a resolução de um relógio de pulso é na ordem do segundo, enquanto os relógios do computador têm uma resolução de milisegundos ou de microsegundos.

A resolução do relógio limita a duração mínima de eventos que podem ser medidos. Por exemplo, um relógio com uma precisão de 1 ms não permite a medição de eventos com duração inferior aos milisegundos. No entanto, esta limitação pode ser contornada se for possível medir a duração de múltiplas ocorrências do mesmo evento.

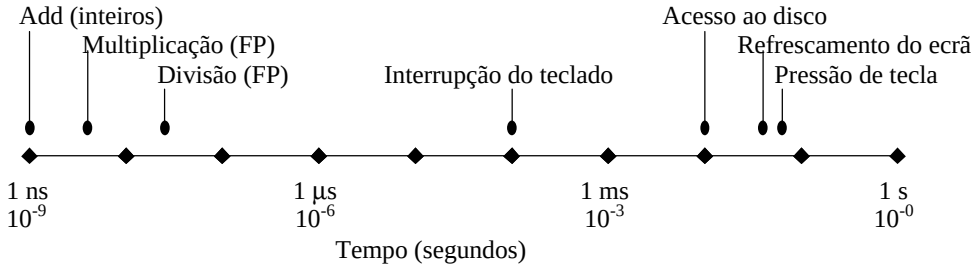

Figura 2 – Escala de tempo de eventos (Máquina a 1 GHz)

O processo de medição deve considerar eventuais sobrecargas introduzidas pelas rotinas de medição, nomeadamente, ao tempo medido deve ser descontado o tempo necessário para efectuar a chamada à rotina do relógio e o tempo introduzido por eventuais ciclos para repetir o evento em medição.

A medição de eventos cuja duração está entre os 10 milisegundos e os segundos é particularmente complexa, devido ao escalonamento de tarefas. Tal resulta do facto de a tarefa em medição poder ser suspensa e escalonada uma nova tarefa. Quando a duração da tarefa é superior a vários segundos este impacto é reduzido. Adicionalmente, no processo de medição devem ser efectuadas várias repetições e utilizar uma média de todas as medições ou das K melhores medições. Desta forma é minimizado o impacto quer do escalonamento, quer de outras tarefas realizadas pelo sistema operativo.

A microarquitectura Intel P6 (do Pentium Pro e posteriores) introduziu um relógio de elevada resolução, constituído por um contador de 64 bit que indica o número de ciclos decorridos desde que o computador foi ligado. Numa máquina de 1 GHz este contador retorna a zero (i.e., "dá a volta") após $2^{64} - 1$ segundos, ou seja, 570 anos. Note-se que se fosse implementado um contador apenas com 32 bits, numa máquina a 1 GHz, este voltaria a zero a cada 4,3 segundos.

O contador de ciclos de relógio pode ser acedido através da instrução rtdcs (*read time stamp counter*), cujo código máquina é 0x0F31. Esta instrução coloca os 32 bits mais significativos no registo EDX e os 32 bits menos significativos em EAX. A instrução rtdcs apenas está acessível quando o processador está no modo 32 bits.

Note-se que, actualmente, o contador de ciclos de relógio também está implementado nos processadores AMD Athlon e Duron, além dos já referidos Pentium Pro, Pentium II, Pentium III, Pentium IV e Pentium Xeon.

3. Conjunto de instruções IA-32

A arquitectura IA-32 apresenta 8 registos com 32 bits de uso genérico. Por questões de compatibilidade alguns dos registos podem ser divididos em registos e 16 bits, que por sua vez podem ser divididos em registos de 8 bits. Existe também um registo de controlo e de estado utilizado para controlar a execução do processador e implementar as condições de salto e um apontador de instruções.

