CS245 – Lecture 5

Optimizing Compilers

Tony Givargis

What is a Compiler?

- Transforms one computer language to another
  - High-level to low-level
  - Low-level to high-level (de-compiler)
  - High-level to high-level (source to source translator)
- Internals
  - Preprocessing
  - Lexical analysis (Lexer)
  - Semantic analysis (Parser)
  - Code Generation
  - Code Optimization

C/C++ Pre-Processing

- Expands and substitutes #xxxxxx directives
- Lacks syntax and semantics knowledge of underlying C
  - Has been criticized
- Macros (#define)
  - Replaced by inline functions (performance)
  - Replaced by templates (extensible)
- Conditional compilation (#if-#then-#else)
  - Replaced by dead code elimination

Lexical Analyzer (Lexer)

- Converting a sequence of characters into a sequence of tokens (a.k.a., scanner)
- Lexical rules of language are captured by regular expressions
  - Finite state machine lexer recognizes tokens
  - Longest match rule
  - White-space delimiters
- Lexer generators
  - Lex, Flex (Rewrite of Lex), JLex (Java), etc.

Semantic Analyzer (Parser)

- Match a sequence of tokens to against a formal grammar of the language
  - On a match, call the back-end
  - Else, issue error
- The language is captured by a context-free grammar
  - Hand parsers or generated parsers recognize productions
  - Typically, parsers recognize a superset of a language
    - Rely on back end to catch errors
- Top-down / bottom-up parsers
  - Recursive descent parser / LALR parser

Code Generation

- Lexer/Parser yield an internal representation
  - Abstract syntax tree
  - Parse tree
- Embedded compilers often make multiple passes over the internal representation
  - Convert the parse tree into a linear sequence of instructions
    - Abstract 3-address code
  - Final passes generate executable code from the abstract 3-address code sequence

Code Optimization

- Code optimization modifies code (as parsed) to make it more efficient (Performance, footprint, power)
  - Must retain program behavior
  - Optimization may takes place at all phases of compilation
- Kinds of optimizations
  - Algorithmic optimization
  - General (dataflow and loop) optimization
  - Machine dependent/independent optimization
- No universal optimization objective
  - Optimization overhead

Optimizations

- Alias and pointer analysis
  - A pointer may point to any memory location (big trouble)
  - Restricting the scope of references helps with subsequent optimizations
- Dataflow optimization
  - Common sub-expression elimination
    - 3 * (X+Y) - 2 * (X + Y) => 3 * Z – 2 * Z
  - Constant folding
    - Y = 4; X * (Y * Y) => X * 16

Optimizations …

- Loop optimization
  - Induction variable elimination
  - Loop fission
  - Loop splitting
  - Loop fusion
  - Loop inversion
    - While(x) {} => if(x) { do {} while(x) }
    - Reduces the number of jumps, helps with pipeline stalls
  - Loop-invariant code motion
  - Loop-nest optimization
  - Loop unrolling

Optimizations …

- Loop optimizations …
  - Loops unswitching
    - Hoist conditional expression out of the loop body
  - Software pipelining
    - Split the loop body into several sections, start new loop iteration when first section is complete and so on
  - Loop parallelization
    - Execute the loop using several threads
- Machine-dependent optimization
  - Register allocation
  - Instruction selection & scheduling

Optimizations …

- General optimizations
  - Dead-code elimination
  - Code factoring (opposite of code inlining)
  - Function versioning
- Advanced optimizations
  - Task partitioning
  - Static evaluation assisted optimizations
  - Trace assisted optimization
- Most code optimizations focus on dataflow
  - Control-flow optimizations are more challenging???

Embedded System Compilers

- Optimization objectives are more stringent
  - A blend of power, performance, and area
- Can afford to spend more time on compilation to achieve the objectives
- Additional issues
  - Often deal with cross compilers
  - Generated code forms part of the firmware
  - Code is generated to execute directly on micro-controller or an RTOS

Forms of Compilation

- Common understanding of a compiler
  - Generates code from high-level language that runs on this machine
- Cross compiler
  - Slight variation on above
- Interpreter
  - As code is generated, it is executed and discarded
- JIT
  - Interpreter that does not discard the code

Embedded Software

- Programming language issues
  - Multi-core and reconfigurable architectures
  - Distributed real-time control and other complex RT systems
  - Reliability, security, and privacy
  - Virtual machines
- Compilers
  - Architecture, RTOS
  - Binary translation & optimization
  - Support for debugging, profiling, and exceptions

Embedded Software …

- Compilers …
  - Optimization for low power, low energy, low code and data size, and high (real-time) performance
- Tools for analysis, specification, design, and implementation of embedded systems
  - Validation and verification
  - System integration and testing
  - Timing analysis, timing predictability, WCET analysis and real-time scheduling analysis
  - Performance monitoring and tuning

JIT Example

```
#include <jit.h>

unsigned char buf[1024];
typedef (*p2f)(int, int);

int main() {
  if( jit_compile(“int min(int x, int y) {”
                  “  return x < y ? x : y;”
                  “}”, buf) ) {
    printf(“%i\n”, (p2f)buf(3, 5));
  }
  else {
    assert( 0 );
  }
  return 0;
}
```

Conditional Expressions

- Determine the execute path
  - The more we know about them, the better we can optimize the control flow
  - Static evaluation can yield information about C.E.
- Control flow optimizations
  - Loop body optimization
  - Loop nest optimization
  - Loop splitting
  - Etc.

C.F. Optimization Example

```
for all x = min to max do
    for all y = min to max do
        if (x^2 + y^2 – x^2 × y == 0) then
            color(x,y,BLACK);
        end if
    end for
end for
```

```
for all x = min to max do
    for all y = min to max do
        if y < 0 then
            continue;
        else
            if (x^2 + y^2 – x^2 × y == 0) then
                color(x,y,BLACK);
            end if
        end if
    end for
end for
```

C. E. Evaluation

- Start with the syntax tree for a C.E. and normalize
  - $2X_0 + X_1 > -4$
  - $2X_0 + X_1 - 4 = 0$
- Solve for roots
  - Assume all, but one variable, as constants
  - Estimate an initial interval for each variable
  - Use interval for all arithmetic in root finding algorithm
  - Repeat
- Partition the domain space
  - Optimize

C. E. Evaluation