Computer Systems and Networks

ECPE 170 – Dr. Pallipuram– University of the Pacific

Performance Optimization

Slides are courtesy of Dr. Jeff Shafer

Lab Schedule

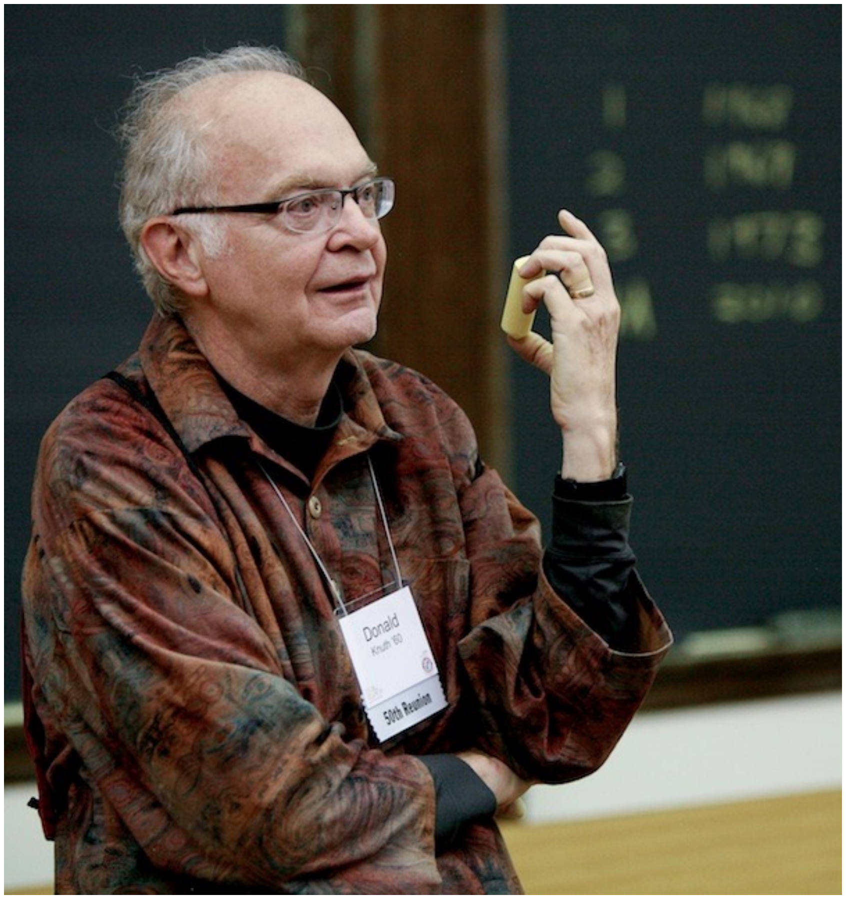
Activities

- **Today**
  - Discussion on Performance Optimization (Lab 6)
- **Next Week**
  - **Lab 6 – Performance Optimization**

Assignments Due

- **Lab 5**
  - **Due by OCT 6th 1159 PM**
- **Lab 6**
  - **Due by OCT 17th 1159 PM**
- **** Video Presentation #1 ****
  - **Released OCT 7th**
  - **Due OCT 16th**

Quotes – Donald Knuth



"People who are more than casually interested in computers should have at least **some idea of what the underlying hardware is like**. Otherwise the programs they write will be pretty weird."
– Donald Knuth

Remember this when we're learning assembly programming later this semester!

Performance Optimization

Vote

- **Who will do a better job improving program performance?**

- *The compiler* -vs- *The programmer*

Lab 6 Goals

1. What can the *compiler* do for programmers to improve performance?
2. What can *programmers* do to improve performance?

The Compiler

Compiler Goals

- **What are the compiler's goals with optimization off?**
- Obvious
  - Generate binary (executable) that produces correct output when run
  - Compile fast
- Less Obvious:
  - Make debugging produce expected results!