Registos de uso genérico

32 bits	16 bits	31 … 16	15 … 8	7 … 0
EAX	AX		AH	AL
EBX	BX		BH	BL
ECX	CX		CH	CL
EDX	DX		DH	DL
EBP	BP		BP	
ESI	SI		SI	
EDI	DI		DI	
ESP	SP		SP	

Registo de controlo e de estado

EFLAGS	FLAGS		FLAGS

Apontador de instruções

EIP	IP		IP

O conjunto de instruções IA-32 apresenta múltiplos modos de endereçamento: imediato, registo, indirecto por registo, indirecto por valor imediato, indirecto por base + deslocamento, etc. A figura seguinte apresenta as alternativas para especificar os operandos em memória:

$$endereço = \left[\underbrace{\begin{bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \\ ESP \end{bmatrix}}_{base} + \underbrace{\left[\begin{pmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \right]}_{índice * escala} + \underbrace{\begin{bmatrix} nenhum \\ 8bits \\ 16bits \\ 32bits \end{bmatrix}}_{desloc} \right]$$

Base, índice, escala e deslocamento são todos opcionais.

Alguns exemplos de endereços:

[eax]
[ebx + ecx]
[edx + 8]
[ebp + esi +12]
[esp + edi*4 + 4].

O conjunto de instruções IA-32 possibilita a referência a operandos em memória (contrariamente à arquitectura MIPS que apenas permite acesso à memória através de instruções de *lw* e *sw*). Frequentemente, devido ao número limitado de registos é necessário colocar variáveis do programa em memória. O seguinte código compara um programa em *assembly* MIPS com o mesmo programa em IA-32:

C
```
int i;
i= i + 1;
```

MIPS
```
addi $s0, $s0, 1
```

IA32
```
i: dw 0
LEA  EBX, i
INC dword ptr [EBX]
```

Em IA-32 é implementado um modelo baseado no registo de *flags*, um registo que memoriza informações sobre o resultado da última operação. Entre as várias condições armazenadas neste registo incluem-se a indicação do sinal do resultado e a igualdade a zero. As instruções de salto condicional utilizam a informação contida no registo de *flags*. A instrução CMP (*compare*) compara o valor de dois operandos, apenas alterando o registo de *flags*.

O seguinte código compara um trecho de programa em *assembly* MIPS com o mesmo trecho em IA-32:

C	**MIPS**	**IA-32**
`if (i<12)`	`slt $t0, $s0, 12`	`CMP dword ptr [EBX],12`
`...`	`beq $t0, $0, else`	`JGE else`

O conjunto de instruções IA-32 utiliza em regra dois operandos, o primeiro funciona geralmente como fonte e destino, enquanto o segundo funciona apenas como fonte. A fonte pode ser um registo, um valor imediato ou um endereço de memória. O destino é um registo ou um endereço de memória. Quando o destino é um endereço de memória a fonte não pode ser também um endereço de memória. A tabela seguinte apresenta algumas das instruções do IA-32 mais relevantes:

```
Src = Rsrc | Mem | Imm32
Dest = Rdest | Mem
Dest e Src não podem ser simultaneamente memória
Rdest, Rsrc = EAX | EBX | ECX | EDX | ESI | EDI | EBP
Mem = [ base + índice * escala + Imm32 ]

    mov Dest, Src         # Dest = Src
    lea Rdest, End        # Rdest = End
    add Dest, Src         # Dest = Dest + Src
    sub Dest, Src         # Dest = Dest - Src
    and Dest, Src         # Dest = Dest & Src
    inc Dest              # Dest = Dest + 1
    dec Dest              # Dest = Dest – 1
    imul Rdest, Src       # Rdest = Src * Rdest
    cmp Rsrc1, Src2       # F = resultado da comparação
    j<cc> End             # <cc> = [L, LE, G, GE, E, NE]
    jmp End               # EIP = End
```

A generalidade dos compiladores de C/C++ permite a utilização de *assembly* embutido. Para tal basta incluir o *assembly* na primitiva asm { .... }. Adicionalmente é possível referenciar variáveis do programa C nas primitivas em *assembly*. Por exemplo, o código seguinte copia o conteúdo do registo EAX para a variável i:

```
int i;
asm { mov i, eax }
```

É também possível forçar um determinado código, mesmo que o *assembler* não reconheça o *assembly* correspondente (por exemplo o *rtdcs*)

```
asm { dw 0x310F }  // RTDSC (little endian)
```

Exercício 1 – Processador

Objectivo:

- Medir as características de desempenho básicas do processador
- Identificar as sequências de instruções necessárias para determinar cada característica do processador

Pretende-se com este exercício desenvolver um programa em C, com *assembly* embutido, que avalie várias características do processador, nomeadamente a frequência do relógio, o CPI de instruções aritméticas e de *load* e a penalização (i.e., ciclos de *stall*) decorrente das dependências de dados do tipo RAW e das dependências de controlo.

