***Análise do desempenho de execução de aplicações** (3)*
(adaptado das aulas 10 e 12 do Bryant)

- Análise de técnicas de optimização *(s/w)*
  - técnicas de optimização independentes da máquina
    - ponto da situação
    - optimizações efectuadas pelo `Gcc`
    - identificação dos "gargalos" de desempenho
      - *program profiling*
      - uso dum *profiler* para apoio à optimização
      - lei de Amdahl
  - técnicas de optimização dependentes da máquina
    - dependentes do processador (já visto)
    - dependentes da hierarquia da memória
      - introdução à hierarquia de memória e à *cache*

Algun dos próximos slides foram retirados desta aula do Prof. Bryant

15-213
"The course that gives CMU its Zip!"

Code Optimization I: Machine Independent Optimizations Sept. 26, 2002

- Topics
  - Machine-Independent Optimizations
    - Code motion
    - Reduction in strength
    - Common subexpression sharing
  - Tuning
    - Identifying performance bottlenecks

`class10.ppt`

Machine-Independent Opt. Summary

Code Motion
- ***Compilers are good at this for simple loop/array structures***
- ***Don't do well in presence of procedure calls and memory aliasing***

Reduction in Strength
- ***Shift, add instead of multiply or divide***
  - *compilers are (generally) good at this*
  - *exact trade-offs machine-dependent*
- ***Keep data in registers rather than memory***
  - *compilers are not good at this, since concerned with aliasing*

Share Common Subexpressions
- ***Compilers have limited algebraic reasoning capabilities***

Important Tools

Measurement
- **Accurately compute time taken by code**
  - Most modern machines have built in cycle counters
  - Using them to get reliable measurements is tricky
- **Profile procedure calling frequencies**
  - **Unix tool `gprof`**

Observation
- **Generating assembly code**
  - Lets you see what **optimizations compiler** can make
  - Understand capabilities/limitations of particular compiler

Optimizações no *Gnu C Compiler*(1)

(de http://gcc.gnu.org/onlinedocs/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.
     -O2 turns on all optional optimizations except for loop unrolling,
     function inlining, and register renaming.
-O3
     Optimize yet more. -O3 turns on all optimizations specified by -O2 and
     also turns on the -finline-functions and -frename-registers options.
-O0
     Do not optimize.
-Os
     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.
```

Optimizações no *Gnu C Compiler*(2)

(de http://gcc.gnu.org/onlinedocs/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 -frerun-cse-after-loop. 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 no *Gnu C Compiler*(3)

(de http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/)

```
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).
```

Code Profiling

Augment Executable Program with Timing Functions

- Computes (approximate) amount of time spent in each function
- Time computation method
  - Periodically (~ every 10ms) interrupt program
  - Determine what function is currently executing
  - Increment its timer by interval (e.g., 10ms)
- Also maintains counter for each function indicating number of times called

Using

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

- Executes in normal fashion, but also generates file `gmon.out`

```
gprof prog
```

- Generates profile information based on `gmon.out`

Uso do profiling program (1)

Uso do `GProf` em 3 passos:

- **compilar com indicação explícita (`-pg`)**
  - ex.: análise do `combine1_sum_int` (vector com 10^7 elementos)

    ```
    gcc -O2 -pg combine1_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…

Uso do profiling program (2)

Análise da primeira parte de `comb1.txt`:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              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  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  vec_length
 0.00      6.56     0.00         2     0.00     0.00  access_counter
 0.00      6.56     0.00         1     0.00     0.00  get_counter
 0.00      6.56     0.00         1     0.00     0.00  new_vec
 0.00      6.56     0.00         1     0.00     0.00  start_counter
```

Uso do profiling program (3)

