



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 de técnicas de otimização (1)



Análise de técnicas de otimização (s/w)

- técnicas independentes da máquina ... *já visto...*
- **técnicas de otimização de código (dep. máquina)**
 - análise sucinta de um CPU atual, P6 (*já visto...*)
 - **loop unroll** sem e com paralelismo
 - **inline functions**
 - **identificação de potenciais limitadores de desempenho**
 - dependentes da hierarquia da memória
- **outras técnicas de otimização**
 - **na compilação**: otimizações efectuadas pelo GCC
 - **na identificação dos "gargalos" de desempenho**
 - *code profiling* e uso dum *profiler* para apoio à otimização
 - **lei de Amdahl**

Técnicas de otimização dependentes da máquina: loop unroll (1)



```
void combine5(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-2;
    int *data = get_vec_start(v);
    int sum = 0;
    int i;
    /* junta 3 elem's no mesmo ciclo */
    for (i = 0; i < limit; i+=3) {
        sum += data[i] + data[i+1]
            + data[i+2];
    }
    /* completa os restantes elem's */
    for (; i < length; i++) {
        sum += data[i];
    }
    *dest = sum;
}
```

Otimização 4:

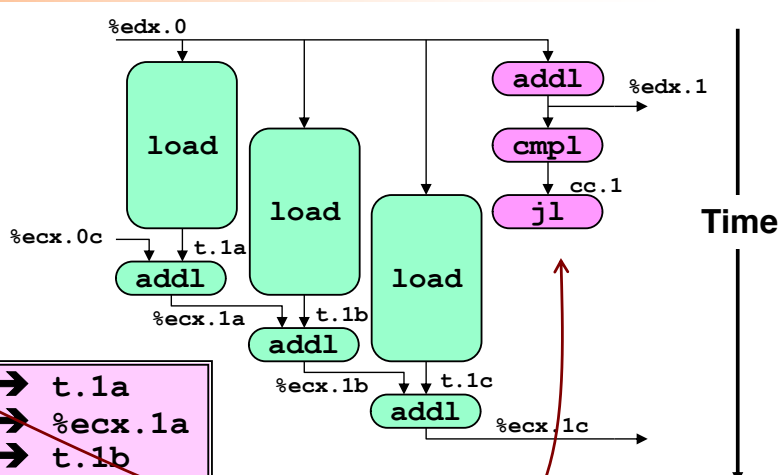
- juntar várias (3) iterações num simples ciclo
- amortiza *overhead* dos ciclos em várias iterações
- termina extras no fim
- **CPE: 1.33**

Técnicas de otimização dependentes da máquina: loop unroll (2)

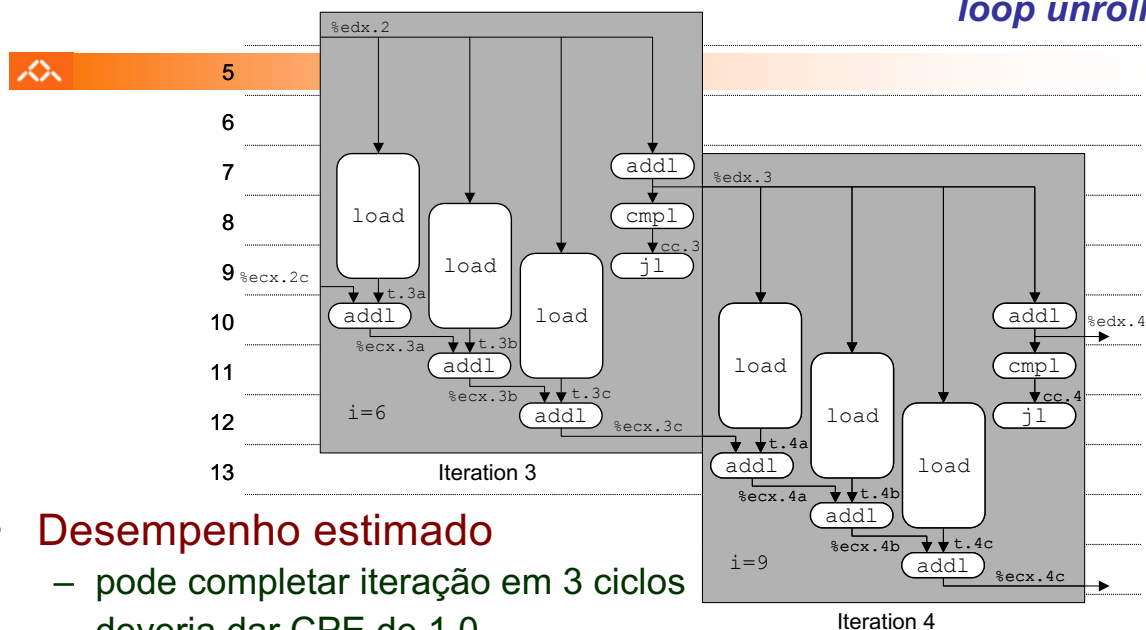


- loads podem encadear, uma vez que não há dependências
- apenas um conjunto de instruções de controlo de ciclo

