Performance Improvement Revisited

Goals of this Lecture
• Help you learn how to:
  • Improve program performance by exploiting knowledge of underlying system
  • Compiler capabilities
  • Hardware architecture
  • Program execution
• And thereby:
  • Help you to write efficient programs
  • Review material from the second half of the course

Improving Program Performance
• Most programs are already “fast enough”
  • No need to optimize performance at all
  • Save your time, and keep the program simple/readable
• Most parts of a program are already “fast enough”
  • Usually only a small part makes the program run slowly
  • Optimize only this portion of the program, as needed
• Steps to improve execution (time) efficiency
  • Do timing studies (e.g., gprof)
  • Identify hot spots
  • Optimize that part of the program
  • Repeat as needed
Ways to Optimize Performance

• Better data structures and algorithms
  • Improves the “asymptotic complexity”
  • Better scaling of computation/storage as input grows
  • E.g., going from $O(n^2)$ sorting algorithm to $O(n \log n)$
  • Clearly important if large inputs are expected
  • Requires understanding data structures and algorithms
• Better source code the compiler can optimize
  • Improves the “constant factors”
  • Faster computation during each iteration of a loop
  • E.g., going from $1000n$ to $10n$ running time
  • Clearly important if a portion of code is running slowly
  • Requires understanding hardware, compiler, execution

Helping the Compiler Do Its Job

• Provide efficient mapping of program to machine
  • Register allocation
  • Code selection and ordering
  • Eliminating minor inefficiencies
• Don’t (usually) improve asymptotic efficiency
  • Up to the programmer to select best overall algorithm
• Have difficulty overcoming “optimization blockers”
  • Potential function side-effects
  • Potential memory aliasing

Optimizing Compilers
Limitations of Optimizing Compilers

• Fundamental constraint
  • Compiler must not change program behavior
  • Ever, even under rare pathological inputs

• Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  • Data ranges more limited than variable types suggest
  • Array elements remain unchanged by function calls

• Most analysis is performed only within functions
  • Whole-program analysis is too expensive in most cases

• Most analysis is based only on static information
  • Compiler has difficulty anticipating run-time inputs

Avoiding Repeated Computation

• A good compiler recognizes simple optimizations
  • Avoiding redundant computations in simple loops
  • Still, programmer may still want to make it explicit

• Example
  • Repetition of computation: n * i

```c
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```c
for (i = 0; i < n; i++) {
  int ni = n * i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Worrying About Side Effects

• Compiler cannot always avoid repeated computation
  • May not know if the code has a "side effect"
  • ... that makes the transformation change the code’s behavior

• Is this transformation okay?

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

```c
int func1(int x) {
  return 4 * f(x);
}
```

• Not necessarily, if

```c
int counter = 0;
int f(int x) {
  return counter++;
}
```

And this function may be defined in another file known only at link time!
Another Example on Side Effects

- Is this optimization okay?
  ```c
  for (i = 0; i < strlen(s); i++) {
      /* Do something with s[i] */
  }
  length = strlen(s);
  for (i = 0; i < length; i++) {
      /* Do something with s[i] */
  }
  ```

- Short answer: it depends
  - Compiler often cannot tell
  - Most compilers do not try to identify side effects

- Programmer knows best
  - And can decide whether the optimization is safe

Memory Aliasing

- Is this optimization okay?
  ```c
  void twiddle(int *xp, int *yp) {
      *xp += *yp;
      *xp += *yp;
  }
  ```

- Not necessarily, what if xp and yp are equal?
  - First version: result is 4 times *xp
  - Second version: result is 3 times *xp

Memory Aliasing

- Memory aliasing
  - Single data location accessed through multiple names
  - E.g., two pointers that point to the same memory location

- Modifying the data using one name
  - Implicitly modifies the values seen through other names

- Blocks optimization by the compiler
  - The compiler cannot tell when aliasing may occur
  - ... and so must forgo optimizing the code

- Programmer often does know
  - And can optimize the code accordingly
Another Aliasing Example

• Is this optimization okay?

