Performance and Code Tuning

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
When to Optimize
When to optimize

- **Know your performance goals**
  - Don’t create a complex implementation if simple is fast enough
  - E.g. compute display frames 30/sec
  - Question: are you sharing the machine? If not, use it all!
  - Question: why do programmers use slow high level languages (Ruby, Haskell, Spreadsheets)?

- **Introduce complexity only where necessary for speed**

- **Don’t guess, measure!**

- **Goal: understand where the time is going and why before changing anything!**
Choose good algorithms

- Matters if processing *lots* of data
- Try to use existing implementations of good algorithms
- Otherwise, only use complex algorithms if data size merits doing so
- Leave tracks in comments or elsewhere if your algorithms won’t scale
Measuring Performance
What are we measuring

- **Time**
  - Elapsed time → "wall clock"
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0:01.25 94.4%     0+0k 0+0io 0pf+0w
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0:01.25 94.4% 0+0k 0+0io 0pf+0w
```
What are we measuring

- **Time**
  - Elapsed time \(\rightarrow\) “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0:01.25 94.4%     0+0k 0+0io 0pf+0w
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0.01.25 94.4% 0+0k 0+0io 0pf+0w
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

```
$ time quick_sort < rand1000000
1.116u 0.077s 0:01.25 94.4% 0+0k 0+0io 0pf+0w
```

This program took 1.25 seconds to complete.
The CPU was working on the program 94.4% of that time.
1.116 seconds of CPU time was spent in the user’s own code.
0.077 were spend in the operating system doing work for the user (probably code to support file input/output)
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems
What are we measuring

- **Time**
  - Elapsed time \(\rightarrow\) “wall clock”
  - CPU time from your code (user time)
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

Your program can also include timing checks *in the code* to measure specific sections. For more information, explore:

