Optimização do Desempenho:
Técnicas Independentes da Máquina

Arquitectura de Computadores

Lic. em Engenharia Informática

Luís Paulo Santos

Optimização: Independência do Processador

	10 – Optimização do Desempenho
Conteúdos	10.1 – Capacidades e Limitações dos Compiladores
	10.2 – Técnicas Independentes do Processador: optimização de ciclos, invocação de procedimentos e acessos a memória
Resultados de Aprendizagem	R10.1 – Descrever, aplicar e avaliar técnicas de optimização de desempenho

Optimização do desempenho

A optimização decorre ao longo de 2 eixos:

- escolha cuidada dos **algoritmos** e **estruturas de dados**
  (responsabilidade do programador)

- geração de código optimizado
  (responsabilidade do compilador …
  … com a ajuda do programador)

OPTIMIZAÇÃO = COMPROMISSO

Aumento do desempenho ... mas ...

- aumento do tempo de desenvolvimento
- código mais ilegível
- diminuição da modularidade
- eventual perca de portabilidade

Optimização do desempenho

- Optimizações **independentes** da máquina
  Aplicáveis em qualquer plataforma alvo.
  Exemplos:
    - movimentação de código (*code motion*)
    - redução de acessos à memória
    - redução das invocações de procedimentos

- Optimizações **dependentes** da máquina
  Requerem um modelo da organização e temporização da máquina alvo
  Exemplos:
    - utilização de instruções mais rápidas
    - cálculo eficiente de expressões e endereços
    - exploração de múltiplas unidades funcionais *(loop unrolling)*
    - exploração do *pipeline (loop splitting)*

Optimização: limitações dos compiladores

- Limitação Fundamental:
  - Não podem, em nenhuma circunstância, causar alterações do comportamento do programa
  - Frequentemente, isto impede a aplicação de optimizações que apenas afectariam o comportamento em condições patológicas.
- Conhecimento limitado do contexto do programa:
  - Análise feita maioritariamente dentro de um procedimento (método ou função)
    - Análise de todo o programa impossível ou demasiado morosa na maior parte dos casos
  - Análise baseada maioritariamente em informação estática
    - O compilador não pode antecipar o valor de dados em tempo de execução
- A compilação deve terminar em tempo útil

Bloqueadores de Optimização

```
func1 (int *xp, int *yp)
{
  *xp += *yp;
  *xp += *yp;
}
```

⇔

??

```
func2 (int *xp, int *yp)
{
  *xp += 2*(*yp);
}
```

... mas se `xp==yp` então `func1()` calcula `*xp*4`,
enquanto `func2()` calcula `*xp*3`

Memory aliasing – os cálculos realizados usando apontadores podem ter resultados inesperados se estes referirem a mesma zona de memória.

Bloqueadores de Optimização

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

⇔

??

```
int func2 (int x)
{
  return 3*f(x);
}
```

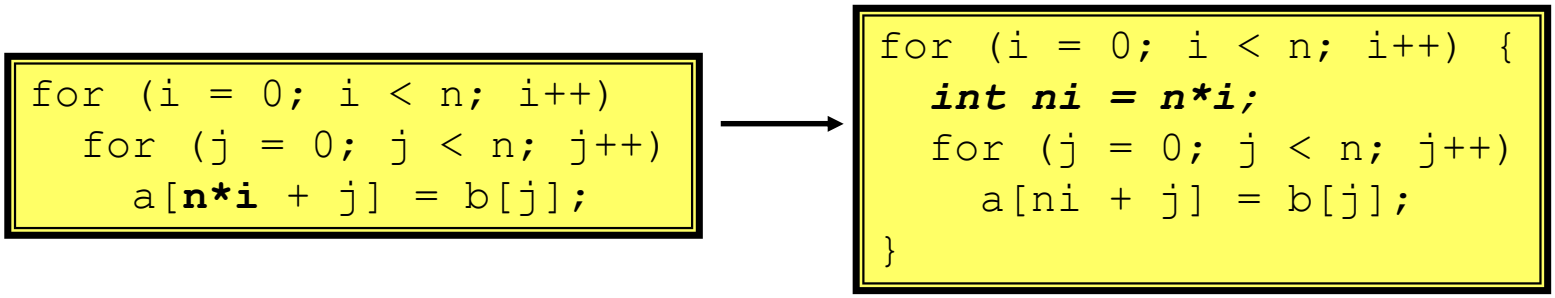
... mas se `f()` altera o estado então …

```
int counter = 0;
int f(int p) {
  return(counter++);
}
```

