Introduction

CS 2210 Compiler Design

Wonsun Ahn
What is a Compiler?

• Compiler: A program that translates source code written in a programming language to a target code
• Source code: A collection of instructions that performs a desired computation (typically in human readable form)
  – E.g. C, C++, Fortran, Java, Python, Matlab
• Target code: The result of compilation. In the form of…
  – Object code (a.k.a. binary code): A collection of instructions that target machine understands (e.g. MIPS)
  – Source code (for a different language)
  – Byte code: A collection of instructions that an abstract virtual machine understands (e.g. JVM)
• Invariant: original program semantics should stay the same
What is an Interpreter?

- Interpreter: A program that reads in source code and executes it on the target machine, typically line by line
  - Software becomes more portable
    - Binary code is tied to a specific machine architecture
    - Source code can run anywhere with correct interpreter
  - Software becomes more secure and safe
    - Interpreter can perform runtime checks
    - E.g. Checks to prevent insecure read of protected data
  - Interpreter is relatively simpler to implement than compiler (Popular for languages with small user base)
    - Interpretation much slower compared to binary execution (Source line has to be interpreted each time it’s executed)
Just-In-Time (JIT) Compiler

• Just-In-Time (JIT) Compiler: A compiler that performs translation at runtime (just in time before execution)
  – Traditional compilers are called Ahead-Of-Time (AOT)
  – Typically implemented when an interpreted language gains enough of a user base to warrant its development

• Pros / Cons compared to AOT compiler
  + Software can be run in a portable, secure, safe way
  – Slight performance overhead due to JIT compilation time (But much less than interpretation since compiled code can be cached and reused)
  + Opportunities for better performance
    • By profiling program behavior and optimizing against it
    • By tailoring target code to details of underlying machine
AOT vs. JIT Compilation

**Ahead-Of-Time (AOT) Compilation**
- Compile
- Binary
- Distribute

**Languages:**
- C/C++
- Fortran

**Just-In-Time (JIT) Compilation**
- Translate/Minify
- Bytecode/Source
- Distribute

**Languages:**
- Java
- C#/.net
- JavaScript
- Python
- PHP
- Lua
- R
- OpenCL
Topics Covered
Compiler Phases

Source Program: IF (a<b) THEN c=1*d;
The front end of a compiler is the group of phases that **analyzes** the source code and builds one or more internal representations (IRs) out of that analysis

- Comprised of lexical, syntax, and semantic analyses
- IRs can be syntax trees, 3-address codes, etc.

**Lexical Analysis,**

- Input: Source code text
- Output: Sequence of tokens (identifiers, keywords, constants, operators …)
- Scans source code from left to right and uses regular expressions to chop text into tokens
- Also checks for illegal tokens (e.g. malformed identifier)
Front End

- **Syntax Analysis**
  - Input: Sequence of tokens
  - Output: Syntax tree
  - Uses language grammar rules to group tokens into a hierarchy called a syntax tree that represents program
  - Also checks for syntax errors (e.g. malformed if statement)

- **Semantic Analysis**
  - Input: Syntax tree
  - Output: Lower level IR (e.g. 3-address code)
  - Uses symbol table to aid in the understanding of what variables mean and where they are declared
  - Generates a pseudo-code IR that can easily be translated to machine code
The back end of a compiler is the group of phases that **synthesizes** machine code from the internal representations (IRs) generated by the front end

- Comprised of code optimization and code generation

**Code Optimization**

- Input: IR (e.g. 3-address code)
- Output: Optimized IR
- Apply modifications to IR such that code runs faster and consumes less memory
Back End

• Code Generation
  – Input: (Optimized) IR
  – Output: Target Code
  – Perform following tasks to transform IR to target code:
    • Instruction Selection – select actual machine ISA instructions to implement computation in IR
    • Register Allocation – allocate often used locations in processor registers
    • Stack Allocation – allocate rest of the locations in program stack memory
  • An additional code optimization phase may follow code generation to apply target machine specific optimizations
Multiple Front Ends and Back Ends

• Modern compilers typically have multiple front ends
  – E.g. GCC (GNU Compiler Collection) has front ends for C, C++, Fortran, and Java among others
  – Means all front ends generate the same IR format that can be passed to the back end for code generation
• Modern compilers typically have multiple back ends
  – E.g. GCC has back ends for x86, ARM, SPARC, etc.
  – Means same IR can be translated to multiple targets by the code generator
• A common IR is central in enabling this diversity
Why Learn Compilers?
Why Learn Compilers?

• Understanding compilers will allow you write better, more robust, more efficient programs
  – You will have a deep understanding of how your program will execute on the actual machine

• Techniques used in compilers such as lexical analysis can be used for other purposes
  – Compiler techniques can be used to process and analyze any kind of structured text data

• You may need to write a domain specific language (DSL)
  – Long term, it is often cost-effective to design a language tailored to a specific domain

• You may end up doing research on compilers
Why Compilers is Still a Research Issue

• You might think…
  – Compilers have been around since the 1950s
  – The C language has been around since the 1970s
  – We are still using the C language
  – 40+ years is enough time to spend on a single problem
  – There is probably not much left to do in terms of compilers

• But you may be wrong
  – Changes in programming environment is producing new challenges in the front end of the compiler
  – Changes in computer architecture design is producing new pressures in the back end of the compiler
Popularity of Programming Languages

* Number of programmers coding in given language over time
  (Reference: www.ohloh.net)

- C/C++ dominated in 2005
- Now, JavaScript and Python are the most used languages

Mobile Web
Big Data Analysis

- AOT
- JIT
Front End Challenges

• The need for fast product prototyping and use of computing in non-CS fields is leading to unprecedented use of scripting languages such as JavaScript, Python, PHP etc.
  – Very flexible by design. E.g. Dynamic Typing:
    ```
    var x = 1, y, z;
    if (...) y = 2;
    else y = “2”;
    z = x + y; // z == 3 or z == “12”
    ```
  – How to generate code efficiently for such languages?

• Increasingly, domain specific languages are being developed by domain specialists with limited CS background (e.g. R language for statistics, MATLAB for engineering)
  – How to enable efficient code generation for these languages without rewriting compilers from scratch?
Back End Challenges

• End of Dennard scaling (a.k.a the Power Wall) means new increases in performance needs to come from parallelism
  – How can the compiler automatically parallelize or vectorize code so that it can run on multiple cores or wide data parallel architectures such as GPGPUs?

• Energy efficiency is also dictating a move to heterogeneous and accelerator heavy architectures
  – How can the compiler generate code for these designs?
  – How can the compiler decide to run what where?

• Hardware cache coherence in the many-core era is becoming too power hungry to realize
  – How can the compiler better utilize the memory hierarchy and manage data sharing?
Other Issues

• How can the compiler generate code such that it can efficiently detect any insecure access to protected data or execution of malicious code?

• How can the compiler generate code to increase reliability (e.g. robustness in the face of software bugs or random bit flips due to hardware soft errors)?

• How can compilers and programming languages help in writing bug free programs?
  – Automatic data race detection and analysis
  – Automatic memory error detection and analysis

• All of the above can be helped by compiler means through
  – Static data flow / control flow analysis OR
  – Dynamic checks by compiler instrumented code