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

```
index  % time    self  children    called            name
                 0.42    2.73        2/2                 main [2]
[1]    100.0     0.42    2.73        2               combine1 [1]
                 2.53    0.00 20000000/20000000          get_vec_element [3]
                 0.20    0.00 20000002/20000002          vec_length [4]
-----------------------------------------------
                                                     <spontaneous>
[2]    100.0     0.00    3.15                        main [2]
                 0.42    2.73        2/2                 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]
-----------------------------------------------
                 2.53    0.00 20000000/20000000          combine1 [1]
[3]     80.3     2.53    0.00    20000000            get_vec_element [3]
-----------------------------------------------
                 0.20    0.00 20000002/20000002          combine1 [1]
[4]      6.3     0.20    0.00    20000002            vec_length [4]
-----------------------------------------------
                 0.00    0.00        1/2                 start_counter [12]
                 0.00    0.00        1/2                 get_counter [10]
[9]      0.0     0.00    0.00        2               access_counter [9]
…
```

Code Profiling Example

Task
- Count word frequencies in text document
- Produce sorted list of words from most frequent to least

Steps
- Convert strings to lowercase
- Apply hash function
- Read words and insert into hash table
  - Mostly list operations
  - Maintain counter for each unique word
- Sort results

Data Set
- Collected works of Shakespeare
- 946,596 total words, 26,596 unique
- Initial implementation: 9.2 seconds

Shakespeare's most frequent words

29,801	the
27,529	and
21,029	I
20,957	to
18,514	of
15,370	a
14010	you
12,936	my
11,722	in
11,519	that

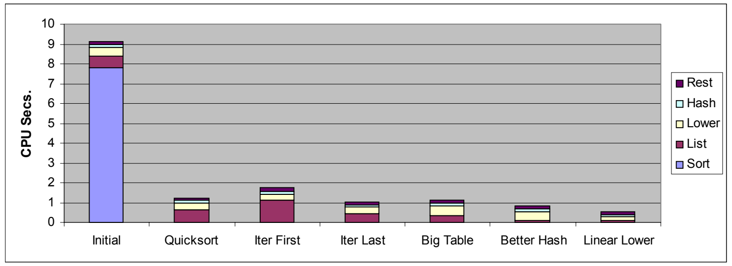
Profiling Results

```
%    cumulative   self              self     total
time    seconds   seconds    calls  ms/call  ms/call  name
86.60      8.21      8.21        1  8210.00  8210.00  sort_words
 5.80      8.76      0.55   946596     0.00     0.00  lower1
 4.75      9.21      0.45   946596     0.00     0.00  find_ele_rec
 1.27      9.33      0.12   946596     0.00     0.00  h_add
```

Call Statistics
- Number of calls and cumulative time for each function

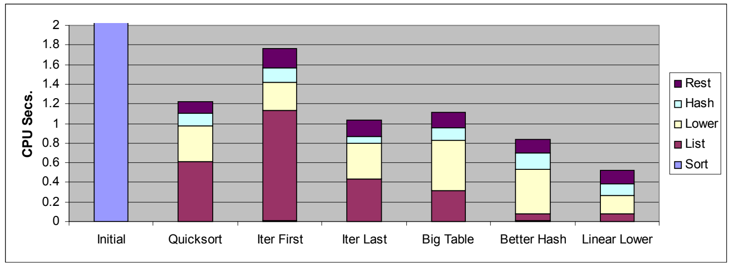
Performance Limiter
- Using inefficient sorting algorithm
- Single call uses 87% of CPU time

Code Optimizations



- First step: Use more efficient sorting function
- Library function `qsort`

Further Optimizations



- Iter first: Use iterative function to insert elements into linked list
  - Causes code to slow down
- Iter last: Iterative function, places new entry at end of list
  - Tend to place most common words at front of list
- Big table: Increase number of hash buckets
- Better hash: Use more sophisticated hash function
- Linear lower: Move `strlen` out of loop

Profiling Observations

Benefits
- Helps identify performance bottlenecks
- Especially useful when have complex system with many components