```
man 2 times
man 3 clock
man 2 getrusage
```
What are we measuring

- **Time**
  - Elapsed time → “wall clock”
  - CPU time from your code
  - CPU time from system work on your behalf
  - Waiting time: suggests I/O problems

- **I/O (amount read/written and rate at which I/O is done)**

- **Paging rates (OS writes data to disk when memory is over-committed)**

- **Contention from other users**
Performance Modeling
What is a performance model?

- A model you develop to *predict* where time will be spent by your program.

- To build a model, you need:
  - A clear understanding of how computer hardware performs.
  - A clear understanding (or guess) as to how compilers will translate your programs.
  - A high level picture of the flow of your program and where it will spend time.

- Your model will also depend on the size and nature of the data your program processes.

- *When you care about performance, build the best mental model you can, then measure to learn and update your model.*

The first few times you do this you’ll find that your predictive powers aren’t good...keep at it, and you’ll develop a good intuitive sense of why your programs perform as they do.
What is a performance model?

- A model you develop to predict where time will be spent by your program.
- To build a model, you need:
  - A clear understanding of how computer hardware performs.
  - A clear understanding (or guess) as to how compilers will translate your programs.
  - A high level picture of the flow of your program and where it will spend time.
- Your models will also depend on the size and nature of the data your program will process.
- When you care about performance, build the best mental model you can, then measure to learn and update your model.

The first few times you do this you’ll find that your predictive powers aren’t good…keep at it, and you’ll develop a good intuitive sense of why your programs perform as they do.

...even programmers who are skilled at modeling and prediction can make the mistake of optimizing “prematurely”...
Don't do premature optimization. Write the program well. Choose a good algorithm. Then *measure* performance, then improve the parts that most affect performance.
Hot spots

The bank robbery model of code tuning:

Q: Mr. Sutton, why do you rob banks?

A: That's where the money is.

Remember: 80% of the time is usually spent in 20% of the code.
Tune that 20% only!
Aside: more on Willie Sutton

In his partly ghostwritten autobiography, Where the Money Was: The Memoirs of a Bank Robber (Viking Press, New York, 1976), Sutton dismissed this story, saying:

"The irony of using a bank robber's maxim [...] is compounded, I will now confess, by the fact that I never said it. The credit belongs to some enterprising reporter who apparently felt a need to fill out his copy...

"If anybody had asked me, I'd have probably said it. That's what almost anybody would say...it couldn't be more obvious.

Source: http://en.wikipedia.org/wiki/Willie_Sutton#Urban_legend
Aside: more on Willie Sutton

In his partly ghostwritten autobiography, Where the Money Was: The Memoirs of a Bank Robber (Viking Press, New York, 1976), Sutton dismissed this story, saying:

“"The irony of using a bank robber's maxim [...] is compounded, I will now confess, by the fact that I never said it. The credit belongs to some enterprising reporter who apparently felt a need to fill out his copy..."

"If anybody had asked me, I'd have probably said it. That's what almost anybody would say...it couldn't be more obvious.

Or could it?

Why did I rob banks? Because I enjoyed it. I loved it. I was more alive when I was inside a bank, robbing it, than at any other time in my life. I enjoyed everything about it so much that one or two weeks later I'd be out looking for the next job. But to me the money was the chips, that's all.

Go where the money is...and go there often.

Source: http://en.wikipedia.org/wiki/Willie_Sutton#Urban_legende
Measurement and Analysis Techniques
Tools for understanding what your code is doing

- **Statistics from the OS:**
  - Overview of resources your program uses
  - Examples: `/usr/bin/time` command; `top`; etc. many others
  - Also look for tools that can report on I/O or network activity if pertinent
Tools for understanding what your code is doing

- **Statistics from the OS:**
  - Overview of resources your program uses
  - Examples: `/usr/bin/time` command; `top`; etc. many others
  - Also look for tools that can report on I/O or network activity if pertinent

- **Profilers:**
  - Typically bill CPU time (or other resources) to specific functions or lines of code
  - Examples: `gprof` (older, but simple & fast to use); `kcachegrind`
kcachegrind analysis of a quick sort

Main:
- partition 39.81%
- swap 20.16%

Callers:
- quick_sort 43.57%
- print_array 30.73%
- read_values 25.60%
- swap 21.09%
- strtol 11.86%
- getdelim 8.91%
- __IO_file_xsportn@1@GL 7.64%

Source:
- IO_file
- xsportm
- __strtol_l
- internal

Callee Map:
- getdelim
- malloc
- __dl_runtime_resolve (ld-2.5.so)
Tools for understanding what your code is doing

- **Statistics from the OS:**
  - Overview of resources your program uses
  - Examples: `/usr/bin/time` command; `top`; etc. many others
  - Also look for tools that can report on I/O or network activity if pertinent

- **Profilers:**
  - Typically bill CPU time (or other resources) to specific functions or lines of code
  - Examples: `gprof` (older, but simple & fast to use); `kcachegrind`

- **Instrument your code**
  - Add counters, etc. for code or I/O events that concern you
  - Use system calls to do microsecond timing of important sections of code
  - Print output at end, or (more sophisticated) implement real time monitoring UI
  - *Make sure you understand whether your instrumentation is affecting performance of your code!*
Tools for understanding what your code is doing

- **Statistics from the OS:**
  - Overview of resources your program uses
  - Examples: `/usr/bin/time` command; `top`; etc. many others
  - Also look for tools that can report on I/O or network activity if pertinent

- **Profilers:**
  - Typically bill CPU time (or other resources) to specific functions or lines of code
  - Examples: `gprof` (older, but simple & fast to use); `kcachegrind`

- **Instrument your code**
  - Add counters, etc. for code or I/O events that concern you
  - Use system calls to do microsecond timing of important sections of code
  - Print output at end, or (more sophisticated) implement real time monitoring UI
  - *Make sure you understand whether your instrumentation is affecting performance of your code!*

- **Read the generated assembler code**
Aside: why use C if we’re reading the Assembler?

1. You probably won’t be reading most of the assembler, just critical parts

2. Writing C and reading assembler is usually easier than writing asm

3. If you write in C and move to another machine architecture, your tuning modifications may no longer work well, but at least the program will run
Barriers to Performance
Performance killers

- **Too much memory traffic**
  - Remember: memory may be 100x slower than registers
  - Performance model: learn to guess what the compiler will keep in registers
  - Watch locality: cache hits are still much better than misses!
  - Note that large structures usually result in lots of memory traffic & poor cache performance

- **Malloc / free**
  - They are slow, and memory leaks can destroy locality

- **Excessively large data structures – bad locality**

- **Too many levels of indirection (pointer chasing)**
  - Can result from efforts to create clean abstractions!
Example of unnecessary indirection

- Compare:

```c
struct Array2_T {
    int width, height;
    char *elements;
};
...
struct Array2_T
 With:
 struct Array2_T {
    int width, height;
    char elements[];
};
...
struct Array2_T
```
Example of unnecessary indirection

- Compare:

```c
struct Array2_T {
    int width, height;
    char *elements;
};
...
a->elements[i]  /* same as *(*(a).elements + i) */
```

- With