Desenvolva um programa que permita medir o seguinte:

a) frequência de relógio do processador. Implemente primeiro uma rotina para aceder ao contador de ciclos de relógio do processador, utilizando a instrução *RTDSC*. Depois pode utilizar essa rotina em conjunto com a função do Windows *GetTickCount*, que permite obter o tempo decorrido, para determinar a frequência de relógio.

b) CPI da instrução LOOP *end* (que decrementa o registo ECX e salta para o endereço *end* se ECX for diferente de 0) e o CPI obtido utilizando como alternativa o par de instruções DEC ECX e JNZ *end*. Poderá obter o número de ciclos com precisão se colocar inicialmente em ECX um valor relativamente elevado. O número de ciclos necessário para cada iteração do ciclo será igual ao número de ciclos necessário para executar todo o ciclo dividido pelo número de iterações.

c) CPI das instruções ADD, IMUL e MOV de memória para registo. Para tal altere o ciclo da alínea anterior por forma a que este inclua várias instruções ADD, IMUL ou MOV **independentes** entre si.

d) CPI das instruções ADD, IMUL e MOV quando existem dependências RAW. Para tal altere o ciclo da alínea anterior por forma a que este inclua várias instruções ADD, IMUL ou MOV **dependentes** entre si. No caso do MOV, para que cada instrução dependa do resultado da anterior, deve ser percorrida uma estrutura de dados em memória do tipo lista ligada.

e) penalização decorrente de um salto previsto erradamente. Para tal desenvolva um ciclo que contenha um *if* que é sempre executado e compare-o com um ciclo que contém um *if* dependente de um valor aleatório (obtido com *rand*). A diferença de CPI entre as duas versões corresponde a metade da penalização dos saltos previstos erradamente, uma vez que no segundo caso a previsão falha 50% das vezes.

Exercício 2 – Hierarquia de memória

Objectivo:

- Medir a desempenho dos vários níveis da hierarquia de memória

Pretende-se com este exercício desenvolver um programa em C, possivelmente com *assembly* embutido, que meça o desempenho vários níveis da hierarquia de memória, desde a cache L1 até um disco na rede local.

Desenvolva um programa que permita medir o seguinte:

a) número médio de ciclos do processador necessário para aceder à memória cache nível 1, nível 2 e RAM. Simultaneamente poderá determinar a dimensão da cache nível 1 e nível 2.

b) número médio de ciclos do processador necessário para transferir cada unidade de informação do disco duro. Utilize a função *fread* para ler blocos de informação do disco.

c) idêntico à alínea b) mas criando o ficheiro numa rede local (por exemplo, através de um directório partilhado).

Exercício 3 – Codificação de software

Objectivo:

- Medir o impacto no desempenho de diversas formas de codificação de software

Pretende-se com este exercício desenvolver um programa em C que compare o desempenho de duas funções codificadas de forma distinta, explicando a proveniência das diferenças obtidas no desempenho.

Desenvolva um programa que permita medir o seguinte:

a) comparar o desempenho de uma função que calcula o factorial de forma iterativa e uma função que o calcula de forma recursiva. Justifique os resultados obtidos, possivelmente recorrendo à visualização do *assembly* gerado para cada uma das duas funções.

Função iterativa	Função recursiva

`int factIter(int n) {`
`  int f=1;`
`  while(n > 1) { f*=n; n--; }`
`  return(f);`
`}`
	`int factRec(int n) {`
	`  if (n <= 1 ) return(1);`
	`  return( n * factRec(n-1) );`
	`}`

b) comparar o desempenho de uma função que some os elementos de um *array* de duas dimensões, efectuando a soma por linhas e uma função que efectue a mesma soma dos elementos por colunas. Justifique os resultados obtidos.

Soma por linhas	Soma por colunas

`soma=0;`
`for(int i=0; i<M; i++)`
`  for(int j=0; j<N; j++)`
`  soma += a[i][j];`
	`soma=0;`
	`for(int j=0; j<N; ij++)`
	`  for(int i=0; i<M; i++)`
	`  soma += a[i][j];`