Software Optimization

Making your codes run more efficiently

Dr. Hyrum D. Carroll

Middle Tennessee State University

November 15 & 17, 2016

The Philosophy of Optimization

History

> *"As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise – By what course of calculation can these results be arrived at by the machine in the shortest time?"*

Analytical Engine: Passages from the Life of a Philosopher by CHARLES BABBAGE. CHAPTER VIII OF THE ANALYTICAL ENGINE, 1864

The Philosophy of Optimization

Key Points

- A slow program which works is MUCH more valuable than a fast program which doesn't work.
- 80% of the execution time is spent in about 20% of the code (the Pareto Principle)
- 4% of the lines account for 50% of the execution time (Knuth, 1971)
- it is almost impossible to optimize as you program
- throughput is more important than code speed
- **optimization without performance goals is pointless**

Wallin's Second Law of Computing

A program can run arbitrarily fast as long as you don't care about the accuracy or correctness of the results.

The Philosophy of Optimization

Quotes

"If the development time saved by implementing the simplest program is devoted to optimizing the running program, the result will be a faster running program than one in which optimization efforts have been exerted indiscriminately as the program was developed."

Stevens 1981- [Quoted in "Code Complete" p 595]

Quotes

"The best is the enemy of the good."

Quoted in "Code Complete" p592

Optimization Targets

You can optimize by improving a number of different aspects of a code (again, from "Code Complete")

- hardware
- code compilation
- module and routine design
- operating system interactions
- code tuning

What is really possible?

A simple example

```
1 nsize = 8000;
2 irank = 0;
3 for i = 1: nsize
4   for j = 1: nsize
5     irank = irank + 1;
6     a(i,j) = irank;
7   end
8 end
```

What is really possible?

Results (real/user)

- skynet w/octave- 9530 seconds

What is really possible?

Results (real/user)

- skynet w/octave- 9530 seconds
- harlie - 17.841/1.506 seconds
  - The base run is on harlie using gfortran with no optimization.

What is really possible?

Results (real/user)

- skynet w/octave- 9530 seconds
- harlie - 17.841/1.506 seconds
  - The base run is on harlie using gfortran with no optimization.
- skynet-no mods - 1.385/1.120 seconds

What is really possible?

Results (real/user)

- skynet w/octave- 9530 seconds
- harlie - 17.841/1.506 seconds
  - The base run is on harlie using gfortran with no optimization.
- skynet-no mods - 1.385/1.120 seconds
- Loops reversed - 0.575/0.336 seconds

What is really possible?

Results (real/user)

- skynet w/octave- 9530 seconds
- harlie - 17.841/1.506 seconds
  - The base run is on harlie using gfortran with no optimization.
- skynet-no mods - 1.385/1.120 seconds
- Loops reversed - 0.575/0.336 seconds
- Adding -O2 - 0.341/0.104 seconds

What is really possible?

Results (real/user)

- skynet w/octave- 9530 seconds
- harlie - 17.841/1.506 seconds
  - The base run is on harlie using gfortran with no optimization.
- skynet-no mods - 1.385/1.120 seconds
- Loops reversed - 0.575/0.336 seconds
- Adding -O2 - 0.341/0.104 seconds
- Using ifort - 0.298/0.032 seconds

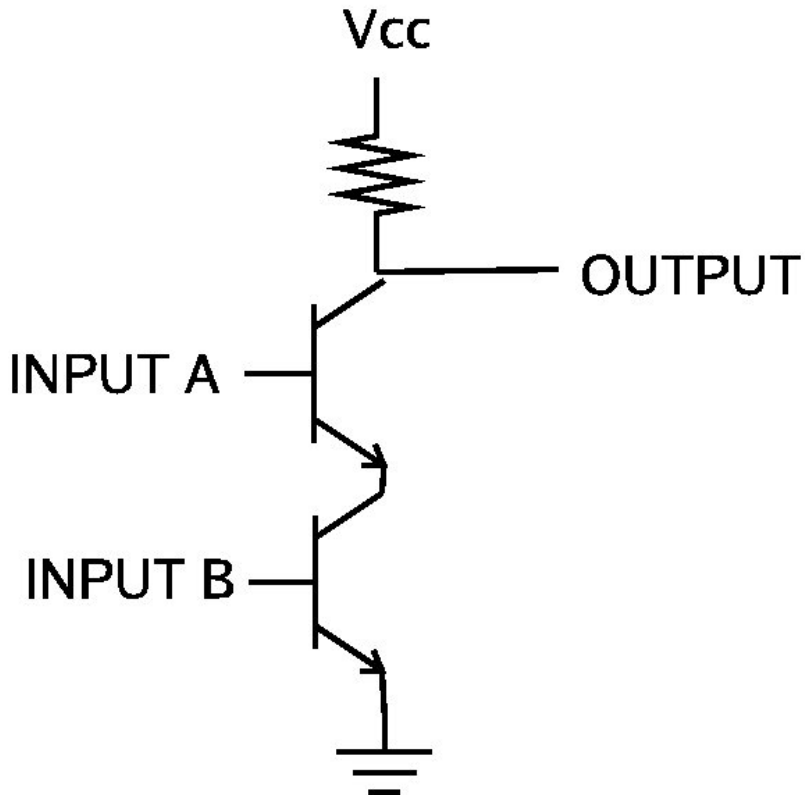
What is really possible?

Results (real/user)

- skynet w/octave- 9530 seconds
- harlie - 17.841/1.506 seconds
  - The base run is on harlie using gfortran with no optimization.
- skynet-no mods - 1.385/1.120 seconds
- Loops reversed - 0.575/0.336 seconds
- Adding -O2 - 0.341/0.104 seconds
- Using ifort - 0.298/0.032 seconds
- (ifort -without loop change = 1.256/0.964)

Real time change = 60 times faster
User time change = 45 times faster
Final optimization = 31980 times faster than octave

The NAND Gate



All computers can be built from NAND gates.

Basic Microprocessor Instructions

Microprocessor CPU's can only execute a limited number of functions.

- Load - load data from memory into the CPU
- Store - store data from the CPU into memory
- Branch or Jump - alter order of instruction execution
- Math and Logical Operations - internal operations within or between different words (shifts, adds, XOR, etc)

The specific operations are often more complex, such as "load from memory indirectly from a pointer in register X to register Y." However, they all fall into those simple categories.

The Magic in the Machine

Despite the underlying simplicity of how CPU's work, the actual implementations have become complex to increase performance. The big changes have been:

- CPU design
- Memory design and caching
- Compilers
- Operating Systems

Processors

CPU have greatly improved over the last 25 years. The changes in CPU design led to much higher system performance. As outlined by Dowd and Severance, the basic phases of this evolution are:

- Complex Instruction Set Computers
- Reduced Instruction Set Computers
- Super-scalar and super-pipelined processors
- Post-RISC Computers

An excellent review of optimization and high performance computing is in "High Performance Computing" by Kevin Dowd and Charles Severance (O'Reilly, 1998). Many of these notes are based on this book.

Complex Instruction Set Computers (CISC)

The first few generations of microprocessors all had CISC designs. The idea was simple - lots of instructions minimized memory and made the CPU's easier to use.
A rich set of instructions makes it easier to write complex algorithms. Complicated ideas could be concisely expressed. Coding these instructions into the chips hardware made sense, since programmers could more easily work within the system constraints. Most compilers did not take full advantage of the extra machine instructions, so they didn't fully optimize the performance of the high-level codes. Improving performance through clever compiling obfuscated the need for special instruction sets. Only one instruction could be acted on at any given time in these systems as well.

Reduced Instruction Sets (RISC)

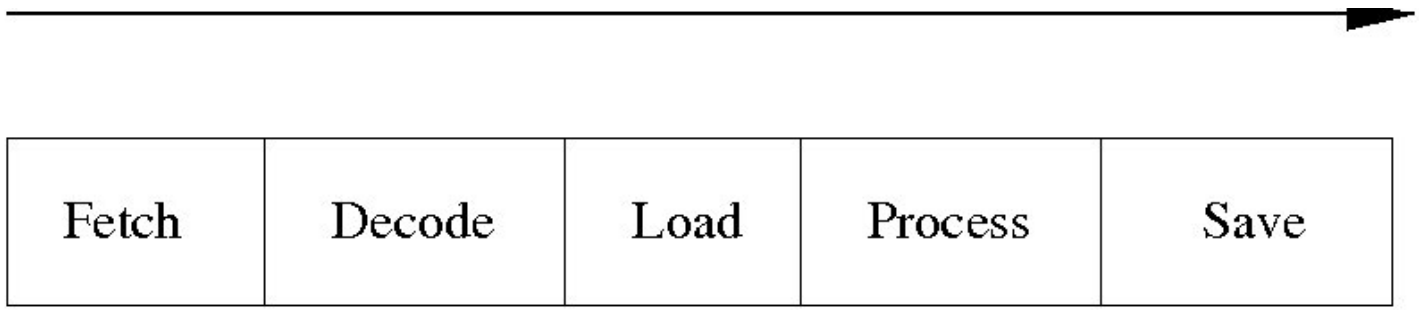
RISC machines have small, highly optimized instruction sets. However, the main reasons for the high performance of RISC machines is more complicated.
The common characteristics of RISC machines are:

- instruction pipelining
- uniform instruction length
- simple addressing modes
- load/store architecture
- delayed branching
- pipelining floating point numbers

