Building a Runnable Program and Code Improvement

Dario Marasco, Greg Klepic, Tess DiStefano
Building a Runnable Program
Review

- Front end code
  - Source code analysis
  - Syntax tree
- Back end code
  - Target code and algorithms
  - Will discuss further now
- Can be an arbitrary distinction
Back-End Compiler Structure

- Less uniformity in functions, structures, code improvement than front end
  - Programmer preference, constraints, users
- Example of a compiler structure on the following slides
- Phases and Passes
  - A pass is a sequential group of uninterrupted phases
  - Ex: front end can be one pass and back end is another
- Compiler(code) -> assembly language
- Assembler can be considered another pass
  - Assigns addresses to data and code
  - Translate operations into binary codes which represent them
- Next step: linking
Example (Fig 14.1)

- Scan & parse syntax tree
- Analyzer reviews semantics and attributes
- Intermediate makes a control flow graph with no branches in or out
  - Ops have virtual registers
- MICI eliminates redundancy locally (cmds) & globally (blocks)
- TCG combines blocks to program (still uses VRs)
- MSCI allocates registers and schedules instructions
GCD example
Intermediate Forms

- Steps between front and back end of the compiler
- High Level: based on trees or directed (acyclic) graphs
- Medium Level: three addressed instructions based on control flow graphs
  - Instructions are simplified, with unlimited registers
- Low Level: assembly language for particular machine to execute code
  - Want to make this part simple if running on many different kinds of machines
  - Otherwise most code improvement is here to make it like the target machine’s assembly
- Goal: Intermediate Forms should not confine to source or machine
  - Can build one front end and one back end per machine it will run on; n+m not m*n
- Information must be stored linearly, ex: replacing arrows with pointers
- Option: Stack Based for simplicity (code as well as input) and brevity
  - Reduce memory and bandwidth
Code Generation - simplified backend & registers

- Attribute Grammar: used to formalize intermediate code generation
- Evaluation:
  - Determine registers to use
  - Generate code
- Target code keeps an array of free registers to form a local stack
  - If run out of architectural registers: register allocator spills register data into memory and clears the register to be temporarily reused
  - Poor use of resources
    - Doesn’t minimize registers used or
    - Doesn’t keep variables used frequently to minimize loads and stores
Address Space Organization

- Linker gets relocatable object code in three kinds of tables
  - Import Table: Finds instructions without knowing where they point (yet)
  - Relocation Table: Finds instructions pointing elsewhere in the file
  - Export Table: Gathers names and addresses in current file to be used in other files

- Loader gets executable code file
  - No usage of external symbols
  - Defines starting address for execution

- Object files are divided into sections
  - First: stores tables listed above and description of how much space is needed
  - Others: code, data (read only and writable), symbol table information
Segments of a Running Program

- Uninitialized Data
  - Allocated at load time, filling space with zeros to allocate it
- Stack
  - Allocated small at load time, expands when programs try to access beyond segment end
- Heap
  - Allocated small at load time, expands when heap-management library subroutines make system calls
- Files
  - Library routines map files to memory, giving the address where the file starts, with page faults triggering when to etch next segments
- Dynamic Libraries
  - When libraries are shared, most code is shared as well, but some segments hold "linkage information" to map copies into memory of private libraries
Assembly

- Assembler processes assembly language generated by compilers (last step discussed above)
- Translate symbolic representation into executable code
  - Replace commands with machine language codes
  - Replace symbolic names with addresses
- Assembler Front End
  - Front end for when human input is needed
- Compiler sends Information to assembler for its front end
  - Dumps output in a formatted way. First symbolically, then as a text dump optionally
  - Or - use a disassembler instead of dumping assembly language (more flexibility and clarity)
The Assembler

- Not always a one-to-one connection between operations and instruction codes
  - Want to make assembly language easier for programmers
- MIPS uses pseudo-instructions where one command can point to different instructions based on the parameters
  - Polymorphism
- Phases
  - Scan and parse input to build internal representation
  - Identify symbols (internal, external) and assign location to internal ones
  - Produce object code
- Object file stores table of its exported symbols along with addresses
  - Stores absolute and relocatable words
  - Absolute are known at runtime (constants and registers)
  - Relocatable uses addresses, like a jump instruction going to an addressed location

```
add $r1 $r2 $r3
add $r1 0 $r3
```
Linking

- Puts together compilation units out of fragments of program assembled separately
  - Sometimes broken by files (by programmer), sometimes by subroutines (by environment)
  - Each unit has a relocatable object file
- Static Linker: makes an executable file to run the program
- Dynamic Linker: links program after program is sent to memory to execute
- Linking (link loading) involves
  - Relocating: use object files to get import, export, relocation tables
    - Static linker gathers units, chooses order, notes addresses of each, then processes, replaces external references with addresses, and updates instructions
  - Resolving external references
## Linking

- Libraries are made of separately compiled program fragments
  - Linker has to find which object files have the fragments a program needs
  - Recursively bring in multiple fragments
- When two modules are linked together, it’s important to ensure they use the same versions of each datum
- When compiling one, a dummy symbol is made to hold the header, which is referenced in the other module as well
  - Textual representation of most recent modification to a header (cache?): clocks don’t sync
  - Checksum of header files to verify with small possibility of confusion
  - Name mangling keeps an object file maintaining all imported and exported names, as well as their types. With all the types used for objects and functions, when they are very complicated hashing can be used to compress the data.
Dynamic Linking