Efeitos colaterais – Se uma função altera variáveis globais, então a sua invocação não pode ser substituída por outra operação mais eficiente

Movimentação de código

- Reduzir o número de vezes que um cálculo é feito
  - Se, e só se, produz sempre o mesmo resultado
  - Em particular, mover expressões para fora dos ciclos



Movimentação de código

```
func (char *s) {
  int i;

  for (i=0 ; i<strlen(s); i++)
    s[i] += 10;
}
```

```
func (char *s) {
  int i, l;

  l = strlen(s);
  for (i=0 ; i<l; i++)
    s[i] += 10;
}
```

O cálculo de `strlen(s)` pode ser movido para fora do ciclo, mas:

- o compilador tem dificuldade em determinar que o corpo do ciclo não altera o resultado do cálculo de `strlen(s)`

- o compilador não consegue determinar se `strlen()` tem efeitos colaterais, tais como alterar alguma variável global.

Redução de acessos à memória

```
func (int *s, int *acc) {
  int i;

  *acc=0;
  for (i=0 ; i<10; i++)
    *acc +=s[i];
}
```

```
 movl 12(%ebp), %esi    #%esi = acc
 movl $0, 0(%esi)       # *acc=0
 movl $0, -4(%ebp)      # i=0
Ciclo:
 cmpl $10, -4(%ebp)     # i<10 ??
 jge fim
 movl -4(%ebp), %eax
 sall $2, %eax
 addl 8(%ebp), %eax
 movl (%eax), %edx      # %edx=s[i]
 addl %edx, 0(%esi)     # *acc+=s[i]
 incl -4(%ebp)          # i++
 jmp Ciclo
```

Compilador **não pode** evitar acessos a `*acc` na memória com receio de *memory aliasing*
Poderia no entanto evitar colocar `i` em memória