```c
int *x, *y;
x = 5;
y = 10;
printf("x=%d \n", *x);
```

• Not necessarily
  • If y and x point to the same location in memory...
  • … the correct output is “x = 10v”

Summary: Helping the Compiler

• Compiler can perform many optimizations
  • Register allocation
  • Code selection and ordering
  • Eliminating minor inefficiencies
• But often the compiler needs your help
  • Knowing if code is free of side effects
  • Knowing if memory aliasing will not happen
• Modifying the code can lead to better performance
  • Profile the code to identify the “hot spots”
  • Look at the assembly language the compiler produces
  • Rewrite the code to get the compiler to do the right thing

Exploiting the Hardware
Underlying Hardware

- Implements a collection of instructions
  - Instruction set varies from one architecture to another
  - Some instructions may be faster than others
- Registers and caches are faster than main memory
  - Number of registers and sizes of caches vary
  - Exploiting both spatial and temporal locality
- Exploits opportunities for parallelism
  - Pipelining: decoding one instruction while running another
  - Benefits from code that runs in a sequence
  - Superscalar: perform multiple operations per clock cycle
  - Benefits from operations that can run independently
  - Speculative execution: performing instructions before knowing they will be reached (e.g., without knowing outcome of a branch)

Addition Faster Than Multiplication

- Adding instead of multiplying
  - Addition is faster than multiplication
- Recognize sequences of products
  - Replace multiplication with repeated addition

```java
for (i = 0; i < n; i++) {
    int ni = n * i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```

Bit Operations Faster Than Arithmetic

- Shift operations to multiple/divide by powers of 2
  - “x >> 3” is faster than “x/8”
  - “x << 3” is faster than “x * 8”
- Bit masking is faster than mod operation
  - “x & 15” is faster than “x % 16”
Caching: Matrix Multiplication

- **Caches**
  - Slower than registers, but faster than main memory
  - Both instruction caches and data caches

- **Locality**
  - Temporal locality: recently-referenced items are likely to be referenced in near future
  - Spatial locality: items with nearby addresses tend to be referenced close together in time

- **Matrix multiplication**
  - Multiply n-by-n matrices A and B, and store in matrix C
  - Performance heavily depends on effective use of caches.

Matrix Multiply: Cache Effects

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        for (k=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
}
```

- **Reasonable cache effects**
  - Good spatial locality for A
  - Poor spatial locality for B
  - Good temporal locality for C

- **Rather poor cache effects**
  - Bad spatial locality for A
  - Good temporal locality for B
  - Bad spatial locality for C
Matrix Multiply: Cache Effects

```c
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++)
            c[i][j] += a[i][k] * b[k][j];
    }
}
```

- Good cache effects
  - Good temporal locality for A
  - Good spatial locality for B
  - Good spatial locality for C

Parallelism: Loop Unrolling

- What limits the performance?

```c
for (i = 0; i < length; i++)
    sum += data[i];
```

- Limited apparent parallelism
  - One main operation per iteration (plus book-keeping)
  - Not enough work to keep multiple functional units busy
  - Disruption of instruction pipeline from frequent branches

- Solution: unroll the loop
  - Perform multiple operations on each iteration

Parallelism: After Loop Unrolling

- Original code

```c
for (i = 0; i < length; i++)
    sum += data[i];
```

- After loop unrolling (by three)

```c
// Combine three elements at a time */
limit = length / 3;
for (i = 0; i < limit; i+=3)
    sum += data[i] + data[i+1] + data[i+2];

// Finish any remaining elements */
for (; i < length; i++)
    sum += data[i];
```
Program Execution

Avoiding Function Calls

- Function calls are expensive
  - Caller saves registers and pushes arguments on stack
  - Callee saves registers and pushes local variables on stack
  - Call and return disrupt the sequence flow of the code

- Function inlining:
  ```c
  void g(void) {
      /* Some code */
  }
  void f(void) {
      ... g();
  }
  
