Mestr. Integr. Engª Informática

1º ano

2016/17

A.J.Proença

Tema

Avaliação de Desempenho (IA-32)

Estrutura do tema Avaliação de Desempenho (IA-32)

1. A avaliação de sistemas de computação
2. Técnicas de otimização de código (IM)
3. Técnicas de otimização de *hardware*
4. Técnicas de otimização de código (DM)
5. Outras técnicas de otimização
6. Medição de tempos …

"*Análise do desempenho*": para quê?

– **para avaliar Sistemas de Computação**
  - identificação de métricas
    – latência, velocidade, ...
  - ligação entre métricas e fatores na arquitetura que influenciam o desempenho de um CPU/núcleo

$$Core_{time} = N^{o}_{instr} * CPI * T_{clock}$$

e . . .

– **... construi-los mais rápidos**

– **... melhorar a eficiência de execução de *app's***

Análise dos componentes da fórmula:

– **$Core_{time}$**
  - tempo de execução no CPU/*core*, inclui acessos à memória, ...

– **N^{o}_{instr}**
  - efetivamente executadas; depende essencialmente de:
    – eficiência do compilador
    – do *instruction set*

– **CPI** (*Clock-cycles Per Instruction*)
  - tempo médio de exec de 1 instr, em ciclos; depende essencial/:
    – complexidade da instrução (e acessos à memória ...)
    – paralelismo na execução de instruções

– **T_{clock}**
  - período do *clock*; depende essencialmente de:
    – complexidade da instrução (ao nível dos sistemas digitais)
    – microeletrónica

"*Análise do desempenho*": para quê?

– ... **melhorar a eficiência de execução de *app's***

- análise de técnicas de otimização do código
  - algoritmo / **codificação / compilação** / *assembly*
  - compromisso entre legibilidade e eficiência...
  - potencialidades e limitações dos compiladores...
  - técnicas independentes / dependentes da máquina
  - uso de *code profilers*
- técnicas de medição de tempos
  - escala microscópica / macroscópica
  - uso de *cycle counters* / *interval counting*
  - métodos de medição confiável de tempos de execução

– **um compilador moderno já inclui técnicas que**

- exploram oportunidades para simplificar expressões
- usam um único cálculo de expressão em vários locais
- reduzem o nº de vezes que um cálculo é efetuado
- tiram partido de algoritmos sofisticados para
  - alocação eficiente dos registos
  - seleção e ordenação de código
- ... **mas** está limitado por certos fatores, tais como
  - nunca modificar o comportamento correto do programa
  - limitado conhecimento do programa e seu contexto
  - necessidade de ser rápido!

– **e certas otimizações estão-lhe vedadas...**

– **exemplos de otimizações vedadas aos compiladores:**

- pode trocar `twiddle1` por `twiddle2` ?

```
void twiddle1(int *xp,int *yp)
{
  *xp += *yp;
  *xp += *yp;
}
```

```
void twiddle2(int *xp,int *yp)
{
  *xp += 2* *yp;
}
```

teste: `xp` igual a `yp`; que acontece?

- pode trocar `func1` por `func2` ?

```
int f(int n)
int func1(x)
{
  return f(x)+f(x)+f(x)+f(x);
}
```

```
int f(int n)
int func2(x)
{
  return 4*f(x);
}
```

teste: e se `f` for...?

```
int counter = 0;

int f(int x)
{
  return counter++;
}
```