```c
struct Array2_T {
    int width, height;
    char elements[];
};
...
a->elements[i]  /* *(*(a + offset(elements)) + i) */
```

Tricky C construct:
If you malloc extra space at the end of the struct, C will let you address those as array elements!
Performance killers

- **Too much memory traffic**
  - Remember: memory may be 100x slower than registers
  - Performance model: learn to guess what the compiler will keep in registers
  - Watch locality: cache hits are still much better than misses!
  - Note that large structures usually result in lots of memory traffic & poor cache performance

- **Malloc / free**
  - They are slow, and memory leaks can destroy locality

- **Excessively large data structures – bad locality**

- **Too many levels of indirection (pointer chasing)**
  - Can result from efforts to create clean abstractions!

- **Too many function calls**
Barriers to Compiler Optimization
What inhibits compiler optimization?

- **Unnecessary function calls**
  - For small functions, call overhead may exceed time for useful work!
  - Compiler can eliminate some not all

- **Unnecessary recomputation in loops (especially function calls)**
  - Bad: `for(i=0; i < array.length(); i++)` ← constant?
  - Better: `len = array.length();
    for(i=0; i < len; i++)`

- **Duplicate code:**
  - Bad: `a = (f(x) + g(x)) * (f(x) / g(x))`
  - Better: `ftmp = f(x);
    gtmp = g(x);
    a = (ftmp + gtmp) * (ftmp / gtmp)`

- **Aliasing** (next slide)
Aliasing: two names for the same thing

*Compare these functions, which are quite similar*

```c
int my_function(int a, int b) {
    b = a * 2;
    a = a * 2;
    return a + b;
}

int c = 3;
int d = 4;
result = my_function(c, d);

Result is: 14
```

```c
int my_function(int *a, int *b) {
    *b = (*a) * 2;
    *a = (*a) * 2;
    return *a + *b;
}

int c = 3;
int d = 4;
result = my_function(&c, &d);

Result is: 14
(c and d are updated)
```
Aliasing: two names for the same thing

```c
int my_function(int *a, int *b) {
  *b = (*a) * 2;
  *a = (*a) * 2;
  return *a + *b;
}

int c = 3;
result = my_function(&c, &c);
```

Correct result is: \(24\) (and \(c\) is left with value 12!)
int my_function(int *a, int *b) {
  *b = (*a) * 2;
  *a = (*a) * 2;
  return *a + *b;
}

int c = 3;
result = my_function(&c, &c);

Correct result is: 24 (and c is left with value 12!)
Aliasing

Even worse, if `my_function` is in a different source file, compiler has to assume the function may have kept a copy of `&c`...now it has to assume most any external function call could `c` (or most anything else!)

```c
int my_function(int *a, int *b) {
    *b = (*a) * 2;
    *a = (*a) * 2;
    return *a + *b;
}

int c = 3;
result = my_function(&c, &c);```
Optimization Techniques
Code optimization techniques

- Moving code out of loops
- Common subexpression elimination
  - When in doubt, let the compiler do its job
    - Higher optimization levels (e.g. gcc –O2)
- Inlining and macro expansion
  - Eliminate overhead for small functions
Macro expansion to save function calls

```c
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```c
int x = max(c,d);
```
Macro expansion to save function calls

```c
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```c
#define max(a,b) ((a) > (b) ? (a) : (b))
```

```c
int x = max(c++, d);
```

Expands directly to:

```c
int x = ((c) > (d) ? (c) : (d));
```

What’s the problem with:

```c
int x = max(c++, d); ??
```
Inlining: best of both worlds

```c
int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

```c
#define max(a,b) ((a) > (b) ? (a) : (b))
```

Now there’s no problem with:

```c
int x = max(c++, d);
```
Inlining: best of both worlds

```c
int max(int a, int b)
{
    return (a > b) ? a : b;
}

int x = max(c, d);
```

Most compilers do a good job anyway, but you can suggest which functions to inline.

Still, we mostly discourage explicit inline specifications these days, as compilers do a very good job without them!

```c
inline int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

Compiler expands directly to:

```c
int x = ((c) > (d) ? (c) : (d));
```
Code optimization techniques

- Moving code out of loops
- Common subexpression elimination
- When in doubt, let the compiler do its job
  - Higher optimization levels (e.g. gcc -O2)
- Inlining and macro expansion
  - Eliminate overhead for small functions
- Specialization
Specialization

static int
sum_squares(int x, int y)
{
    return ((x * x) + (y * y));
}

... 
int num1 = 7;
int num2 = 3;

sum = sum_squares(6, 3);

(6*6) + (3*3) = 45

gcc –O2 generates one instruction!: movl $45, %r9d
Specialization with header files – must use ‘static’

```c
static int
sum_squares(int x, int y)
{
    return ((x * x) + (y * y));
}
```

Without static declaration, `sum_square` is declared in duplicate in src1.o and src2.o

```c
#include "sum_squares.h"
sum = sum_squares(6, 3);
```

```c
#include "sum_squares.h"
sum = sum_squares(6, 5);
```

```bash
gcc -p program src1.o src2.o
```
Code optimization techniques

- Moving code out of loops
- Common subexpression elimination
- When in doubt, let the compiler do its job
  - Higher optimization levels (e.g. gcc –O2)
- Inlining and macro expansion
  - Eliminate overhead for small functions
- Specialization
- Optimizations often compound: inlining and specialization
- Don’t worry about asserts unless tests take lots of time
- Conditional compilation of debug code:

```c
#ifdef DEBUG
    can use gcc -DDEBUG to define the flag
```