Redução de acessos à memória

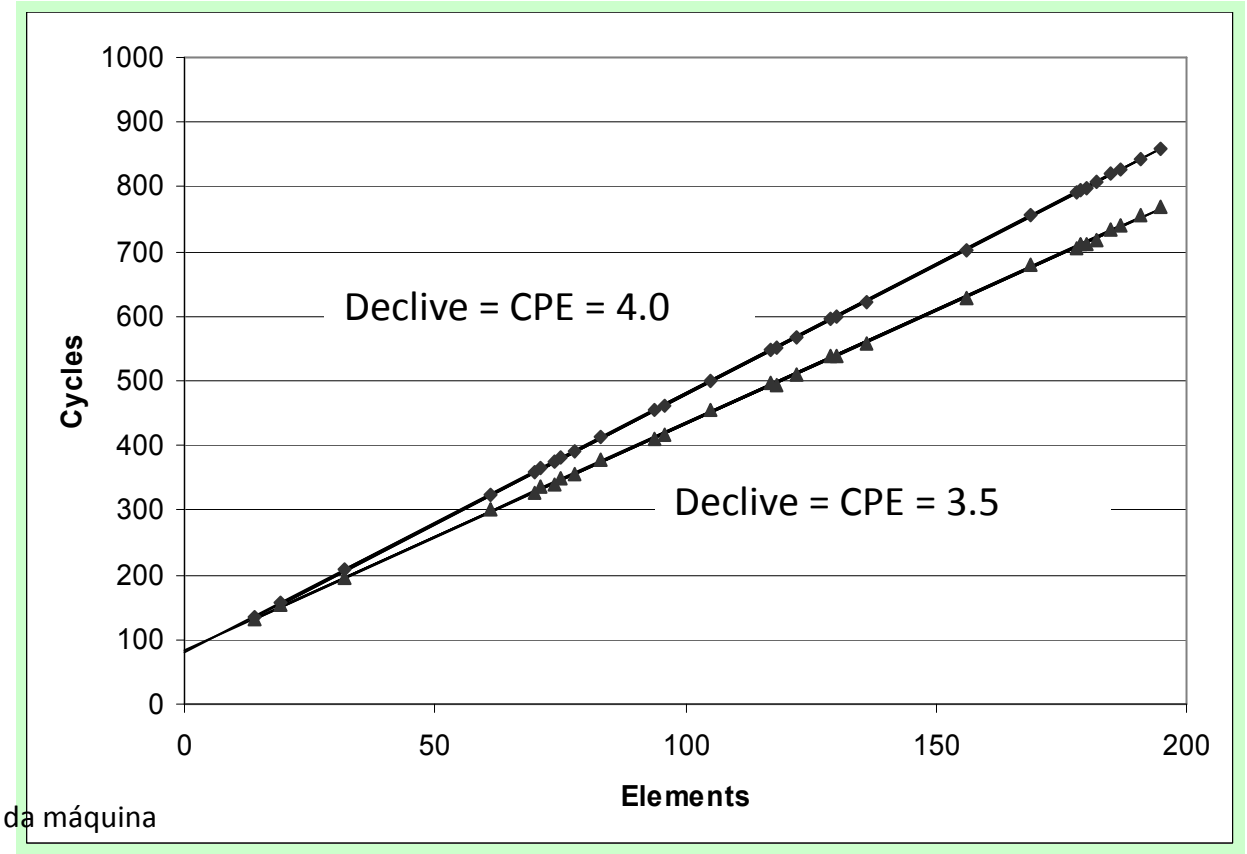
```
func (int *s, int *acc) {
  int i, t;

  t=0;
  for (i=0 ; i<10; i++)
        t +=s[i];
  *acc = t;
}
```

```
 movl $0, %ebx      # t=0
 movl $0, %ecx      # i=0
Ciclo:
 cmpl $10, %ecx       # i<10 ??
 jge fim
 movl %ecx, %eax
 sall $2, %eax
 addl 8(%ebp), %eax
 movl (%eax), %edx    # %edx=s[i]
 addl %edx, %ebx      # t+=s[i]
 incl %ecx            # i++
 jmp Ciclo
Fim:
 movl 12(%ebp), %esi
 movl %ebx, 0(%esi)
```

Compilador **pode** evitar acessos a `*acc` na memória pois não há hipótese de *memory aliasing*

Métrica: Ciclos por Elemento (CPE)

- Exprimir o desempenho de operadores que processam vectores ou listas
- Comprimento = n (nº de elementos no vector)
- T = CPE*n + Custo_Inicial



Exemplo de Optimização

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

– Calcula a soma de todos os elementos do vector

– Guarda o resultado no parâmetro de referência `dest`

Pentium II/III CPE: 42.06 (-g) 31.25 (-O2)

Exemplo de Optimização: *Code Motion*

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Mover vec_length() para fora do ciclo mais aninhado
- CPE: 20.66 (Compiled -O2)

Exemplo de Optimização: *Invocação de funções*

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

– Evitar função para aceder aos elementos de v
  • Ler apontador para início do vector fora do ciclo
– CPE: 6.00 (Compiled -O2)
  • Invocação de funções podem ser dispendiosas

Exemplo de Optimização: *Acessos à Memória*

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

- Resultado só é armazenado no destino no fim
- Variável local `sum` armazenada em registo
- Evita 1 leitura e 1 escrita na memória por iteração
- CPE: 2.00 (Compiled -O2)
  - Acessos à memória podem ser dispendiosos