Code Tuning

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
Today

- When & how to tune
- Factors that affect performance
- Barriers to performance
- Code Tuning
When to Optimize
When to optimize

- Introduce complexity only *where necessary* for speed

- 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)?

- 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
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,
  - Memory leaks can destroy locality

- **Excessively large data structures – bad locality, memory fragmentation**

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

Hanson:

Linux Kernel:

Lots more *pointer chasing* and worse locality with Hanson
Lists Compared

Hanson:

Linux Kernel:

Of course, the Hanson lists are *polymorphic*!!
Lists Compared

Hanson:

A

B

C

Key Point:

Performance decisions almost always involve tradeoffs!

Of course, the Hanson lists are polymorphic!!
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) */
```
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 */
```

- These are called flexible array members.
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:**
  
  ```c
  for (i = 0; i < Array_length(arr); i++) ← constant?
  ```

  **Better:**
  
  ```c
  len = Array_length(arr);
  for (i = 0; i < len; i++)
  ```

- **Duplicate code:**
  
  **Bad:**
  
  ```c
  a = (f(x) + g(x)) * (f(x) / g(x))
  ```

  **Better:**
  
  ```c
  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: 12
```

```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: 12
(c and d are updated)
```

Do these functions really always compute the same result?

See sample alias_data.c
Aliasing: two names for the same thing

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

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

Result is: \textbf{24} (and \textit{c} is left with value \textbf{12}!)
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);
```

Result is: **24** (and c is left with value 12!)

Just in case arguments are the same...

...compiler can *never* eliminate the duplicate computation!
Aliasing: two names for the same thing

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

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

Result is: **24** (and c is left with value 12!)

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 modify c (or most anything else!)
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))
```

Expands directly to:

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

What's the problem with:

```c
int z = max(x++, y);
```
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))
```

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

Compiler expands directly to: `int x = ((c) > (d) ? (c) : (d));`

Now there’s *no* problem with: `int x = max(c++, d);`
Inlining: best of both worlds

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

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

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

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

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

Compiler expands directly to: `int x = ((c) > (d) ? (c) : (d));`
Code optimization techniques

- Moving code out of loops
- Common subexpression elimination
- When in doubt, let the compiler do it’s job
  - Higher optimization levels (e.g. gcc –O2)
- Inlining and macro expansion
  - Eliminate overhead for small functions
- Specialization
  - Compile time evaluation
  - Constant folding
  - Constant propagation
Specialization

```c
static int sum_squares(int x, int y)
{
    return ((x * x) + (y * y));
}
...
int num1 = 7;
int num2 = 3;

sum = sum_squares(num1, num2);
```

(7 * 7) + (3 * 3) = 58

gcc –O2 generates one instruction!:  movl $58, %r9d
Specialization with header files – must use ‘static’

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

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

src1.c

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

src2.c

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

gcc –p program src1.o src2.o
Specialization and Inlining work together

Once a function is inlined, it will permit other optimizations (specialization)
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:

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

- Use clean designs with good algorithms
- Use compiler optimization
- Don’t optimize until you know where the problems are
- Measure!
- Avoid unnecessary:
  - Use of memory
  - Indirections (pointer chasing)
- Watch for aliasing problems that kill optimizations
- Use macros and/or function inlining to avoid function call overhead
- Good news: optimizations compound
  - Inlining and specialization work together
  - Once you get out the big problems, things that used to be insignificant are now worth fixing!