Optimization Tools

- philosophy of optimization
- some basic hardware
- compilers
- timing and profiling

The Philosophy of Optimization

"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"]

The Philosophy of Optimization

- 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)
- it is almost impossible to optimize as you program
- throughput is more important the code speed
- **optimization without performance goals is pointless**

Optimization Targets

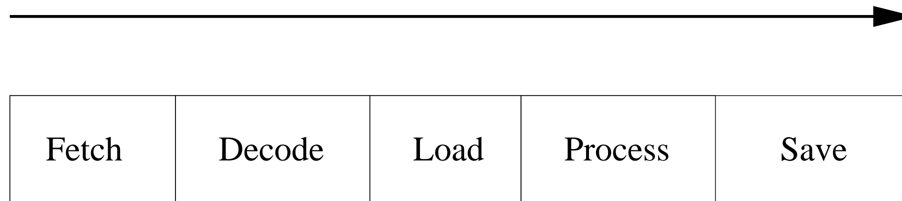
You can optimize by improving a number of different aspects of a code (again, from "Code Complete")

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

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



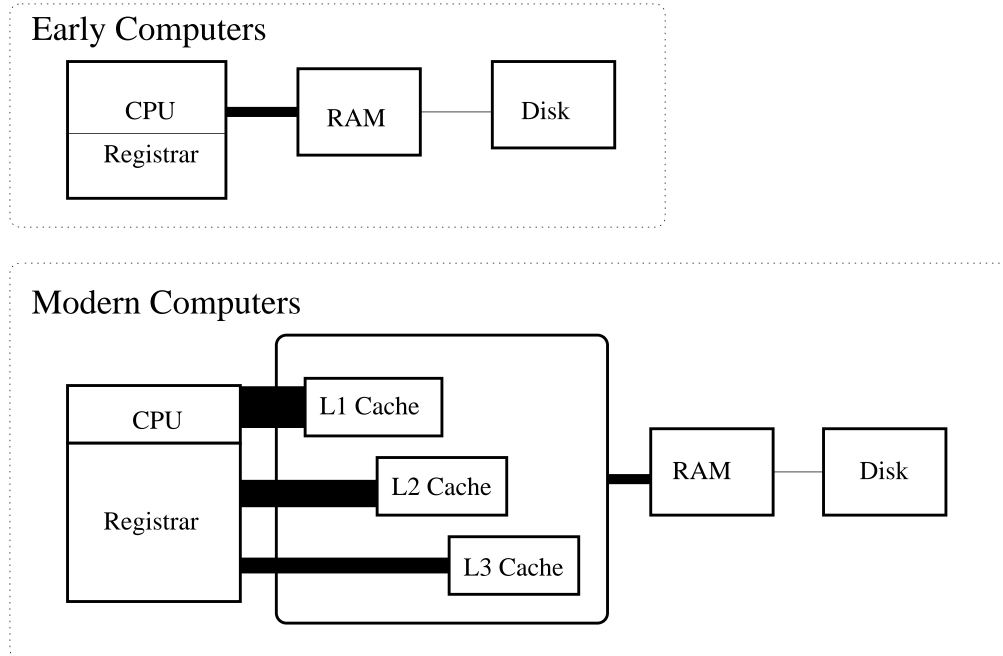
Problems with Pipelines

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

Memory



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

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.

Compilers

Modern compilers are essential to the 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
- simplification of inductive loops

Removal of Inaccessible Code

```
do i = 1, 1000
 j = i + 100
enddo

stop

do k = 1, 1000
 s = sin(k)
enddo
.
.
.
```

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

Removal of Code the Produces Unused Results

```
subroutine stupid
integer :: i,j

do i = 1, 1000
 j = i + 100
enddo

return
end subroutine
```

Why did you bother calculating j if you are not going to use it for something?

Simplification of Constants

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

becomes

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

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

Constant Folding

```
k = 23
tmp1 = 100
do i = 1, 1000
 j = i + tmp1 * k
enddo
```

becomes

```
k = 23
tmp1 = 100
do i = 1, 1000
  j = i + 2300
enddo
```

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

Common Subexpression Elimination

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

becomes

```
tmp1 = b*c
a = i * tmp1
d = tmp1 * 5
```

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

Loop-Invariant Code

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

becomes

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

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

Inductive Loop Simplification

```
do i = 1, 1000
 k = 34 * i + 35
 d = a * i
enddo
```

becomes k = 35

```
do i = 1, 1000
 k = k + 34
enddo
```

multiplications take more CPU time than additions, so we can simplify the internal 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

Every time you do a subroutine call, there is an associated overhead when you enter and exit a routine.

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 a 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.

```
do i = 1, 1000
  if (i < 100) then
    a(i) = 10
  else
    a(i) = 20
  endif
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.

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

would 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

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

could be better written as

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

Eliminate Loops with Low Trip Counts

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

could better be written as

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

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.

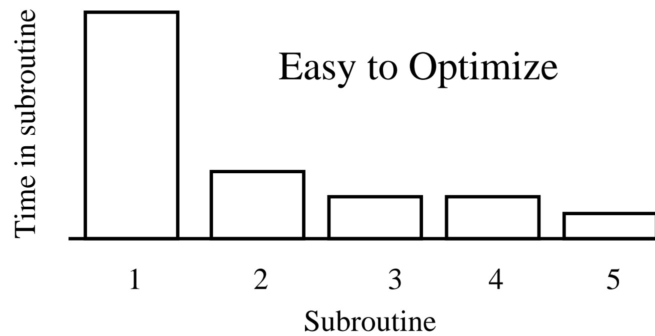
```
do i = 1, 500
 do j = 1,500
  a(i,j) = i * exp(j)
 enddo
enddo
```

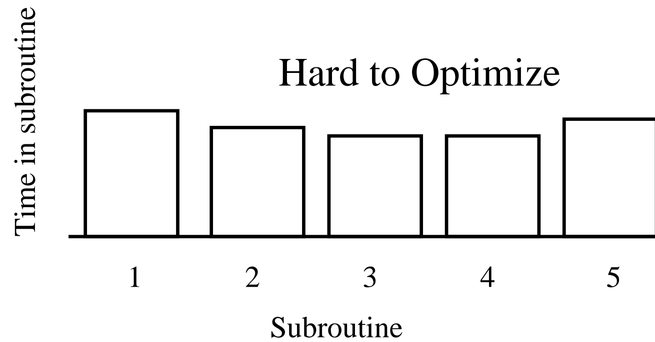
will take a different amount of execution time than

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

Timing & Profiling

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





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 simplest Unix system command is “time.” The “time” command is executed by typing “time program.”

```
harlie [~/code/SPH] (81) % time ktest
2.458u 0.050s 0:07.06 35.4% 0+0k 25+0io 14pf+0w
```

The columns are

- seconds of user time devoted to process
- seconds of system time devoted to process
- elapsed time
- percentage utilization
- average shared + unshared memory used
- number of block input + output operations
- page-faults + swaps

Accessing the System Clock

Time does not let you inside your code. You can't tell what part of your code is slow or what resources need to be optimized.

Clock call can let you time the internals of your code. The particular syntax varies between languages (call system_clock(cn,cr) for F90 and cn = etime(tarray) for C). However, the general idea is to query the system clock for the particular time that line has been executed.
If you print this information out in a loop, you might the result of :

```
1008409474,  10000
1008410450,  10000
1008411473,  10000
1008412816,  10000
```

The first number is the clock time, the second happens to be the cycles per second.

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
- run a set of numerical experiments
- examine the results from the performance measurements
- modify the program and 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.