  Some compilers support "inline" keyword directive.
  ```

Writing Your Own Malloc and Free

- Dynamic memory management
  - `malloc()` to allocate blocks of memory
  - `free()` to free blocks of memory

- Existing `malloc()` and `free()` implementations
  - Designed to handle a wide range of request sizes
  - Good most of the time, but rarely the best for all workloads

- Designing your own dynamic memory management
  - Forego using traditional `malloc()` and `free()`, and write your own
    - E.g., if you know all blocks will be the same size
    - E.g., if you know blocks will usually be freed in the order allocated
    - E.g., «insert your known special property here>
Stand on the Shoulders of Giants

- Find good working code
  - Our cache example can be improved using “blocking”
  - The library `qsort()` function has many optimizations
  - Various malloc/free libraries exist – even Google has one
- Learn from the lessons of others
  - Read about prior systems – papers, online, textbooks
  - Understand what precipitated change
  - Learn how they reacted to change
- Look before you leap
  - Some hardware already unrolls small loops
  - Small code takes less cache space
  - Pointers can slow down processors (but not always)

Conclusion

- Work smarter, not harder
  - No need to optimize a program that is “fast enough”
  - Optimize only when, and where, necessary
- Speeding up a program
  - Better data structures and algorithms: better asymptotic behavior
  - Optimized code: smaller constants
- Techniques for speeding up a program
  - Coax the compiler
  - Exploit capabilities of the hardware
  - Capitalize on knowledge of program execution

Course Wrap Up
The Rest of the Semester

- **Final Assignment Due:** Tuesday, May 11
- **Deans Date:** Tuesday, May 11
- Cannot submit final assignment after 5:00PM
- **Final Exam:** Thursday, May 20
  - 1:30 PM in room Friend 101
  - Exams from previous semesters are online at
  - Covers entire course, with emphasis on second half of the term
  - Open book, open notes, open slides, etc. (just no computers!)
  - No need to print/bring the IA-32 manuals
- **Office hours during reading/exam period**
  - Times TBA on course mailing list

Goals of COS 217

- **Understand boundary between code and computer**
  - Machine architecture
  - Operating systems
  - Compilers
- **Learn C and the Unix development tools**
  - C is widely used for programming low-level systems
  - Unix has a rich development environment
  - Unix is open and well-specified, good for study & research
- **Improve your programming skills**
  - More experience in programming
  - Challenging and interesting programming assignments
  - Emphasis on modularity and debugging

Relationship to Other Courses

- **Machine architecture**
  - Logic design (306) and computer architecture (471)
  - COS 217: assembly language and basic architecture
- **Operating systems**
  - Operating systems (318)
  - COS 217: virtual memory, system calls, and signals
- **Compilers**
  - Compiling techniques (320)
  - COS 217: compilation process, symbol tables, assembly and machine language
- **Software systems**
  - Numerous courses, independent work, etc.
  - COS 217: programming skills, Unix tools, and ADTs
Lessons About Computer Science

- **Modularity**
  - Well-defined interfaces between components
  - Allows changing the implementation of one component without changing another
  - The key to managing complexity in large systems

- **Resource sharing**
  - Time sharing of the CPU by multiple processes
  - Sharing of the physical memory by multiple processes

- **Indirection**
  - Representing address space with virtual memory
  - Manipulating data via pointers (or addresses)

Lessons Continued

- **Hierarchy**
  - Memory: registers, cache, main memory, disk, tape, ...
  - Balancing the trade-off between fast/small and slow/big

- **Bits can mean anything**
  - Code, addresses, characters, pixels, money, grades, ...
  - Arithmetic can be done through logic operations
  - The meaning of the bits depends entirely on how they are accessed, used, and manipulated

Have a Great Summer!!!