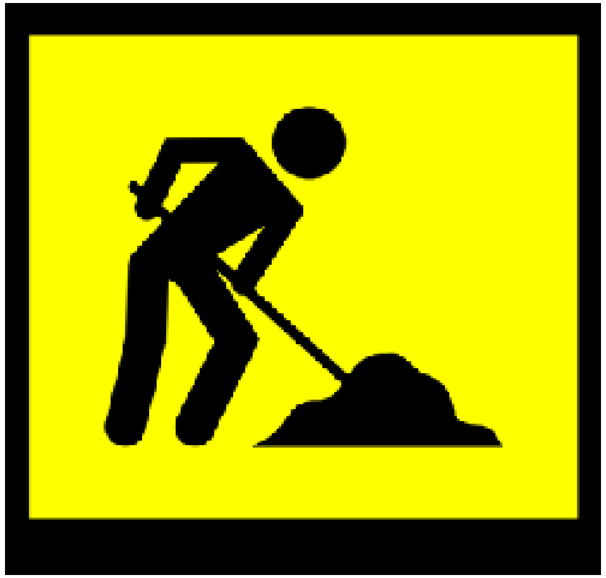
Computer Architecture and Operating Systems
Lecture 15: Optimizations

Andrei Tatarnikov

atatarnikov@hse.ru
@andrewt0301

Work

- **Definition:** The work of a program (on a given input) is the sum total of all the operations executed by the program.



Optimizing Work

- Algorithm design can produce dramatic reductions in the amount of work it takes to solve a problem, as when a Θ(n lg n)-time sort replaces a $\Theta(n^2)$-time sort.
- However, reducing the work of a program does not automatically reduce its running time due to complex nature of computer hardware:
  - instruction-level parallelism (ILP)
  - caching
  - vectorization
  - speculation and branch prediction
  - etc.
- Nevertheless, reducing the work serves as a good heuristic for reducing overall running time

Performance Assessment

- Analytical assessment (asymptotic notation) is not enough. Implementation of an algorithmically-efficient algorithm can be slow because of inefficient use of hardware. Constant factors matter!

- Create tests with benchmarks and use profiling tools to find bottlenecks and compare algorithms.

Recommendations

- Data structures
  - Packing and encoding
  - Augmentation
  - Precomputation
  - Compile-time initialization
  - Caching
  - Lazy evaluation
  - Sparsity
- Loops
  - Hoisting
  - Sentinels
  - Loop unrolling
  - Loop fusion
  - Eliminating wasted iterations
- Logic
  - Constant folding and propagation
  - Common-subexpression elimination
  - Algebraic identities
  - Short-circuiting
  - Ordering tests
  - Creating a fast path
  - Combining tests
- Functions
  - Inlining
  - Tail-recursion elimination
  - Coarsening recursion

Hoisting

- The goal of hoisting — also called loop-invariant code motion — is to avoid recomputing loop-invariant code each time through the body of a loop.

```
for (int i = 0; i < 100; i++) {
    a[i] = x + y;
}
```

```
int t = x + y;
for (int i = 0; i < 100; i++) {
    a[i] = t;
}
```

Loop Unrolling

- Loop unrolling attempts to save work by combining several consecutive iterations of a loop into a single iteration, thereby reducing the total number of iterations of the loop and, consequently, the number of times that the instructions that control the loop must be executed.
  - Full loop unrolling: All iterations are unrolled.
  - Partial loop unrolling: Several, but not all, of the iterations are unrolled.

Full Loop Unrolling

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += A[i];
}
```

```
int sum = 0;
sum += A[0];
sum += A[1];
...
sum += A[9];
```

Partial Loop Unrolling

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += A[i];
}
```

```
int sum = 0;
int j;
for (j = 0; j < n-3; j += 4) {
    sum += A[j];
    sum += A[j + 1];
    sum += A[j + 2];
    sum += A[j + 3];
}
for (int i = 0; i < 10; i++) {
    sum += A[i];
}
```

- Benefits of loop unrolling
    - Lower number of instructions in loop control code
    - Enables more compiler optimizations
- Unrolling too much can cause poor use of instruction cache

Loop Fusion

- The idea of loop fusion — also called jamming — is to combine multiple loops over the same index range into a single loop body, thereby saving the overhead f loop control.

```
for (int i = 0; i + n; ++i) {
    C[i] = (A[i] += B[i]) ? A[i] : B[i];
}
for (int i = 0; i + n; ++i) {
    D[i] = (A[i] += B[i]) ? B[i] : A[i];
}
```

```
for (int i = 0; i + n; ++i) {
    C[i] = (A[i] += B[i]) ? A[i] : B[i];
    D[i] = (A[i] += B[i]) ? B[i] : A[i];
}
```

Eliminating Wasted Iterations

- The idea of eliminating wasted iterations is to modify loop bounds to avoid executing loop iterations over essentially empty loop bodies.

Optimizing Compilers

- Provide efficient mapping of program to machine
    - register allocation
    - code selection and ordering (scheduling)
    - dead code elimination
    - eliminating minor inefficiencies
- Do not (usually) improve asymptotic efficiency
    - up to programmer to select best overall algorithm
    - big-O savings are (often) more important than constant factors
        - but constant factors also matter
- Have difficulty overcoming “optimization blockers”
    - potential memory aliasing
    - potential procedure side-effects

Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do inter-procedural analysis within individual files
    - But, not between code in different files
- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Optimization Blocker: Procedure Calls

- Warning: compiler treats procedure call as a black box
  - Procedure may have side effects
  - Alters global state each time called
  - Function may not return same value for given arguments
  - Depends on other parts of global state
- Remedies:
  - Use of inline functions
  - Do your own code motion

Conclusion

- Avoid premature optimization. First get correct working code. Then optimize, preserving correctness by regression testing.
- Reducing the work of a program does not necessarily decrease its running time, but it is a good heuristic.
- The compiler automates many low-level optimizations.
- To tell if the compiler is actually performing a particular optimization, look at the assembly code.

Any Questions?

```
            .text
__start:    addi t1, zero, 0x18
            addi t2, zero, 0x21
cycle:      beq t1, t2, done
            slt t0, t1, t2
            bne t0, zero, if_less
            nop
            sub t1, t1, t2
            j cycle
            nop
if_less:    sub t2, t2, t1
            j cycle
done:       add t3, t1, zero
```