Instruction Pipelining

Every instruction goes through a similar set of stages when it is processed. For example, in a given processor, the stages might be:

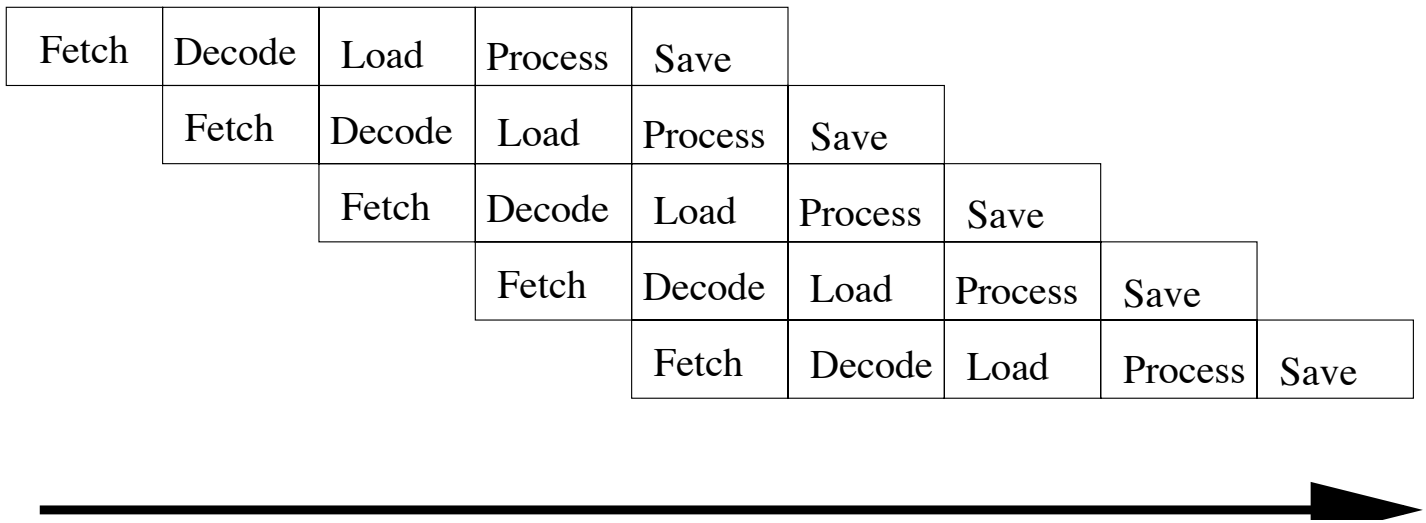
- fetching the instruction
- decoding the instruction
- loading the operands
- processing the instruction
- saving the results to memory



Instruction Pipelining

Simultaneous Execution

Since each of these steps is more or less independent from the other steps, it is possible to execute multiple instructions at the same time.



At any intermediate time slice, effectively five instructions are simultaneously executing.

Instruction Pipelining

Pipeline Efficiency

To make pipelines work effectively, three simple modifications are added to the internal architecture.

- **Uniform instruction Length:** all instructions have a uniform byte length. This means loading instructions is always the same, and decoding the instructions is straightforward.
- **Simple Address Modes:** only simple address modes are allowed. Complicated memory calculations are not allowed in any single program step.
- **Simple Load/Store Modes:** only simple load and store commands are allowed. There are no complicated multi-cycle load or store commands in the processor.

Instruction Pipelining

Pipeline Efficiency

All the modifications are done for the sake of efficiency. This has no real effect on the types of programs allowed in a high level language. It only impacts how the compiler translates the program into machine code.

Instruction Pipelining

Problems with Pipelines

Unfortunately, you don't always known what the next instruction is in real programs. If there is a branch which relies on the "current" system state, you can't predict which path to follow.
There are three approaches to this problem in normal RISC processors:

- treat the branch as a no-op and continue the execution (assume it will fail)
- guess the branch route based on recent behavior at this location
- begin to process the instructions after the branch

When Speculative Execution Fails

All of these work moderately well. However, all can fail in some cases. If the guess is wrong, the processor simply dumps the incorrectly executed instructions and starts filling up the pipeline again.

Super-RISC systems

First generation RISC machines have been improved upon in two ways.

- **Super-scalar processors** execute several instructions at the same time. This only works, of course, if the instructions are independent of each other. However, the compilers can figure this out. This is essentially a subset of parallel computing.
- **Super-pipeline processors** have enlarged pipelines. Instead of five stages, they might have ten to 80.

Super-scalar processors allow multiple "threads" to execute at the same time.

Post-RISC

Modern processors often are super-scalar. In order to keep several instructions executing at the same time, they often have to resort to some strange sounding tricks.

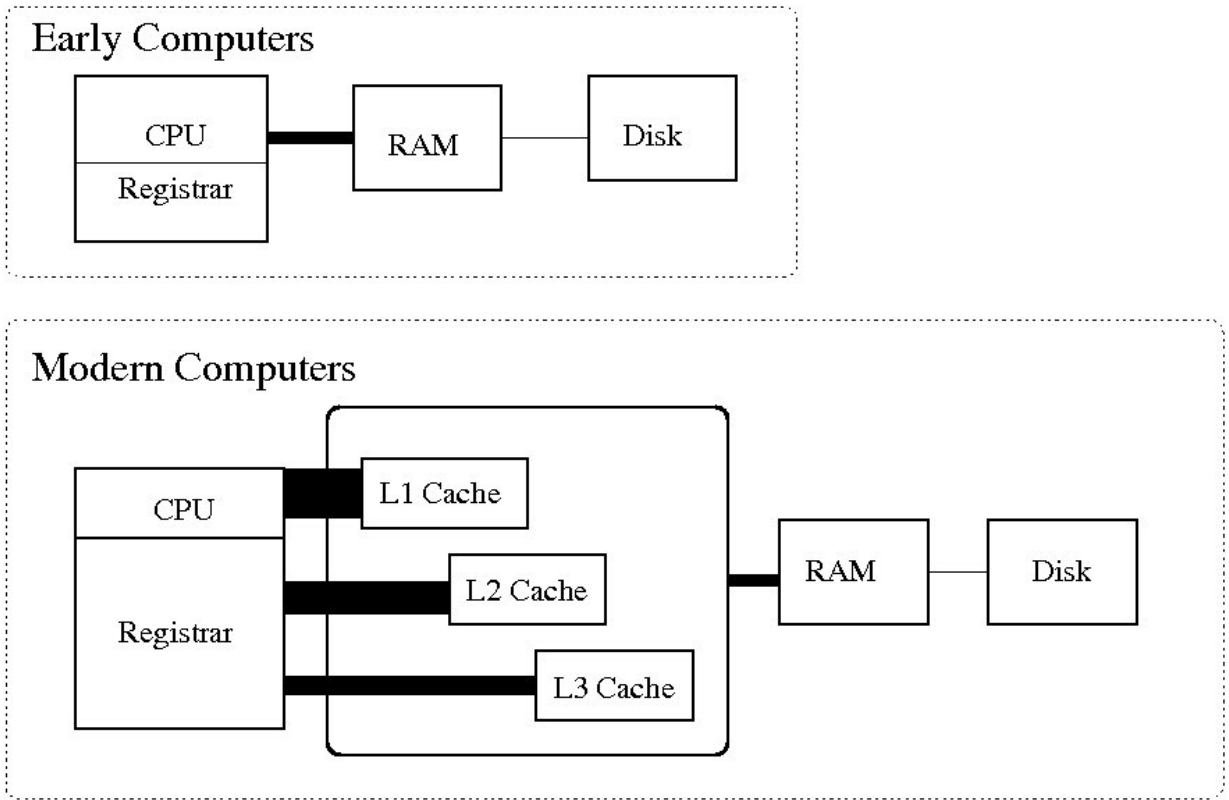
- **Out of order execution** makes sense in some cases. Even if instructions need to be dumped, you win if you guess correctly some or most of the time.
- **Speculative execution** also is used in modern processors. They literally do things they think you might want to have done. Again, guessing is effective if it is right some or most of the time.

Again, these are winning strategies if the guesses are helped by a smart compiler.

Floating Point Pipelines

Floating point pipelines are also EXTREMELY important in scientific computing.
The idea is the same as normal instruction pipelining. A set of floating point instructions is applied through a pipeline. Filling the floating point pipeline can greatly increase the speed of the instruction.
Unpipelined floating point operations can be executed, but usually MUCH slower than in fully pipelined machines.

Memory



The use of very high speed caches and large internal registers has significantly increased computer speeds.