| | |
|-----------------------|-----------|
| load (%eax,%edx.0,4) | → t.1a |
| iaddl t.1a, %ecx.0c | → %ecx.1a |
| load 4(%eax,%edx.0,4) | → t.1b |
| iaddl t.1b, %ecx.1a | → %ecx.1b |
| load 8(%eax,%edx.0,4) | → t.1c |
| iaddl t.1c, %ecx.1b | → %ecx.1c |
| iaddl \$3,%edx.0 | → %edx.1 |
| cmpl %esi, %edx.1 | → cc.1 |
| j1 -taken cc.1 | |



Técnicas de otimização dependentes da máquina: loop unroll (3)



- **Desempenho estimado**
 - pode completar iteração em 3 ciclos
 - deveria dar CPE de 1.0
- **Desempenho medido**
 - CPE: 1.33
 - 1 iteração em cada 4 ciclos

AJProença, Sistemas de Computação, UMinho, 2018/19

5

Técnicas de otimização dependentes da máquina: loop unroll (4)



Valor do **CPE** para várias situações de *loop unroll*:

| Grau de <i>Unroll</i> | | 1 | 2 | 3 | 4 | 8 | 16 |
|-----------------------|---------|------|------|------|------|------|------|
| Inteiro | Soma | 2.00 | 1.50 | 1.33 | 1.50 | 1.25 | 1.06 |
| Inteiro | Produto | 4.00 | | | | | |
| <i>fp</i> | Soma | 3.00 | | | | | |
| <i>fp</i> | Produto | 5.00 | | | | | |

- apenas melhora nas somas de inteiros
 - restantes casos há restrições com a latência da unidade
- efeito não é linear com o grau de *unroll*
 - há efeitos subtis que determinam a atribuição exacta das operações

AJProença, Sistemas de Computação, UMinho, 2018/19

6

- a computação...

$$((((((((((((((((1 * x_0) * x_1) * x_2) * x_3) * x_4) * x_5) * x_6) * x_7) * x_8) * x_9) * x_{10}) * x_{11})$$

- N elementos, D ciclos/operação
- N*D ciclos

- a computação...

$$\begin{aligned} &((((((((1 * x_0) * x_2) * x_4) * \\ &\quad * x_6) * x_8) * x_{10}) * \\ &((((((((1 * x_1) * x_3) * x_5) * \\ &\quad * x_7) * x_9) * x_{11}) \end{aligned}$$

- N elementos, D ciclos/op
- $(N/2+1)*D$ ciclos
- melhoria de $\sim 2x$

Técnicas de otimização dependentes da máquina: loop unroll com paralelismo (1)



```
void combine6(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int x0 = 1;
    int x1 = 1;
    int i;
    /* junta 2 elem's de cada vez */
    for (i = 0; i < limit; i+=2) {
        x0 *= data[i];
        x1 *= data[i+1];
    }
    /* completa os restantes elem's */
    for (; i < length; i++) {
        x0 *= data[i];
    }
    *dest = x0 * x1;
}
```

... versus paralela!

Otimização 5:

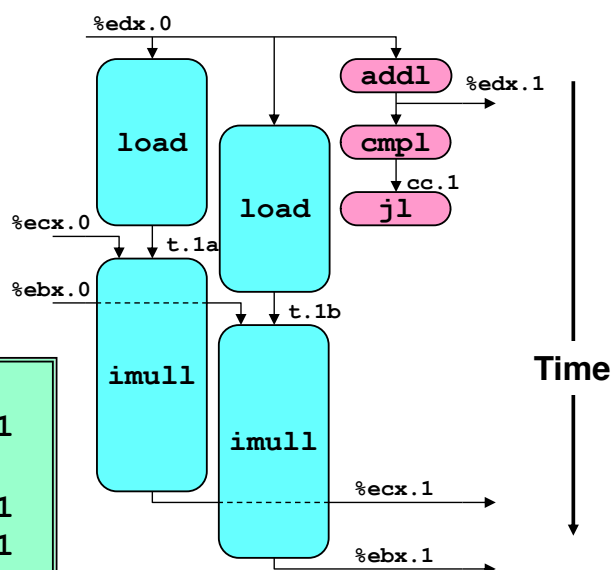
- acumular em 2 produtos diferentes
 - pode ser feito em paralelo, se OP fôr associativa!
- juntar no fim
- Desempenho
 - **CPE: 2.0**
 - melhoria de 2x