Limitations
- Only shows performance for data tested
- E.g., linear lower did not show big gain, since words are short
  - Quadratic inefficiency could remain lurking in code
- Timing mechanism fairly crude
  - Only works for programs that run for > 3 seconds

Lei de Amdahl

Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

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

where **f** - fraction of a program that is enhanced,
s - speedup of the enhanced portion

Ex.1
If 10% of a program runs 90 times faster, then

Overall speedup = 1.11

Ex.2
If 90% of a program runs 90 times faster, then

Overall speedup = 9.09

Os próximos slides foram retirados desta aula do Prof. Bryant

15-213
"The course that gives CMU its Zip!"

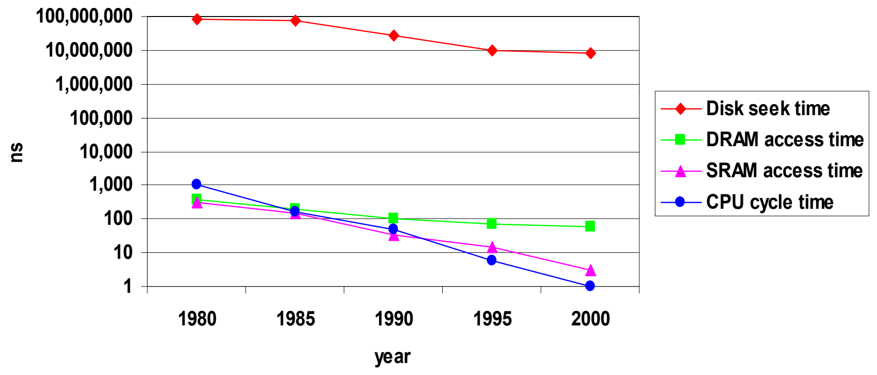
The Memory Hierarchy Oct. 3, 2002

Topics
- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

class12.ppt

The CPU-Memory Gap

- The increasing gap between DRAM, disk, and CPU speeds.



Locality

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: Recently referenced items are likely to be referenced in the near future.
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.

Locality Example:

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

- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference `sum` each iteration: **Temporal locality**
- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Memory Hierarchies

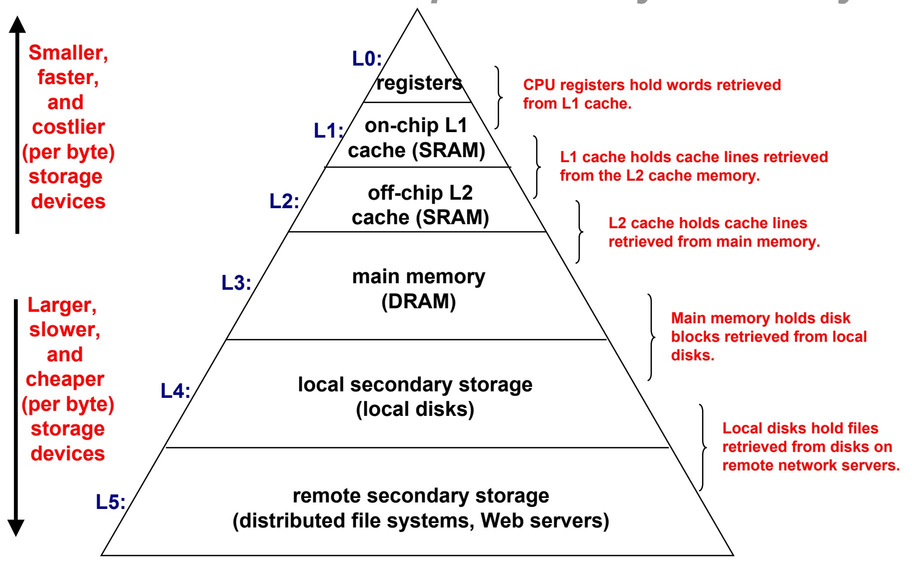
Some fundamental and enduring properties of hardware and software:

- Fast storage technologies cost more per byte and have less capacity.
- The gap between CPU and main memory speed is widening.
- Well-written programs tend to exhibit good locality.