Memory vs Cache

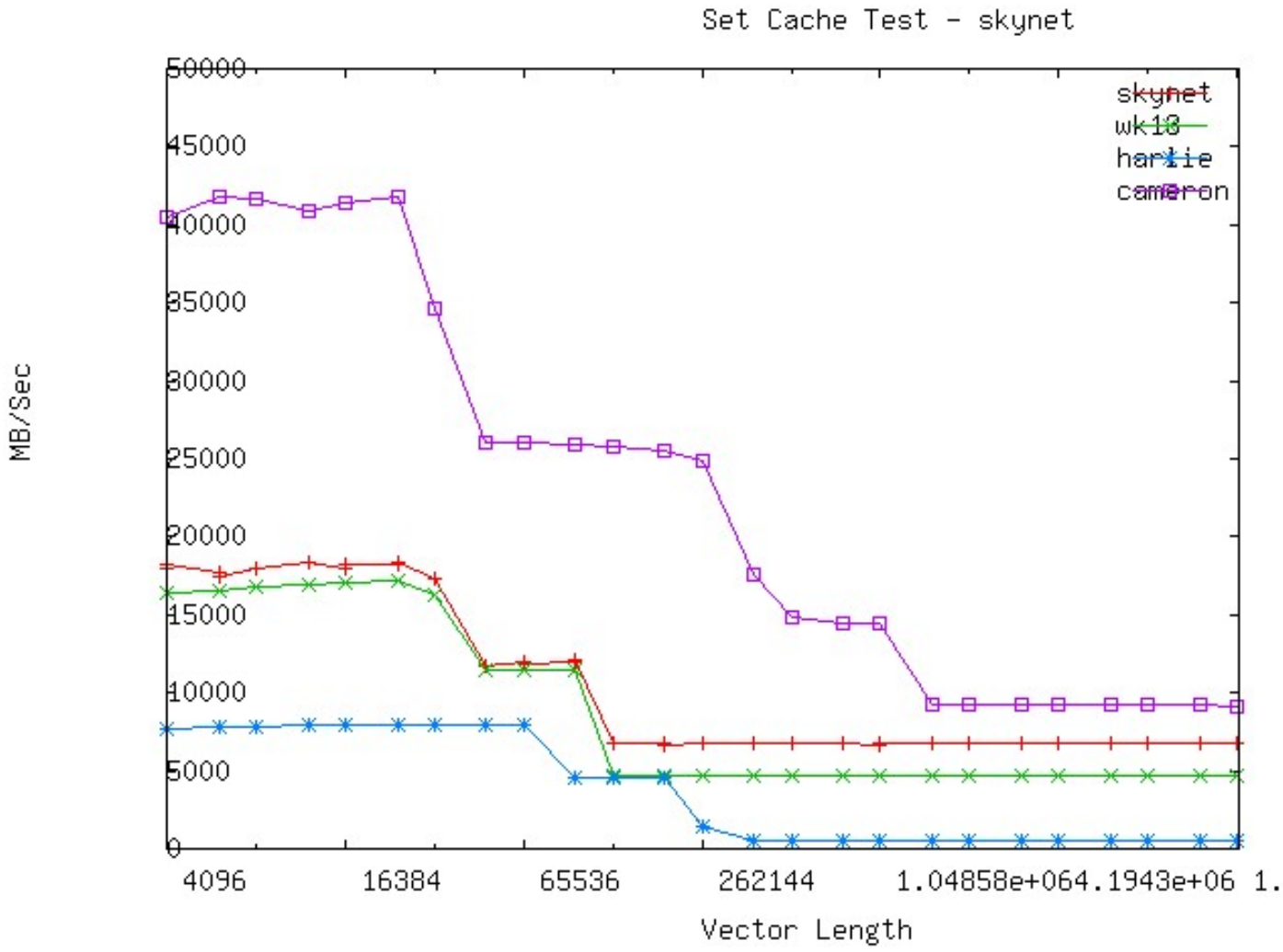
The use of very high speed caches and large internal registers has significantly increased computer speeds. Modern computers also use virtual memory for large jobs. This means that some of the programs storage is actually on disk, rather than in RAM.

Testing the Cache

```
1  double benchmark_cache_memory_set(REGISTER void *x,
       REGISTER long bytes, long *oloops, double *ous){
2      REGISTER long loops = 0;
3      FLUSHALL(1);
4      keepgoing = 1;
5      assert( signal(SIGALRM,handler) != SIG_ERR);
6      alarm(duration);
7      TIMER_START;
8      while (keepgoing) {
9          memset( x, 0xf0, bytes);
10         loops++;
11     }
12     TIMER_STOP;
13     fake_out_optimizations( x, bytes);
14     *ous = TIMER_ELAPSED;
15     *oloops = loops;
16     return((double)loops*(double)bytes);
17 }
```

From http://icl.cs.utk.edu/projects/llcbench/index.html
a benchmark by Philip J. Mucci

Testing the Cache Speeds



Testing the Cache Speeds

- Cameron - Spring 2010 - Intel 920 I7 - quad with hyper threading
  - L1 cache 40000 MB/sec
  - L2 cache 25000 MB/sec
  - L3 cache 15000 MB/sec
  - RAM 9100 MB/sec (1800 Mhz/ triple channel)
- Skynet - Summer 2008 - Intel 9420 - quad core
  - L1 cache 18000 MB/sec
  - L2 cache 12000 MB/sec
  - RAM 7000 MB/sec (1200 Mhz, 64 bit bus)
- Harlie - 2002ish, 2.4 Ghz dual core Athelon
  - L1 Cache 7700 MB/sec
  - L2 Cache 4500 MB/sec
  - RAM 500 MB/sec (100 Mhz, 32 bit bus)

Memory Quick Facts

- Reading one byte of memory from outside the cache involves the same addressing overhead as a larger read
- Reading subsequent bytes of memory is usually done in a single memory clock cycle
- Memory bandwidths are normally in the 800 Mhz range
- Seek times for disks are typically 8 milliseconds

Cache Principles

Memory is cached to avoid the cost of accessing RAM through a slow speed bus. Items cached support *memory locality*.
There are two types of locality

- spatial - regions in main memory that are physically close together
- temporally - regions in main memory that are accessed close to each other in time

Cache Replacement Management

When you need to access a block, a slot must be freed in the cache. The cleared block is chosen by several algorithms:

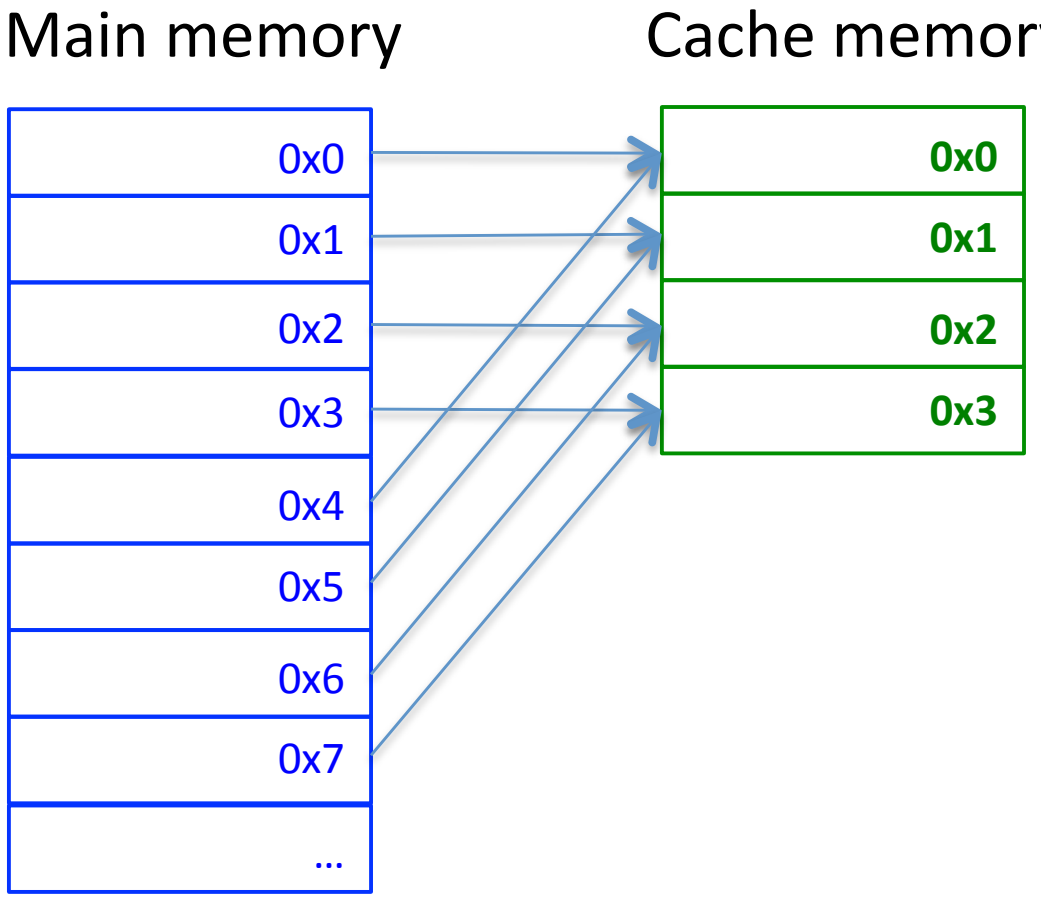
- LRU - Least Recently Used
- FIFO - First In, First Out
- LFU - Least Frequently Used
- Random

There are many algorithms used to find these blocks efficiently.