Compiler Goals

- **What are the compiler's goals with optimization on?**
- Reduce program **code size**
- Reduce program **execution time**
- **These may be mutually exclusive!**

Compiler Optimization Levels

O1: Moderately optimize the code, but do not increase the compilation time

```
gcc -O1 -o myexec main.c
```

O2: Optimize more, take time, but do not increase the code size

```
gcc -O2 -o myexec main.c
```

O3: Optimize aggressively, take time, even if code size increases!

```
gcc -O3 -o myexec main.c
```

Optimization Tradeoffs

- **What might we lose when we turn on optimization?**
- Compilation will take a lot longer
- Debugging is harder

Compiler Optimizations

- **Inline Functions**

```
int max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

```
max1 = max(w,x);
max2 = max(y,z);
printf("%i %i\n",
    max1, max2);
```

```
if(w>x) max1 = w;
else max1 = x;

if(y>z)max2 = y;
else max2 = z;

printf("%i %i\n",
    max1, max2);
```

- **Pros?** Lower overhead
- **Cons?** Bigger binary
  (except for tiny functions – like this?)

P1

Compiler Optimizations

- **What specific overhead exists here?**

```
int max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

- Calling a function
  - Save variables in the processor ("registers") to memory (in the stack)
  - Jump to the function
  - Create new stack space for function and its local variables
- Returning from function
  - Load old values from stack
  - Jump to prior location

Compiler Optimizations

- **Unroll Loops**

- **Pros?** Lower overhead
  Parallelism (potentially)

- **Cons?** Bigger binary

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

→

```
int x;
for (x = 0; x < 100; x+=5)
{
    delete(x);
    delete(x+1);
    delete(x+2);
    delete(x+3);
    delete(x+4);
}
```

Compiler Optimizations

- **What specific loop overhead exists here?**

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

- Top of loop
  - Compare x against 100
  - If less than, jump to …
  - Otherwise, jump to…
- Bottom of loop
  - Increment x by 1
  - Jump to top of loop
- *Impact on Branch Predictor (CPU microarchitecture)*

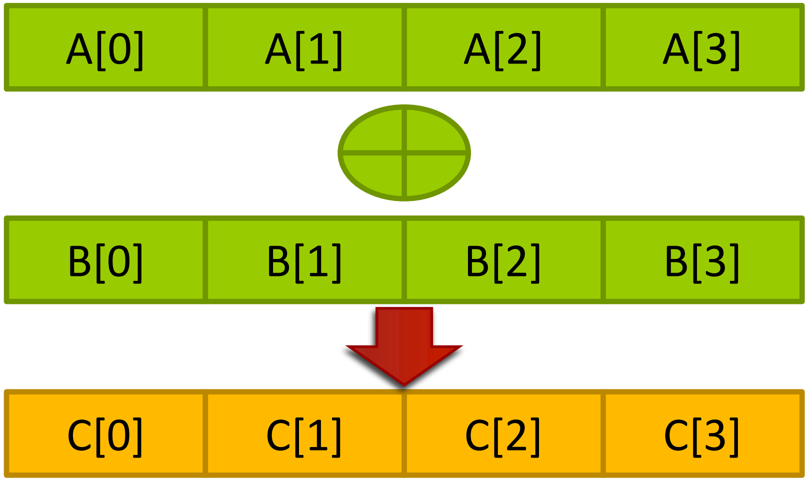
Compiler Optimizations

- **Loops Vectorization**

```
for(i=0; i<16; i++)
{
   C[i]=A[i]+B[i];
}
```

- **Pros?** Parallelism
- **Cons?** Requires specific features in CPU

Vector units:



Compiler Optimizations

- A large number of common compiler optimizations won't make sense until we learn assembly code later this semester
  - The compiler is optimizing the *assembly code*, not the high-level source code

The Programmer

The Compiler –vs– The Programmer

- Humans can do a better job at optimizing code than the compiler
  - Tradeoff: many developer-hours of time
- Big picture idea: The compiler must be **safe** and only make optimizations that function for **all possible data sets**.
  - Even if the programmer knows that a particular corner case cannot happen, the compiler doesn't know that

The Compiler –vs– The Programmer

- **Is this optimization safe for a compiler to do?**

```
void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}
```

```
void twiddle2(int *xp, int *yp)
{
    *xp += 2 * *yp;
}
```

- Twiddle1() needs 6 memory accesses
  - 2x read xp
  - 2x read yp
  - 2x write xp
- Twiddle2() needs 3 memory accesses
  - Read xp
  - Read yp
  - Write xp

The Compiler –vs– The Programmer

- **What if *xp and *yp pointed to the same memory address?**
- Twiddle1()
  - *xp += *xp;
  - *xp += *xp; // ***xp increased 4x**
- Twiddle2()
  - *xp += 2 * *xp; // ***xp increased 3x**
- This is **memory aliasing** (two pointers to the same address), and is hard for compilers to detect
  - But the programmer can know whether aliasing is a concern!