These fundamental properties complement each other beautifully.

They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

An Example Memory Hierarchy

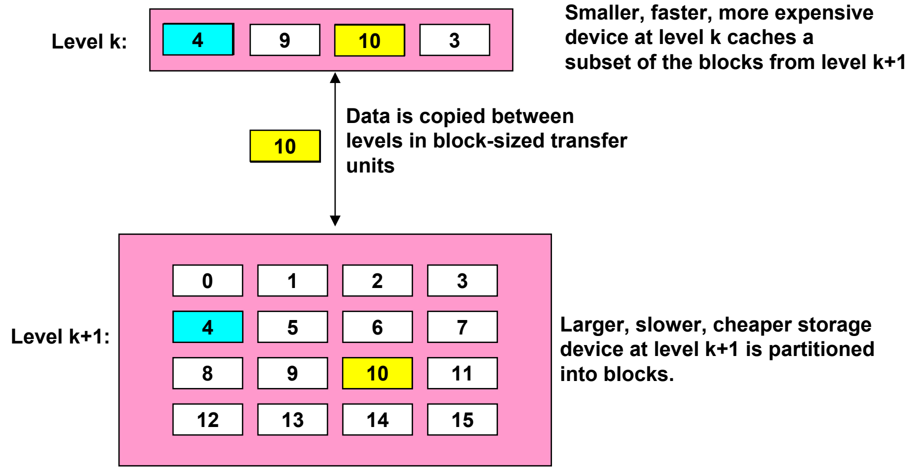


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
registers
CPU registers hold words retrieved from L1 cache.
L1:
on-chip L1 cache (SRAM)
L1 cache holds cache lines retrieved from the L2 cache memory.
L2:
off-chip L2 cache (SRAM)
L2 cache holds cache lines retrieved from main memory.
L3:
main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.
L4:
local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network servers.
L5:
remote secondary storage (distributed file systems, Web servers)

Caches

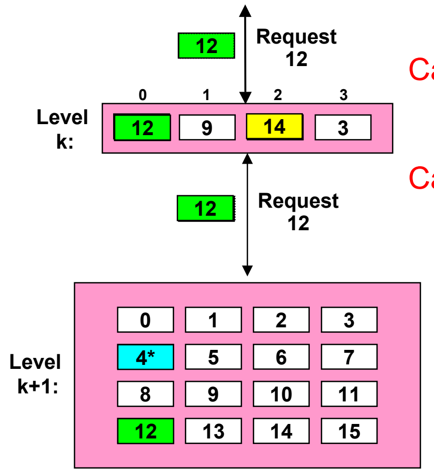
- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
  - Net effect: A large pool of memory that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Caching in a Memory Hierarchy



Level k:
4
9
10
3
Smaller, faster, more expensive device at level k caches a subset of the blocks from level k+1
10
Data is copied between levels in block-sized transfer units
Level k+1:
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Larger, slower, cheaper storage device at level k+1 is partitioned into blocks.

General Caching Concepts



- Program needs object d, which is stored in some block b.

Cache hit

- Program finds b in the cache at level k. E.g., block 14.

Cache miss

- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
  - Placement policy: where can the new block go? E.g., b mod 4
  - Replacement policy: which block should be evicted? E.g., LRU

Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time

- Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
  - 1 clock cycle for L1
  - 3-8 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
  - Typically 25-100 cycles for main memory

Writing Cache Friendly Code

Repeated references to variables are good
(temporal locality)

Reference array elements in succession are good
(spatial locality)

Examples:

- **cold cache, 4-byte words, 4-word cache blocks**

```
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;
}
```

Miss rate = 1/4 = 25%

```
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 = 100%