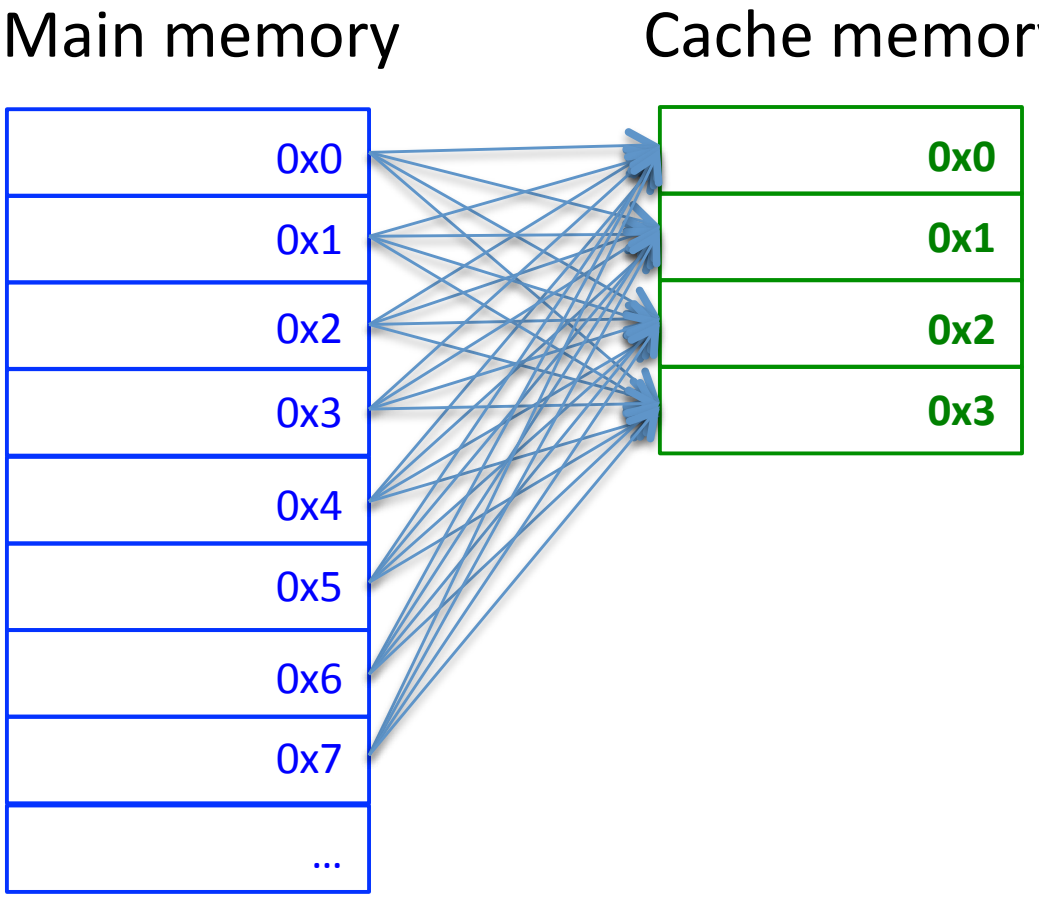
Internal Cache Structure

- Cache and memory is organized in 32 byte blocks.
- Cache memory is called slots.
  - A “tag” to associate a memory address
  - A “valid bit” marks a slot currently being executed.
  - A “dirty bit” marks blocks modified in the cache
- Main memory is organized in blocks.

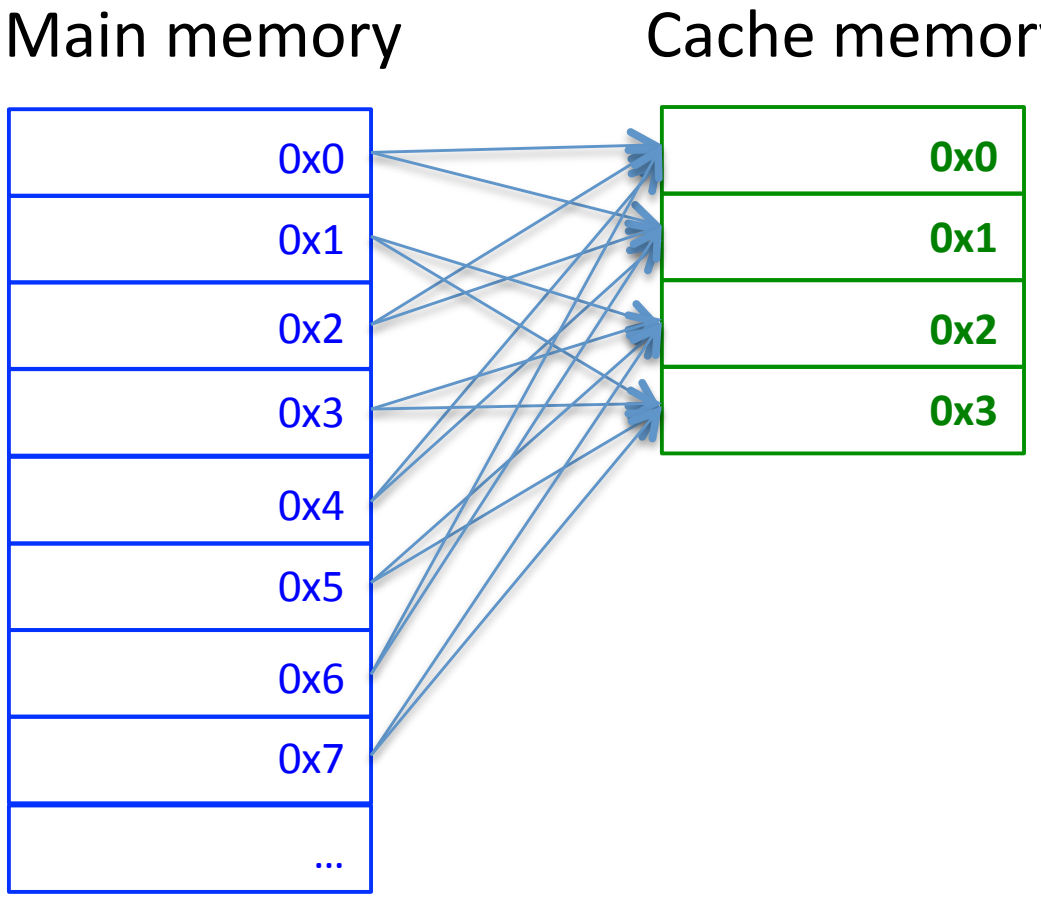
Direct Mapped Cache



Associative Mapped Cache



Set Associative Cache



Cache Associativity Models

- Direct Map Caching
  - Simple to implement
  - Array strides will have cache misses
- Associative Mapping Cache
  - Very flexible memory management
  - Difficult to search for memory in cache
  - Difficult to implement efficiently
- Set Associative Mapped Cache
  - Combines characteristics of both models
  - Used in nearly all modern computers

Cache Analogy

Assume you have a parking lot with 1,000 slots (addresses in cache) and 5,000 students (addresses in memory) that can park there.

- Direct Map Caching
  - Number slots 000 to 999 and use the first 3 digits of M#
  - Each person can only park in 1 slot
  - Who has an M# that starts with 001?
- Associative Mapping Cache
  - Slots are not numbered, you can park anywhere
- Set Associative Mapped Cache
  - Number slots 00 to 99 and use the last 2 digits of M#
  - Each person can park in one of 10 slots

(Source: http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/)

Cache Misses

Classification (Dr. Mark Hill)

- Compulsory misses - first access to a new memory site
- Capacity misses - not enough cache
- Conflict misses - could have been avoided if memory wasn't dumped earlier
  - mapping misses - dependent on association used in cache
  - replacement misses - misses that occur with the replacement policy