The Compiler –vs– The Programmer

- **Is this optimization safe for a compiler to do?**

```
int f();

int func1() {
    return f() + f() + f() + f();
}
```

⬇

```
int func2() {
    return 4*f();
}
```

The Compiler –vs– The Programmer

- **Depends on what f() does!**

```
int counter = 0;

int f()
{
    return counter++;
}
```

- With func1(): 0+1+2+3 = **<u>6</u>**
- With func2(): 4*0 = **<u>0</u>**
- Hard for compiler to detect **side effects**

The Compiler –vs– The Programmer

- Compare two functions that convert a string to lowercase

```
void lower1(char *s)
{
  int i;

  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
}
```

- **What does strlen() do again?**

```
void lower2(char *s)
{
  int i;
  int len = strlen(s);

  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
}
```

- **Could the compiler make this optimization for us?**

The Compiler –vs– The Programmer

- **Could the compiler make this optimization for us?**
- Very hard!
  - strlen() checks the elements of each string…
  - … and the string is being changed as each letter is set to lowercase
  - Would need to determine that the null character is not being set earlier or later in string!

The Compiler –vs– The Programmer

- **An awesome compiler won’t make up for a poor programmer**
  - *No compiler will ever replace a lousy bubble sort algorithm with a good merge sort algorithm*

Problem 2: Programmer Optimization: Code Motion

Rewrite the code below to optimize loop execution speed. Specifically, move a code section from inside the loop to outside because that section does not need to be called repeatedly!

```
for (int x=0; x<strlen(userinput); x++)
{
    if(tolower(game.grid[i][j+x])==tolower(userinput[x]))
    {
        flag=1;
    }
    else
    {
        flag=0;
        break;
    }
}
```

P2

Problem 3: Program Optimization: Reduce Procedure Calls

Can you find out why this code is inefficient and fix it? Reduce function calls as much as you can.

```
struct list {
  struct list *next;
  int num;
};
```

```
for(i=0;i<listsize;i++)
{
  ele = get_num(head,i);
  printf("%d",ele);
}
```

```
int get_num(struct list *head, int position) {
  struct list *temp=head;
  for(int i=0;i<position;i++) {
    temp=temp->next;
  }
  return temp->num;
}
```

P3

Problem 4: Program Optimization: Reduce Unwanted memory accesses.

Where is the inefficiency? Fix it!

Assume level2v and level1v are float arrays

```
for(i=0;i<1e6;i++) {
  level2v[i]+ =
      0.5*(1+atan2(divide((level1v[i]+1.2),18)));
  level2v[i]+= 0.5*(1+atan2(divide((level1v[i]-2),30)));
  level2v[i]+= divide(1,cos(divide((level1v[i]-2),60)));
}
```

P4

Problem 5: Program Optimization: Loop Unrolling

Rewrite your code from Problem 4 using loop unrolling. (Unroll by a factor of 2)

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

```
int x;
for (x = 0; x < 100; x+=5)
{
    delete(x);
    delete(x+1);
    delete(x+2);
    delete(x+3);
    delete(x+4);
}
```

P5

Problem 6-7: Research

Google search: Why is excessive use of global variables discouraged?

Google search: Research a `switch` statement vs an `if-else` ladder. Which one is better for performance?

P6-7

Programmer Optimizations

- Third part of lab will step you through six code optimizations

1. Code motion
2. Reducing procedure calls
3. Eliminating memory accesses
4. Unrolling loops x2
5. Unrolling loops x3
6. Adding parallelism

Programmer Optimizations

- **Should we use these optimizations everywhere?**
- **Beware of premature optimization!** Only spend effort optimizing if the performance monitoring tools point out that a particular algorithm/function is a bottleneck
- *"Premature optimization is the root of all evil (or at least most of it) in programming."*
  - Donald Knuth
- **Amdahl's law**

Amdahl's Law

- The overall performance of a system is a result of the interaction of all of its components
- System performance is most effectively improved when the performance of the **most heavily used components** is **improved** - ***Amdahl's Law***

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

S: overall speedup
f: fraction of work performed by a faster component
k: speedup of the faster component

Amdahl's Law

- **Which produces the greatest speedup?**
  - Accelerate by 8x a component used 20% of the time

$$S = \frac{1}{(1-f)+\frac{f}{k}} = \frac{1}{(1-.2)+\frac{.2}{8}} = 1.212$$

  - Accelerate by 2x a component used 80% of the time

$$S = \frac{1}{(1-f)+\frac{f}{k}} = \frac{1}{(1-.8)+\frac{.8}{2}} = 1.667$$

Amdahl's Law & Parallelism