Técnicas de otimização dependentes da máquina: loop unroll com paralelismo (2)

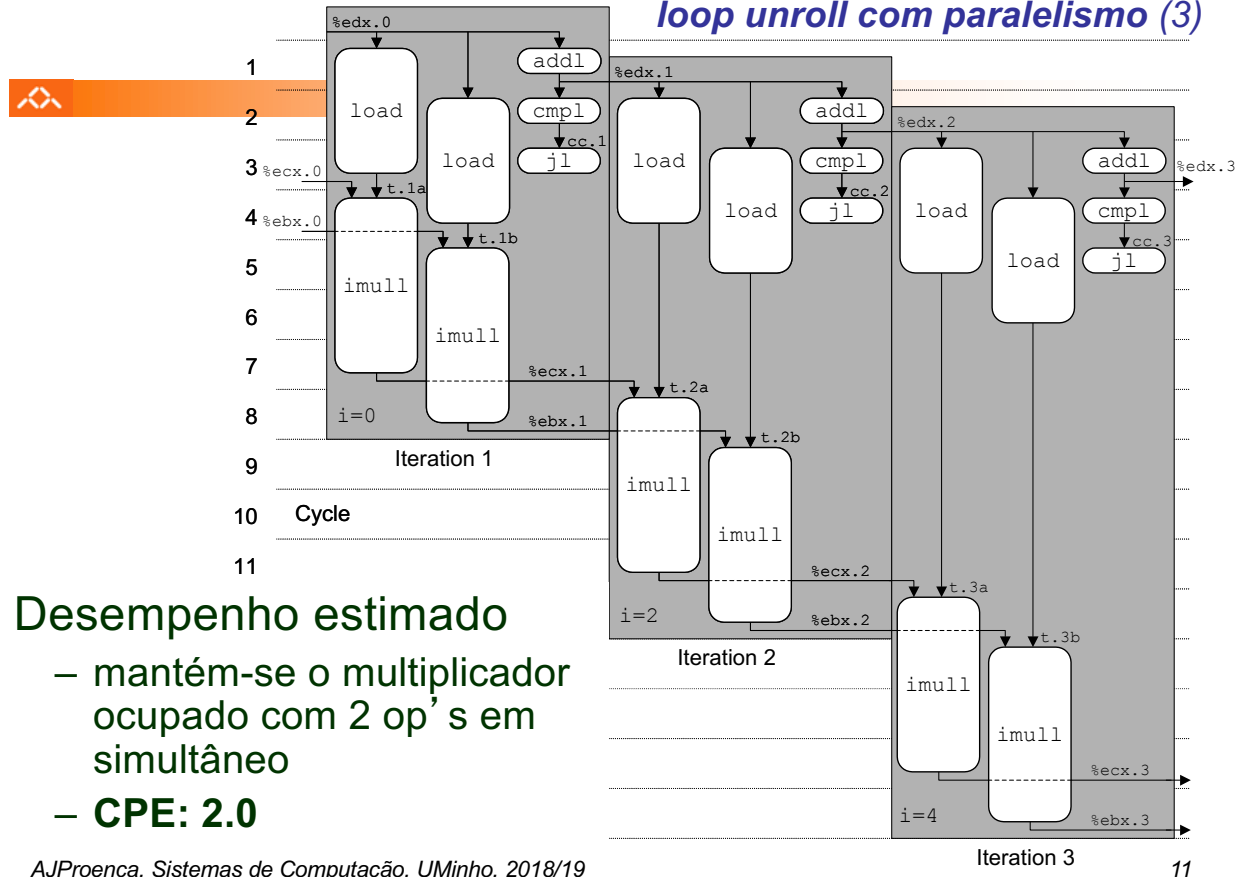


- os dois produtos no interior do ciclo não dependem um do outro...
- e é possível encadeá-los
- *iteration splitting*, na literatura

```
load (%eax,%edx.0,4)  → t.1a
imull t.1a, %ecx.0     → %ecx.1
load 4(%eax,%edx.0,4) → t.1b
imull t.1b, %ebx.0     → %ebx.1
iaddl $2,%edx.0       → %edx.1
cmpl %esi, %edx.1     → cc.1
j1-taken cc.1
```



Técnicas de otimização dependentes da máquina: loop unroll com paralelismo (3)