Hardware

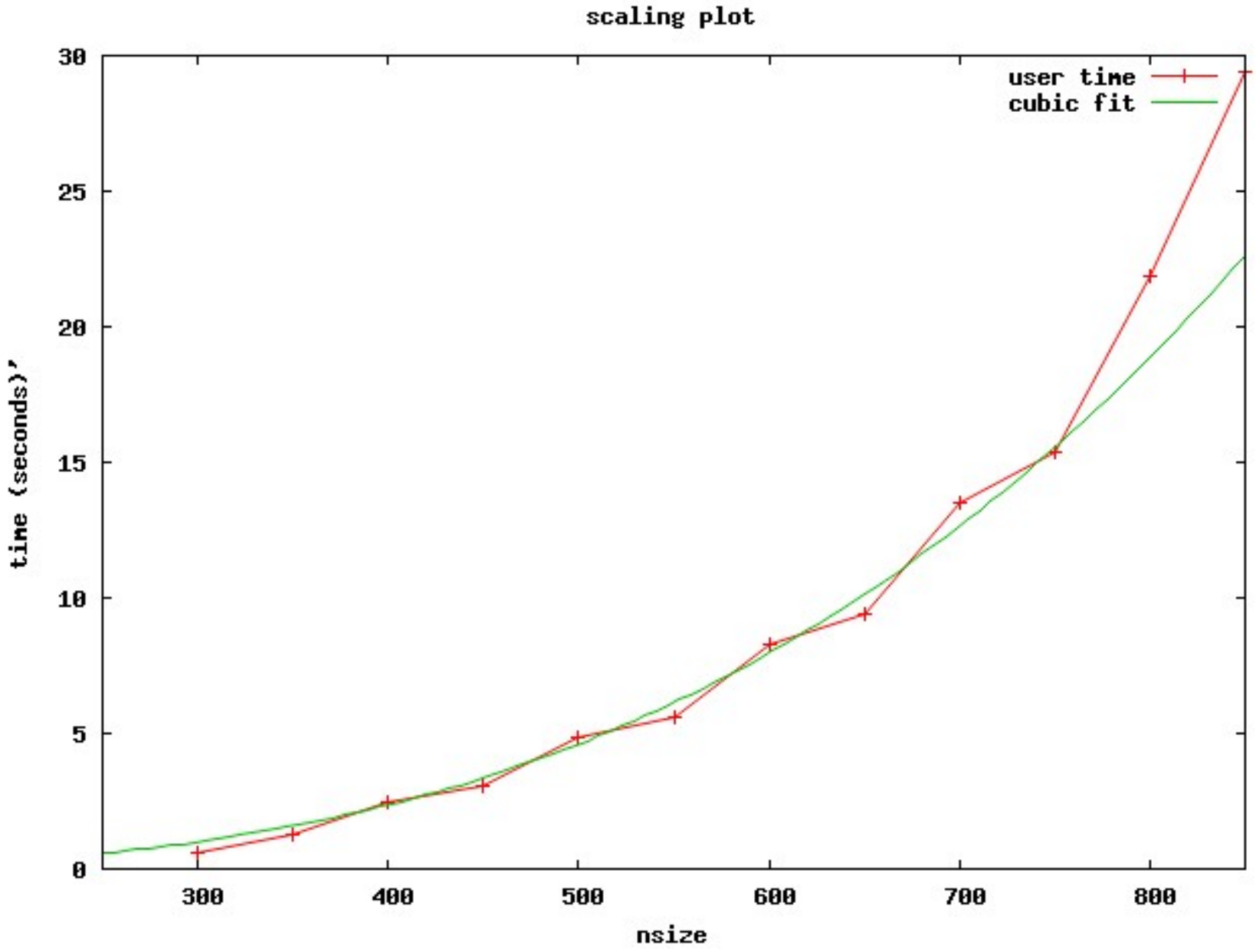
Page Faults

Scaling Example

```
1  integer (kind=4) :: i,j,k, irank
2  integer, parameter :: nsize = 800
3
4  real (kind=8), dimension(:,:,:), allocatable :: a
5  integer( kind=4) :: istat
6  allocate( a(nsize, nsize, nsize), stat=istat)
7
8  do i = 1, nsize
9    do j = 1, nsize
10     do k = 1, nsize
11       irank = irank+ 1
12       a(i,j,k) = irank
13     enddo
14   enddo
15 enddo
16
17 print*, a(nsize, nsize, nsize)
```

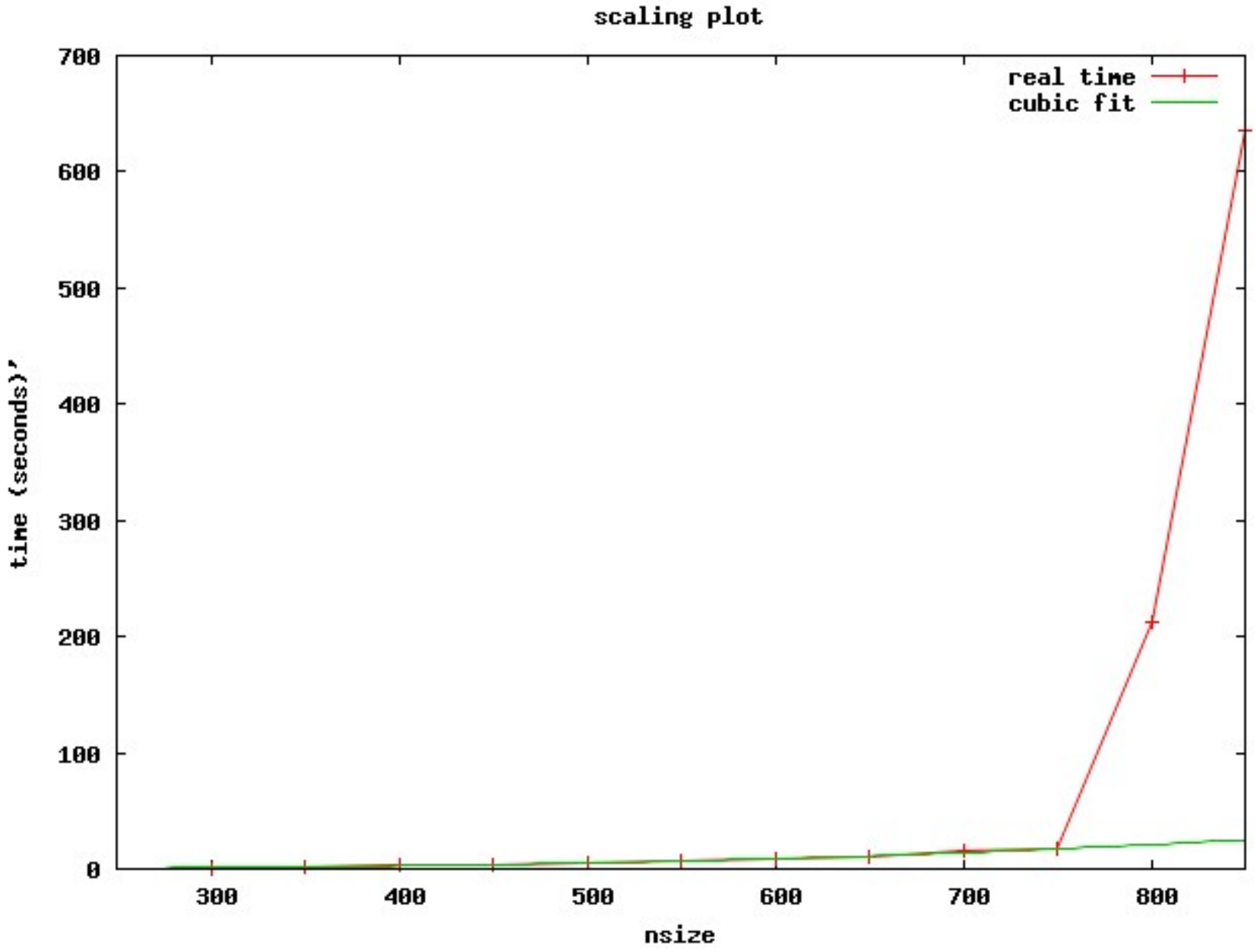
Scaling Example

User time - CPU time used



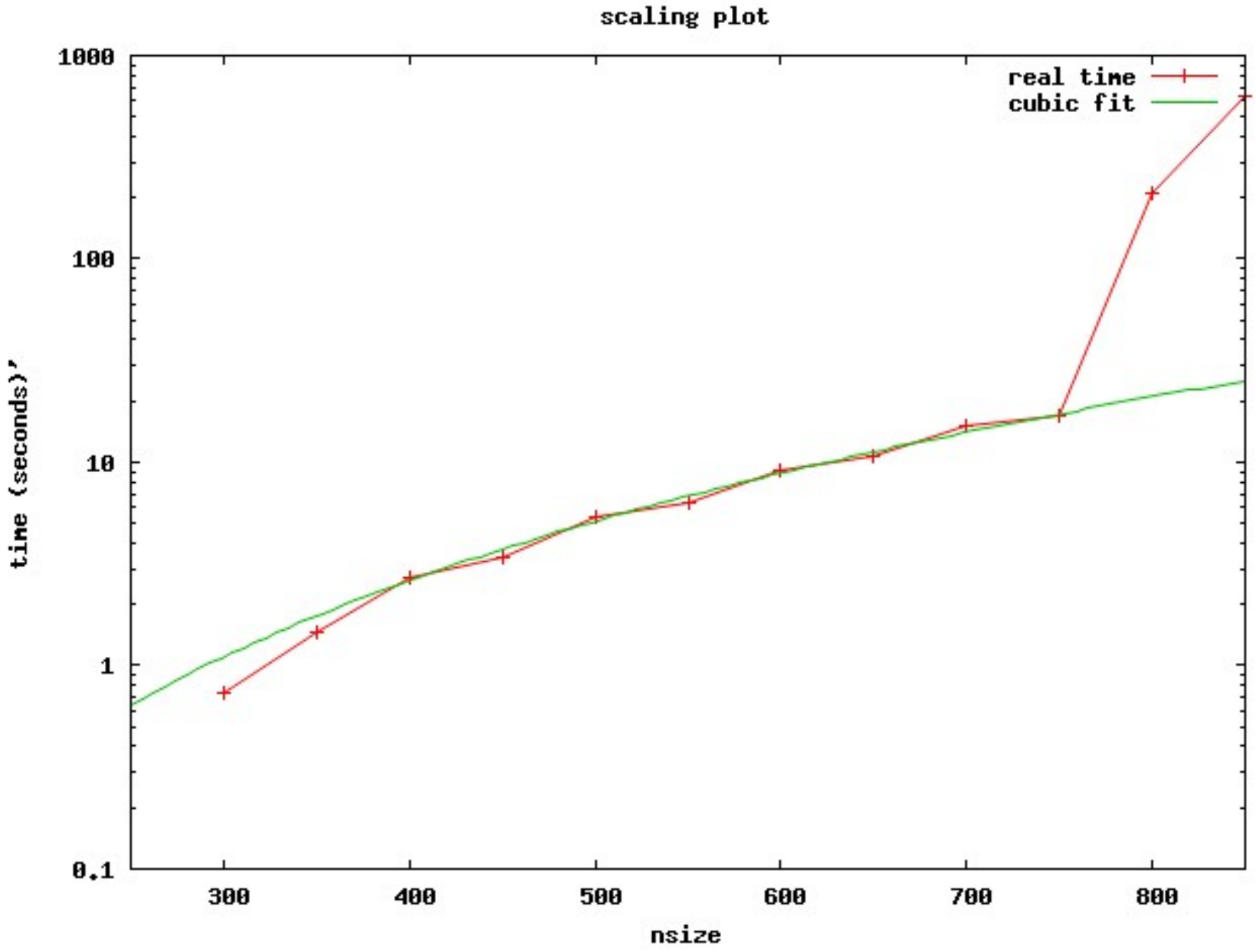
Scaling Example

Real time - clock time used



Scaling Example

Real time - clock time used



Scaling Example

The real time to execute the program went from 17 second to 212 seconds!
What happened?

