



Estrutura do tema Avaliação de Desempenho (IA32)

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

Optimizações no Gnu C Compiler (1) (em <http://gcc.gnu.org/online/docs/gcc-3.2/gcc/>)



Options That Control Optimization

These options control various sorts of optimizations:

- O
-O1
Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. (...) With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- O2
Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. (...) this option increases both compilation time and the performance of the generated code.
- O3
-O2 turns on all optional optimizations except for loop unrolling, function inlining, and register renaming.
- O0
Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions and -frename-registers options.
- Os
Do not optimize.
- Oz
Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.



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 optimização
 - lei de Amdahl
 - dependentes da hierarquia da memória
 - a localidade espacial e temporal dum programa
 - influência da *cache* no desempenho

Optimizações no Gnu C Compiler (2) (em <http://gcc.gnu.org/online/docs/gcc-3.2/gcc/>)



Optimizações para código com arrays e loops:

-funroll-loops

Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. -funroll-loops implies both -fstrength-reduce and -fcommon. This option makes code larger, and may or may not make it run faster.

-funroll-all-loops

Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. -funroll-all-loops implies the same options as -funroll-loops,

-fprefetch-loop-arrays

If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.

-fmove-all-movables

Forces all invariant computations in loops to be moved outside the loop.

-freduce-all-givs

Forces all general-induction variables in loops to be strength-reduced.



Optimizações para inserção de funções em-linha:

-finline-functions

Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared static, then the function is normally not output as assembler code in its own right.

-finline-limit=n

By default, gcc limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (ie marked with the inline keyword ...) n is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of n is 600. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code will be inlined (which presumably means slower programs).

Uso do code profiling (1)



Uso do GProf em 3 passos:

–compilar com indicação explícita (-pg)

- ex.: análise do combined_sum_int (vector com 10⁷ elementos)

`gcc -O2 -pg combined_sum_int.c -o comb1`

–executar o programa

`./comb1`

- vai gerar automaticamente o ficheiro gmon.out

–invocar o GProf para analisar os dados em gmon.out

`gprof comb1.exe [ > comb1.txt ]`

- análise parcial do ficheiro comb1.txt a seguir...



Acção

```
gcc -O2 -pg prog.c -o prog
./prog
```

- executa como habitual/, mas tb gera o ficheiro gmon.out

`gprof prog`

- GNU profiler: a partir de gmon.out gera informação que caracteriza o perfil do programa

Factos

- calcula (aproximadamente) o tempo gasto em cada função
- método para cálculo do tempo (mais detalhe adiante)
 - periodicamente (~ cada 10ms) interrompe o programa
 - determina que função está a ser executada nesse momento
 - incrementa o seu temporizador de um intervalo (por ex., 10ms)
- para cada função mantém ainda um contador (nº de vezes que foi invocada)

Uso do code profiling (2)



Análise da primeira parte de comb1.txt:

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative	self	total	time	seconds	seconds	calls	s/call	s/call	name
39.33	2.58	2.58							_mcount
38.57	5.11	2.53			20000000	0.00	0.00	0.00	get_vec_element
12.65	5.94	0.83							mcount
6.40	6.36	0.42			2	0.21	1.57		combine1
3.05	6.56	0.20			20000002	0.00	0.00	0.00	vec_length
0.00	6.56	0.00			2	0.00	0.00	0.00	access_counter
0.00	6.56	0.00			1	0.00	0.00	0.00	get_counter
0.00	6.56	0.00			1	0.00	0.00	0.00	new_vec
0.00	6.56	0.00			1	0.00	0.00	0.00	start_counter



Análise em árvore da execução do prog. (em comb1.txt):

index	% time	self	children	called	name
[1]	100.0	0.42	2.73	2/2	main [2]
		2.53	0.00	2	combine1 [1]
		0.20	0.00	20000000/200000002	get_vec_element [3]
					vec_length [4]

[2]	100.0	0.00	3.15		<spontaneous>
		0.42	2.73	2/2	main [2]
		0.00	0.00	1/1	combine1 [1]
		0.00	0.00	1/1	new_vec [11]
		0.00	0.00	1/1	start_counter [12]
		0.00	0.00	1/1	get_counter [10]

[3]	80.3	2.53	0.00	20000000/200000000	combine1 [1]
		0.20	0.00	20000000	get_vec_element [3]

[4]	6.3	0.20	0.00	200000002/200000002	combine1 [1]
		0.00	0.00	20000002	vec_length [4]

[9]	0.0	0.00	0.00	1/2	start_counter [12]
		0.00	0.00	1/2	get_counter [10]
		0.00	0.00	2	access_counter [9]
...					