Técnicas de otimização de código: análise comparativa de combine

| Método | Inteiro | | Real (precisão simples) | |
|-----------------------------|---------|-------|-------------------------|--------|
| | + | * | + | * |
| Abstract -g | 42.06 | 41.86 | 41.44 | 160.00 |
| Abstract -O2 | 31.25 | 33.25 | 31.25 | 143.00 |
| Move vec_length | 20.66 | 21.25 | 21.15 | 135.00 |
| Acesso aos dados | 6.00 | 9.00 | 8.00 | 117.00 |
| Acum. em temp | 2.00 | 4.00 | 3.00 | 5.00 |
| Unroll 4x | 1.50 | 4.00 | 3.00 | 5.00 |
| Unroll 16x | 1.06 | 4.00 | 3.00 | 5.00 |
| Unroll 2x, paral. 2x | 1.50 | 2.00 | 2.00 | 2.50 |
| Unroll 4x, paral. 4x | 1.50 | 2.00 | 1.50 | 2.50 |
| Unroll 8x, paral. 4x | 1.25 | 1.25 | 1.50 | 2.00 |
| Otimização Teórica | 1.00 | 1.00 | 1.00 | 2.00 |
| Rácio Pior : Melhor | 39.7 | 33.5 | 27.6 | 80.0 |



- Precisa de muitos registos!
 - para guardar somas/produtos
 - apenas 6 registos (p/ inteiros) disponíveis no IA-32
 - tb usados como apontadores, controlo de ciclos, ...
 - 8 registos de fp
 - quando os registos são insuficientes, temp's vão para a *stack*
 - elimina ganhos de desempenho
(ver *assembly* em produto inteiro com *unroll 8x* e paralelismo 8x)
 - re-nomeação de registos não chega
 - não é possível referenciar mais operandos que aqueles que o *instruction set* permite
 - ... principal inconveniente do *instruction set* do IA-32
- Operações a paralelizar têm de ser associativas
 - a soma e multipl de fp num computador não é associativa!
 - $(3.14+1e20)-1e20$ nem sempre é igual a $3.14+(1e20-1e20)$...

*Limitações do paralelismo:
a insuficiência de registos*



- **combine**
 - produto de inteiros
 - *unroll 8x* e paralelismo 8x
 - 7 variáveis locais partilham 1 registo (`%edi`)
 - observar os acessos à *stack*
 - melhoria desempenho é comprometida...
 - *register spilling* na literatura

```
.L165:
    imull (%eax),%ecx
    movl -4(%ebp),%edi
    imull 4(%eax),%edi
    movl %edi,-4(%ebp)
    movl -8(%ebp),%edi
    imull 8(%eax),%edi
    movl %edi,-8(%ebp)
    movl -12(%ebp),%edi
    imull 12(%eax),%edi
    movl %edi,-12(%ebp)
    movl -16(%ebp),%edi
    imull 16(%eax),%edi
    movl %edi,-16(%ebp)
    ...
    addl $32,%eax
    addl $8,%edx
    cmpl -32(%ebp),%edx
    jl .L165
```



Análise de técnicas de otimização (s/w)

- técnicas de otimização de código (indep. máquina)
 - *já visto...*
- técnicas de otimização de código (dep. máquina)
 - dependentes do processador (*já visto...*)
- **outras técnicas de otimização**
 - na compilação: otimizações efectuadas pelo GCC
 - na identificação dos "gargalos" de desempenho
 - *code profiling*
 - uso dum *profiler* para apoio à otimização
 - **lei de Amdahl**
 - dependentes da hierarquia da memória
 - a localidade espacial e temporal dum programa
 - influência da *cache* no desempenho