Memory Usage

nsize = 750

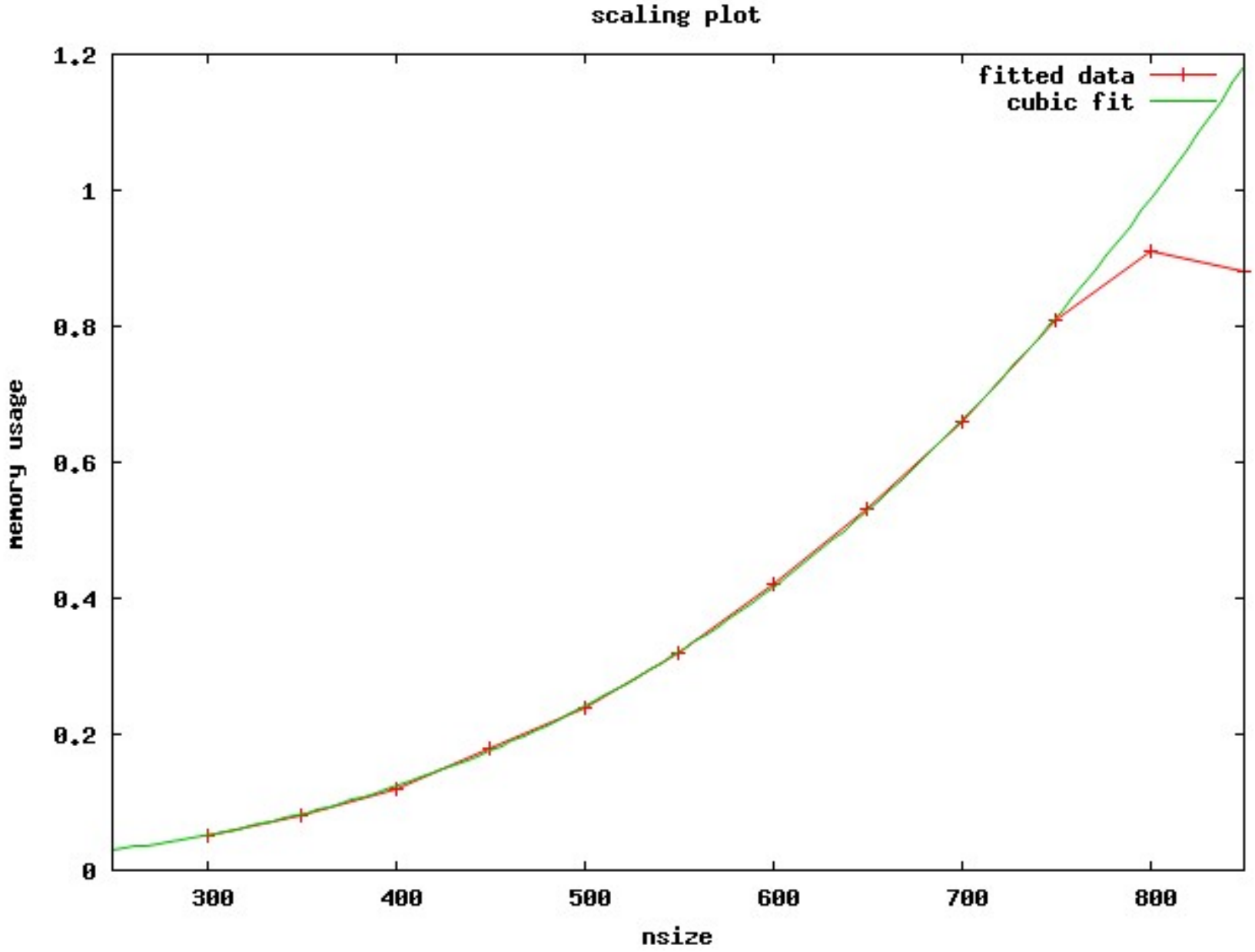
```
top - 15:31:46 up 7 days, 20:47,  4 users,  load average: 0.60, 0.57, 0
Tasks: 143 total,   3 running, 139 sleeping,   1 stopped,   0 zombie
Cpu(s): 22.7%us,  2.4%sy,  0.0%ni, 74.9%id,  0.1%wa,  0.0%hi,  0.0%si,
Mem:   4050856k total,  3530652k used,   520204k free,     1868k buffer
Swap:  7903972k total,  1009664k used,  6894308k free,    49356k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
20774 hcarrol   20   0 3229m 3.0g  452 R  100 76.7   0:05.48 a.out
 8039 root      20   0  221m  21m 3156 S    1  0.5 111:53.85 Xorg
10129 hcarrol   20   0  900m 122m 7108 R    1  3.1  51:33.53 firefox
 8236 hcarrol   20   0  328m  28m  22m S    1  0.7  23:35.60 compiz.rea
20659 hcarrol   20   0 18992  708  448 R    1  0.0   0:01.86 top
```

Memory Usage

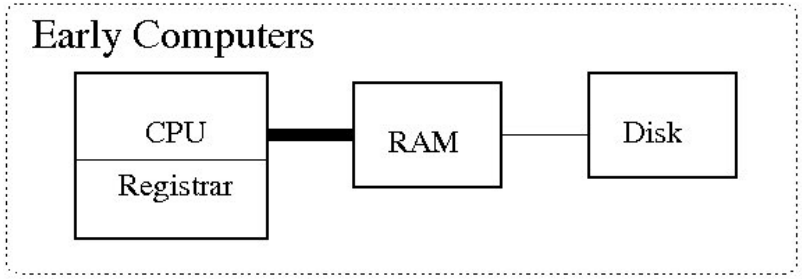
nsize = 800

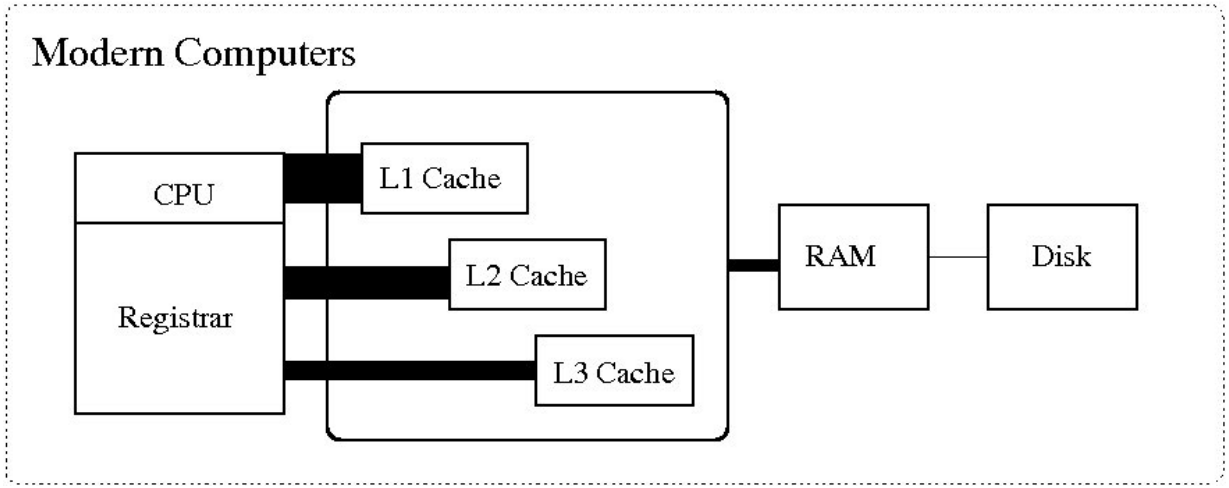
```
top - 15:32:37 up 7 days, 20:48,  4 users,  load average: 3.01, 1.12, 0
Tasks: 143 total,   1 running, 141 sleeping,   1 stopped,   0 zombie
Cpu(s): 0.4%us, 0.2%sy,  0.0%ni, 10.7%id, 88.5%wa,  0.1%hi,  0.1%si,  0
Mem:   4050856k total,  4025768k used,    25088k free,      264k buffer
Swap:  7903972k total,  1013004k used,  6890968k free,    22876k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
20781 hcarrol   20   0 3917m 3.5g  168 D    0 91.8   0:08.90 a.out
 8039 root      20   0  221m  15m 2288 S    0  0.4 111:54.61 Xorg
20659 hcarrol   20   0 18992  708  448 R    0  0.0   0:01.94 top
 8236 hcarrol   20   0  328m  27m  22m D    0  0.7  23:35.86 compiz.rea
10129 hcarrol   20   0  900m 108m 7172 D    0  2.7  51:34.15 firefox
```