Lei de Amdahl

O ganho no desempenho – *speedup* - obtido com a melhoria do tempo de execução de uma parte do sistema, está limitado pela fracção de tempo que essa parte do sistema pode ser utilizada.

$$\text{Overall speedup} = \frac{\text{Tempo_exec}_{\text{antigo}}}{\text{Tempo_exec}_{\text{novo}}} = \frac{1}{(1 - f) + f / s}$$

em que **f** – fracção de um programa que é melhorado,
s – *speedup* da parte melhorada

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



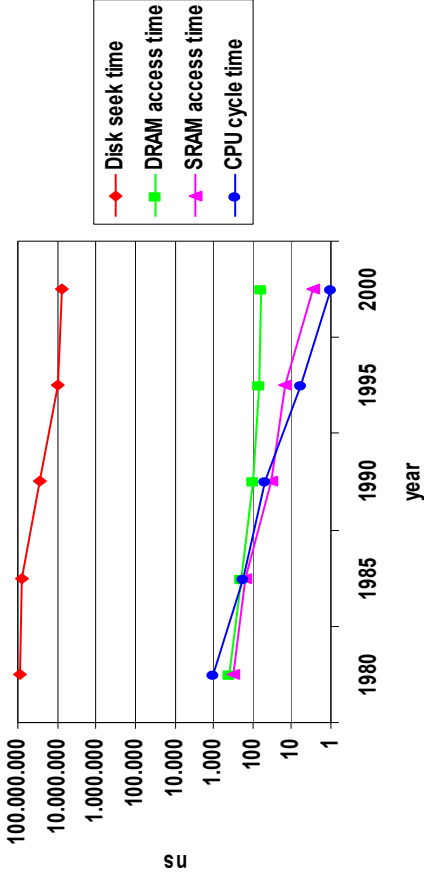
Vantagens

- ajuda a identificar os gargalos de desempenho
- particularmente útil em sistemas complexos com muitos componentes

Limitações

- apenas analisa o desempenho para o conjunto de dados de teste
- a metodologia de medição de tempos é rudimentar
 - apenas usável em programas com tempos de exec > 3 seg

Velocidade do CPU versus memória:
a diferença aumenta



Princípio da Localidade:

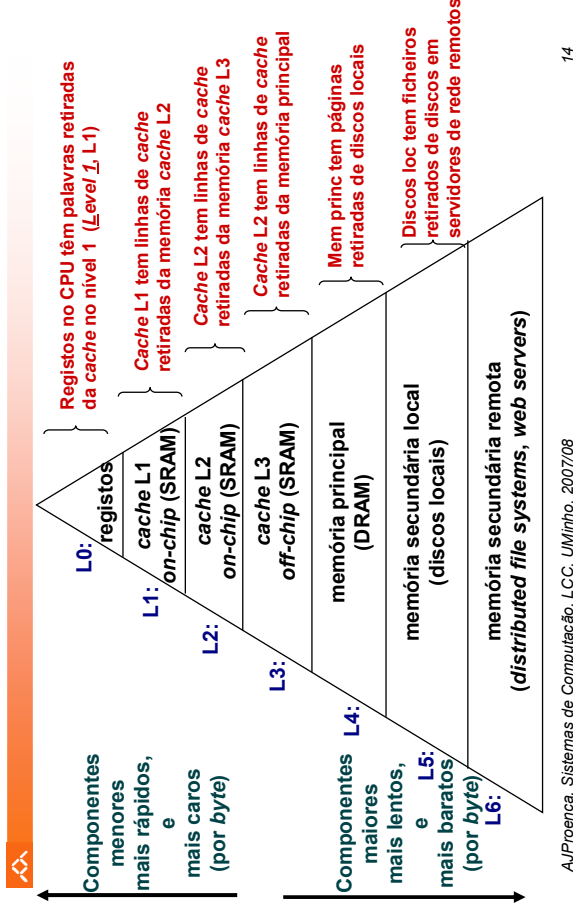
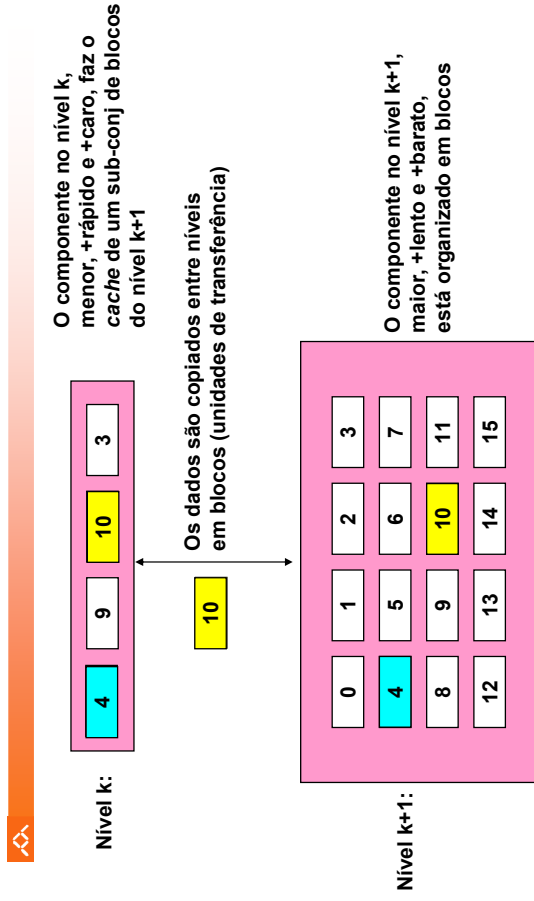
- programas tendem a reusar dados e instruções próximos daqueles que foram recentemente usados, ou que foram recentemente referenciados por eles
- **Localidade Espacial**: itens em localizações contíguas tendem a ser referenciados em tempos próximos
- **Localidade Temporal**: itens recentemente referenciados serão provavelmente referenciados no futuro próximo

Exemplo da Localidade :

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Dados**
 - os elementos do array são referenciados em instruções sucessivas: **Localidade Espacial**
 - a variável `sum` é acessada em cada iteração: **Localidade Temporal**
- **Instruções**
 - as instruções são acessadas sequencial/: **Localidade Espacial**
 - o ciclo é repetidamente acessado: **Localidade Temporal**

A cache numa hierarquia de memória: introdução



A cache numa hierarquia de memória: conceitos

Um programa pede pelo objecto d , que está armazenado num bloco b

Cache hit

- o programa encontra b na cache no nível k . Por ex., bloco 14

Cache miss

- b não está no nível k , logo a cache do nível k deve buscá-lo do nível $k+1$. Por ex., bloco 12
- se a cache do nível k está cheia, então um dos blocos deve ser substituído (retirado); qual?

- **Placement policy**: onde colocar o novo bloco? Por ex., $b \bmod 4$
- **Replacement policy**: que bloco deve ser retirado? Por ex., LRU



Miss Rate

- percentagem de referências à memória que não tiveram sucesso na cache (*misses* / acessos)
- valores típicos:
 - 3-10% para L1
 - pode ser menor para L2 (< 1%), dependendo do tamanho, etc.

Hit Time

- tempo para a *cache* entregar os dados ao processador (inclui o tempo para verificar se a linha está na *cache*)

- valores típicos :

- 1 ciclo de *clock* para L1
- 3-8 ciclos de *clock* para L2

Miss Penalty

- tempo extra necessário para ir buscar uma linha após *miss*
 - tipicamente 25-100 ciclos para aceder à memória principal



Referenciar repetidamente uma variável é positivo!

(**localidade temporal**)

Referenciar elementos consecutivos de um *array* é positivo!

(**localidade espacial**)

Exemplos:

- *cache* fria, palavras de 4-bytes, blocos (linhas) de *cache* com 4-palavras

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = $1/4 = 25\%$

Miss rate = **100%**