Code profiling: análise visual da melhoria de código numa função



Antes,
96% na função
ProportRegionGrow

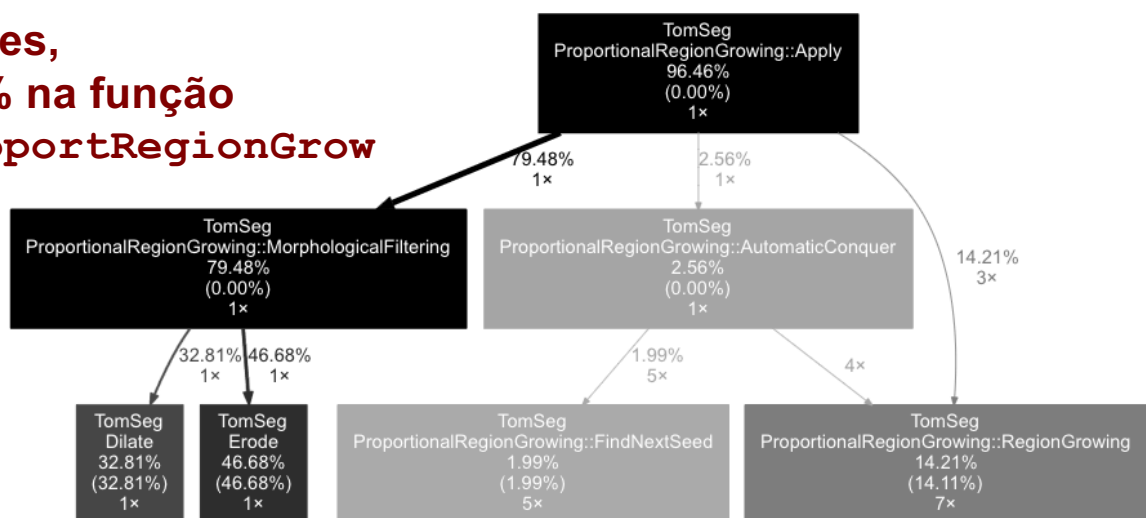


Figure 5.7.: Call-graph of the first version of *Proport. Region Growing* (DS3)

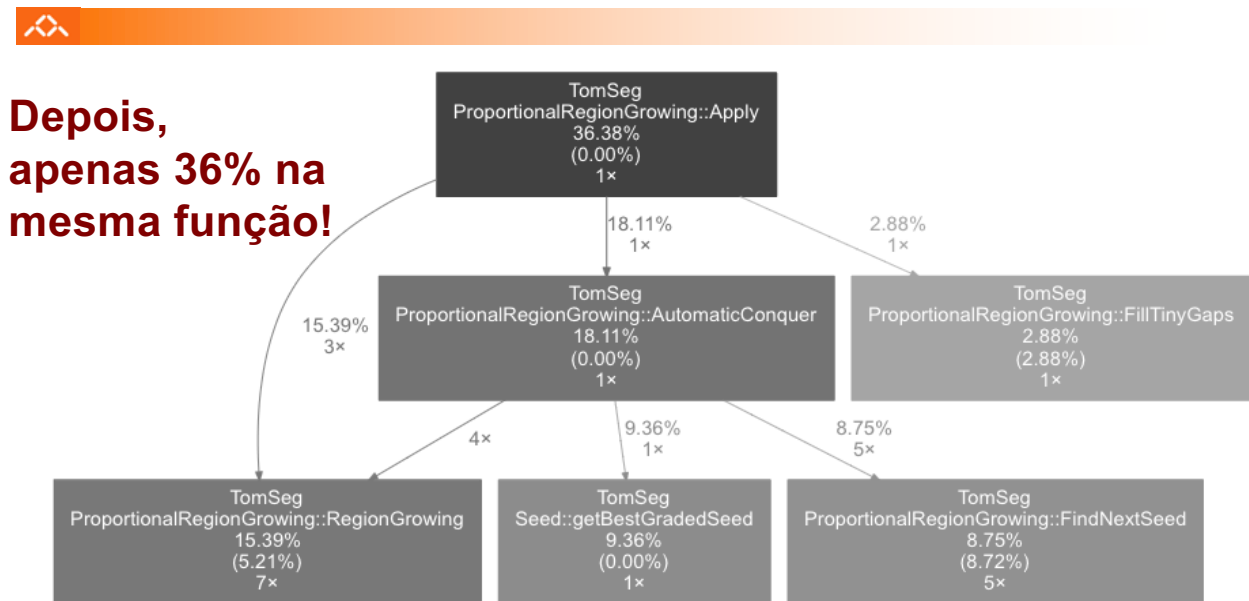


Figure 5.9.: Call-graph from the last version of *Propor. Region Growing* (DS3)

Lei de Amdahl

O ganho no desempenho – *speedup* –

obtido com a melhoria do tempo de execução de uma parte do sistema, está limitado pela fração de tempo que essa parte do sistema pode ser usada.

$$Speedup_{overall} = \frac{Tempo_exec_{antigo}}{Tempo_exec_{novo}} = \frac{1}{\sum (f_i / s_i)}$$

f_i - frações com melhoria s_i
 s_i - *speedup* de cada fração

Ex.1: Se 10% de um prog executa 90x mais rápido, então

Overall speedup = 1.11

Ex.2: Se 90% de um prog executa 90x mais rápido, então

Overall speedup = 9.09

Paralelismo:
se $N_{proc} \equiv speedup$, trocar s_i por N_{proc}