Memory Usage



Cache Misses and Page Faults





Cache Misses and Page Faults

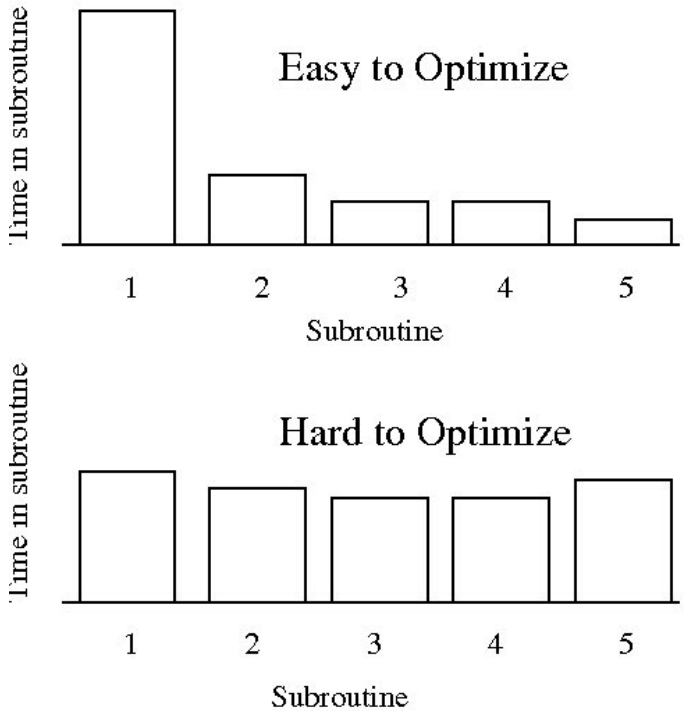
- Cache misses happen when you access memory outside the cache (it pulls it from the RAM)
- Page Faults happen when you request a page that is not in your RAM (it pulls it from the hard drive)

Timing & Profiling

Timing & Profiling

Understanding the resources used by your code is essential to improving its performance. You should NEVER use "One Mississippi, two Mississippi timing" to test your work. It is unreliable and leads to poor decisions about optimization.

Timing & Profiling



(Image from Code Complete, 2nd Edition)

The graphs above illustrate the challenge. If one routine is the problem, we might be able to improve code performance easily. In the second case, the problem is more challenging.

Time

The Unix command `time` displays:

- real (wall-clock)
- user (CPU-seconds dedicated to the program)
- sys (CPU-seconds used by the system on behalf of the program)

to STDERR after a program finishes executing

Example (in seconds):

```
$ time -p ScalingExample
   169353280.00000000
real 1.87
user 1.66
sys 0.19
```

Accessing the System Clock

`time` does "black box" timing

To measure *parts* of your code, use the system clock

`test_system_clock.f90`:

```
1 program test_system_clock
2   integer (kind=4) :: count, count_rate, count_max
3   !integer (kind=8) :: count, count_rate, count_max
4   integer :: i
5   do i = 1, 10000
6      call system_clock(count, count_rate, count_max)
7      write(*,*) "count: ", count, ", tics/sec: ",
           count_rate, ", count_max: ", count_max
8   enddo
9 end program test_system_clock
```

```
$ time -p test_system_clock
 ...
 count: 1103171202 , tics/sec: 1000 , count_max: 2147483647
 count: 1103171202 , tics/sec: 1000 , count_max: 2147483647
real 0.02
user 0.01
sys 0.00
```

Accessing the System Clock

`test_system_clock.f90` (using kind=8):

```
$ time -p test_system_clock
count: 1319657845741013000, tics/sec: 1000000000, count_max: 9223372036854775807
count: 1319657845741051000, tics/sec: 1000000000, count_max: 9223372036854775807
count: 1319657845741055000, tics/sec: 1000000000, count_max: 9223372036854775807
...
real 0.03
user 0.03
sys 0.00
```

Profiling

There are a number of ways to determine the performance and the performance bottlenecks within a code. The most commonly used method is profiling.
The basic steps are:

- create an application
- compile it with profiling flags
- run a set of numerical experiments
- examine the results from the performance measurements
- modify the program
- repeat

Profiling Methods

Profiling is a numerical experiment which tests the time spent within different sections of your code. Typically, profiling is done as a numerical experiment. The types of measurements made include

- **Program Counter Sampling** (pcsamp)- a measure of how often lines are used within codes.
- **Hardware Counter** (hwc) - a sample using the processor hardware counters.
- **CPU time** (usertime, totaltime) - a measure of how much time is spent in each routine
- **Ideal** (ideal) - a measurement made by counting the number of executions of each basic block and the ideal CPU time for each function.

More Profiling Issues

It is important to note that profiling IS an experiment. There are some pathological cases. If, for example, the sampling period is the same as the period in which a particular subroutine is accessed, it might be completely missed.
Make sure you use the right degree of resolution when you profile. Start at the subroutine level, and then move to the particular lines causing the problems.
Also, it is important to measure memory access as well as simply the CPU time. Commands such as `top`, `vmstat`, `ps`, and `size` can help with these issues.

Sample Profile

```
gfortran -pg idriver.f90 -o idriver
./idriver
gprof idriver
```

Sample Profile

Flat Profile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 40.66     60.88    60.88   200000     0.00     0.00  __integrator__diffeq
 30.86    107.10    46.22    50000     0.00     0.00  __integrator__rk4
  7.14    117.78    10.69   276900     0.00     0.00  __align_module__calculate_array_statistics
  6.86    128.05    10.27    92300     0.00     0.00  __align_module__map_points
  5.92    136.92     8.87    92300     0.00     0.00  __align_module__calculate_residual
  5.26    144.79     7.87    92300     0.00     0.00  __align_module__project_points
  3.11    149.44     4.65    50000     0.00     0.00  __init_module__take_a_step
  0.07    149.54     0.10      200     0.00     0.00  __setup_module__profile
  0.05    149.61     0.07   660300     0.00     0.00  __align_module__create_rotation_matrix
  0.03    149.66     0.05   184600     0.00     0.00  __parameters_module__print_genes
  0.02    149.69     0.03    92300     0.00     0.00  __align_module__create_transformation_matrix
```

(See www.cs.utah.edu/dept/old/texinfo/as/gprof.html#SEC5 for explanation)

Sample Profile

Call Graph (or Tree Profile)

```
index % time    self  children    called     name
[2]    100.0    0.00  149.75                main [2]
                0.00  149.75       1/1          MAIN__ [1]
-----------------------------------------------
                4.65  107.10   50000/50000      MAIN__ [1]
[3]     74.6    4.65  107.10   50000          __init_module__take_a_step [3]
               46.22   60.88   50000/50000      __integrator__rk4 [4]
-----------------------------------------------
               46.22   60.88   50000/50000      __init_module__take_a_step [3]
[4]     71.5   46.22   60.88   50000          __integrator__rk4 [4]
               60.88    0.00  200000/200000     __integrator__diffeq [5]
-----------------------------------------------
               60.88    0.00  200000/200000     __integrator__rk4 [4]
[5]     40.7   60.88    0.00  200000          __integrator__diffeq [5]
-----------------------------------------------
                0.02   37.85    7100/7100       MAIN__ [1]
[6]     25.3    0.02   37.85    7100          __init_module__check_fitness [6]
                8.87   10.69   92300/92300      __align_module__calculate_residual [7]
               10.27    0.00   92300/92300      __align_module__map_points [9]
                7.87    0.00   92300/92300      __align_module__project_points [10]
                0.03    0.07   92300/92300      __align_module__create_transformation_matrix [13]
                0.05    0.00  184600/184600     __parameters_module__print_genes [15]
                0.00    0.00    7100/7100       __align_module__align_to_separation [19]
```

(See www.cs.utah.edu/dept/old/texinfo/as/gprof.html#SEC6 for explanation)

Compilers

Compilers

Modern compilers are essential to high performance CPU's. Compilers have a number of stages they pass through translating programs into machine code. They are:

- **preprocessing** - adding definitions and include files
- **lexical analysis** - finding keywords, variables, constants, and operators
- **parsing** - moving the code into an intermediate representation
- **optimization** - simple code changes to improve efficiency
- **code generation** - creation of machine code

Compiler Optimizations

Compilers are getting much better, but the optimization changes they normally make are pretty simple.

- removal of inaccessible code
- removal of code that produces unused results
- simplification of constants
- constant folding (UN-redefined variables)
- common subexpression elimination
- mathematical simplifications
- removal of loop invariant code

Removal of Inaccessible Code

```
1  ...
2  do i = 1, 1000
3    j = i + 100
4  enddo
5
6  stop ! exits the program
7
8  do k = 1, 1000
9    s = sin(k)
10 enddo
```

The second loop would be eliminated since it can never be accessed

Removal of Code that Produces Unused Results

```
1 subroutine wasteOfTime
2   integer :: i,j
3
4   do i = 1, 1000
5     j = i + 100
6   enddo
7
8   return
9 end subroutine
```

Why bother calculating `j` if we're not going to use it?

Simplification of Constants

```
1 do i = 1, 1000
2   j = i + 100 * 15 + sin(3.1) * exp(4)
3 enddo
```

becomes

```
1 do i = 1, 1000
2   j = i + 204.63973
3 enddo
```

All the constant expressions are evaluated and formed into a single value

Constant Folding

```
1 k = 23
2 tmp1 = 100
3 do i = 1, 1000
4   j = i + tmp1 * k
5 enddo
```

becomes

```
1 k = 23
2 tmp1 = 100
3 do i = 1, 1000
4   j = i + 2300
5 enddo
```

two constants are combined into a single constant that is stored in a temporary variable

Common Subexpression Elimination

```
1  a = i * (b * c)
2  d = (b * c)  * 5
```

becomes

```
1  tmp1 = b*c
2  a = i * tmp1
3  d = tmp1 * 5
```

common subexpressions are identified and folded into temporary variables so they are only calculated once

Loop-Invariant Code

```
1 do i = 1, 1000
2   a = (b * c)
3   d = a * i
4 enddo
```

becomes

```
1 a = (b * c)
2 do i = 1, 1000
3   d = a * i
4 enddo
```

the portion of the code that doesn't depend on the loop is removed from the loop

Optimization "by Hand"

Compilers to a good job at improving code, but there are a few simple things you can do that can improve performance.

Procedure In-lining

There is overhead each time a function or routine is called. You can eliminate this overhead by "in-lining" the function or subroutine into the code. This can usually be done in one of three ways

- Specify the routines to in-line on the compiler line
- Putting in-line directives into the code
- Letting the compiler figure it out automatically

However, you can ALWAYS use C-Preprocessing Macros.
These can be used for debugging, conditional compilation, and for optimization through macro definitions of functions.

CPP Macros

The C-preprocessor (cpp) can be invoked with most compilers. It also can be used separately to "preprocess" source codes:

```
cpp -P filename > newfile
```

The most commonly used options for cpp are:

- **#include "fname"** - includes the contents of the file `fname` into the code
- **#define MACRO value** - defines a macro with a given value
- **#define VAR** - sets a variable definition
- **#undef VAR** - undefines a variable
- **#ifdef #endif** block of conditionally included code

Branches in Loops

Branches in loops break vector pipelines.
If at all possible, move conditionals outside of the loops.

OK:

```
1 do i = 1, 1000
2   if (i < 100) then
3     a(i) = 10
4   else
5     a(i) = 20
6   endif
7 enddo
```

Better:

```
1 do i = 1, 99
2     a(i) = 10
3 enddo
4 do i = 100, 1000
5     a(i) = 20
6 enddo
```

Minimize Page Faults and Cache Hits

When you have large strides in matrix and vector operations, the computer has to load additional information into the high level cache.

```
1 do i = 1, 9999
2   a(i) = a(10000 - i) * 5
3 enddo
```

The code above will load memory from two very different locations. In this case, this may not be able to be optimized very much.

Loop Unrolling

Because of the way vector processors work, it is sometimes better to unroll loops

```
1 do i = 1, 400000
2   a(i) = i * exp(i)
3 enddo
```

could be better written as

```
1 do i = 1, 400000,4
2   a(i) = i * exp(i)
3   a(i+1) = (i+1) * exp(i+1)
4   a(i+2) = (i+2) * exp(i+2)
5   a(i+3) = (i+3) * exp(i+3)
6 enddo
```

Eliminate Loops with Low Trip Counts

```
1 do i = 1, 3
2   a(i) = i * exp(i)
3 enddo
```

could better be written as

```
1   a(1) =  exp(1)
2   a(2) =  2*exp(2)
3   a(3) =  3*exp(3)
```

Column and Row Major

Memory Location	Fortran (Row Major)	C / C++ (Column Major)
1	a(1,1)	a[0][0]
2	a(2,1)	a[0][1]
3	a(3,1)	a[0][2]
⋮	⋮	⋮
n	a(n,1)	a[0][n-1]
n+1	a(1,2)	a[1][0]
n+2	a(2,2)	a[1][1]
⋮	⋮	⋮
2n	a(n,2)	a[1][n-1]
2n+1	a(1,3)	a[2][0]
2n+2	a(2,3)	a[2][1]
⋮	⋮	⋮

Rearranging Loop Order

C and Fortran programs have different orders for their arrays. By altering the order that indices are looped over, we can significantly improve the performance of codes.

```
do i = 1, 500
 do j = 1, 625
  a(i,j) = i * exp(j)
 enddo
enddo
```

will take a different amount of execution time than

```
do j = 1, 625
 do i = 1, 500
  a(i,j) = i * exp(j)
 enddo
enddo
```

(see `timing.f90` in this directory)

Changing Loop Order

```
1 do j = 1, 100
2   do i = 1, 5
3     total = total + a(i,j)
4   enddo
5 enddo
```

The inner loop has 6 tests. The outer loop has 100 tests = 600 total tests.

Changing Loop Order (cont'd)

becomes

```
1 do j = 1, 5
2   do i = 1, 100
3     total = total + a(i,j)
4   enddo
5 enddo
```

Note- the array order has been switched so that the we are looping over continuous memory. The inner loop executes 101 times. The outer loop executes 5 times - thus 505 tests in the loop.

Algorithms

- Better algorithms mean better performance
- Huge performance differences are sometimes possible
  - bubble sort vs quicksort
  - fast Fourier transform
  - hierarchical tree codes
  - hashes / dictionaries vs. arrays

Stop Testing When You Know the Answer

```
1 if (x > 10) and (x < 20)
```

replace with

```
1 if (x > 10) then
2     if (x < 20) then
```

or

```
1 if (x < 20) then
2     if (x > 10) then
```

Testing When You Found the Answer

```
found = .false.
do i = 1, large_number
   if( x(i) == target_value) then
      found = .true.
   end if
enddo
```

becomes

```
found = .false.
i = 1
do while( i <= large_number .and. .not. found)
   if( x(i) == target_value) then
      found = .true.
   end if
enddo
```

Testing by Order of Frequency

```
select case (number)
  case (rarely true)
    useful stuff done here
  case (sometimes true)
    something else useful done here
  case (usually true)
    normal thing done here
end select
```

Testing By Order of Frequency (cont'd)

replace with

```
select case (number)
  case (usually true)
    normal thing done here
  case (sometimes true)
    something else useful done here
  case (rarely true)
    useful stuff done here
end select
```

Sentinel Value

```
1  found = .false.
2  i = 1
3  do while (i <= large_number .and. .not. found)
4     if (value(i) == target_value ) then
5        found = .true.
6     else
7        i = i + 1
8     endif
9  enddo
10
11 if (found) then
12 ...
```

Sentinel Value

becomes

```
found = .false.
i = 1
value(large_number + 1) = target_value
do while ( value(i) != target_value)
   i = i + 1
enddo

if (i < large_value) then
...
```

Reduction of Multiplications in a Loop

```
1 increment = xmax /large_number
2 do i = 1, large_number
3   x(i) = i * increment
4 enddo
```

becomes

```
1 increment = xmax /large_number
2 sum = increment
3 do i = 1, large_number
4   x(i) = sum
5   sum = sum + increment
6 enddo
```

Caching Answers

If you have a routine that needs to recalculate the same answer over and over again, you can sometime gain efficiency by caching the answer.

```
1  real function do_calc(x)
2    if (x == old_x)
3      return (old_answer)
4    else
5      calculate some stuff...
6      old_x = x
7      old_answer = answer
8    endif
9
10   return
11 end function do_calc
```

Be Careful of System Routines

```
integer function log2_function( i)
  integer :: i
  log2_function = int( log(i) /log(2) )
end function
```

a better implementation - 30% faster

```
real, parameter :: log2 = 0.69314718
integer function log2_function( i)
  integer :: i
  log2_function = int( log(i) /log2 )
end function
```

Be Careful of System Routines (cont'd)

a much better implementation - 15 times faster

```
integer function log2_function( i)
  integer :: i
  if (i < 2) return 0
  if (i < 4) return 1
  if (i < 8) return 2
  if (i < 16) return 3
...
  if (i < 2147483648) return 30
```

Precompute Results

```
value = (1 + x)^3 * cos(x)
```

if x only has < 50 values, it is *probably* much more efficient to use

```
i = (x-xmin) /dx + 1
value = table(i)
```

Compare Performance of Similar Logic Structures

- if-then-else statements sometimes outperform or under perform compared to case statements
- substantial time differences ($\sim 50\%$) can be seen in some languages and some compilers

Strength Reduction

- replace multiplication with addition
- replace exponentiation with multiplication
- replace floating point with integers
- simplify trig when possible
- replace double precision with single
- replace integer multiplication by two with bit shifts

Reduce the Dimensions of Arrays

- array calculations take some time
- moving to a one-dimensional array reduces internal pointer calculations
- usually a small time saving

Fortran (column-major) example:
array2D(i,j) becomes array1D((j-1)*rowSize + i)

(see `oneDimensionArrayExample.f90` in this directory)

Minimize Array References

often done by compilers–

```
1 do j = 1, big_number
2   do i = 1, large_number
3       a(i) = a(i) * b(j)
4    enddo
5 enddo
```

becomes

```
1 do j = 1, big_number
2   tmp = b(j)
3   do i = 1, large_number
4       a(i) = a(i) * tmp
5    enddo
6 enddo
```

Exploit Algebraic Identities

```
1 if (sqrt(x1**2 + y1**2) < sqrt(x2**2 + y2**2))
```

vs

```
1 if (x1**2 + y1**2 <  x2**2 + y2**2)
```

can show huge CPU time differences.

Improving Speed and Size (Code Complete)

- substitute lookup tables for complicated logic
- jam (combining) loops
- use integers instead of floating point
- initialize data at compile time
- use constants of the correct type
- precompute results
- eliminate common subexpressions
- translate key routines into a low level language

Improve Speed at the Cost of Complexity

- stop testing when you know the answer
- order comparisons
- use lazy evaluations
- unswitch loops that contain if tests
- minimize work inside loops
- use sentinels in search loops
- put the busiest loop on the inside of nested loops

Improve Speed at the Cost of Complexity (cont'd)

- reduce strength of operations inside loops
- move multidimensional arrays to lower dimensions
- minimize array references
- augment data types with indices
- cache frequently used values
- exploit algebraic identities
- reduce strength in logical and mathematical expressions
- beware of system routines
- rewrite routines in-line