- Sometimes a program can be executed multiple times at once
- So operating systems keep track of what is running and use tables to map all instances to shared read-only copies of code (as well as its own writable portion)
- Same for shared libraries
Code Improvement
Code Improvement

- We discussed techniques that work but are not ideal
  - Redundant computation
  - Inefficient use of registers
  - Multiple functional units
  - Too much work for a modern processor
- So compilation needs to generate good, fast code to optimize it
Kinds of Optimization

- **Peephole Optimization**
  - Cleans up generated code with small instruction window

- **Local Optimization**
  - Optimizes blocks of code at a time (sequences of instructions that will always execute entirely) by eliminating redundant code and optimizes to effective instruction scheduling and register allocation

- **Global Optimization**
  - Most aggressive, scope of whole subroutines
  - Redundancy elimination, instruction scheduling, register allocation, loop performance at a multi-basic-block level
  - Use control flow graph and data flow analysis between basic blocks

- **Interprocedural Improvement**
  - Not covered in detail, more aggressive than global
Phases of Code Improvement

- Sometimes need to be in order, sometimes need to be repeated
- Next slide shows set of phases discussed earlier, but in more detail in the machine independant section
  - Identify and eliminate redundant loads, stores, computation in each block
  - Then between blocks
  - Then improve loops which takes up most of runtime
- Machine Specific code improvement 4 phases
  - Work out instruction scheduling and register allocation
Phases of Code Improvement Example
Peephole Optimization of Target Code

- Look for issues in small windows of instructions based on heuristics of what is not optimal. Examples are below, not unique to peephole
  - Elimination of Redundant Loads and Stores - don’t load a value you already have or store twice in a row when you can combine it
  - Constant Folding - move calculations from runtime to compile time (3*2 becomes 6)
  - Constant Propagation - substitute constant references with their values, killing the stored values so as not to bother loading them again.
  - Common Subexpression Elimination - eliminate redundant calculations
  - Copy Propagation - while two registers have the same value, only use one of them
  - Strength Reduction - replace operations with cheaper ones. Ex: add becomes shift
  - Elimination of useless instructions: adding 0, mult by 1
  - Exploitation of the Instruction Set - use simpler instructions, use base plus displacement addressing (only one unique to peephole)

- Quick, small overhead
- Useful to exploit idiosyncrasies and clean up suboptimal code idioms
Redundancy Elimination in Basic Blocks

- Local optimization
- Start by matching basic blocks with fragments of syntax tree
  - Adjacent tree nodes when using in order
- Translate syntax tree of a basic block into an expression DAG or using value numbering
  - Redundant loads, computations merge into nodes with multiple parents
- Value Numbering
  - Symbolically equivalent computations get a name so code can reference that instead of the computation
  - Keep track of loaded and computed values in a directory
    - Also keep: record for each register about whether it should be used under its name, replaced by another, or replaced by a constant
    - Store which register holds the current value for constants and some values
    - What register might hold the result
Global Redundancy and Data Flow Analysis

- Eliminate redundant loads and computations *between* basic blocks
  - Translate code into static single assignment form
  - Perform global value numbering to identify redundancy
- Next step: global common subexpression elimination, and constant and copy propagation
- Global redundancy can also catch local redundancies at the same time
- Global Common Subexpression Elimination aims to identify places where an instruction computing a value for a given (virtual) register is redundant and can therefore be taken out of the program
- Forward Data Flow and Backward Data Flow
  - FDF: available expression analysis
Loop Improvement

- Move invariants into the header so they don’t get evaluated inside the loop
  - Save n-1 instructions where n is the amount of times the loop runs
  - To find invariants, save markers when linearizing syntax tree (for structured languages)
  - Reaching Definitions: tracking where an operand was defined
    - Loop invariant if each operand: is constant, has reaching definitions all outside the loop, has only one reaching definition inside the loop as long as the instruction defining it is invariant

- Reduce maintaining induction variables (registers)
  - Loop indices, subscript computations, variables modified in the loop
  - Give opportunities for strength reduction
  - Usually redundant so can be simplified
Instruction Scheduling

- Instruction scheduling (two rounds) then register allocation
  - Can improve for loops before the target code generation (reasons coming)
- On a pipelined machine, can best optimize performance by having the compiler make the best use of pipeline and filling it
- Delays can happen when an instruction
  - Has to wait for another function to be done with a functional unit
  - Waits for data computed by other instructions
  - Must wait for the target of a branch to be executed
- Scheduling heuristic options
  - Nodes that can start right away (without stalling)
  - Favor nodes with maximum delay at the end of the block
  - First come first served
  - Favor nodes with largest numbers of children in the DAG
  - Favor nodes that would be the final time using a register (reduce register pressure)
  - Favor nodes using a pipeline that hasn't received an instruction recently
Loop Improvement

- **Loop Unrolling**
  - Restructure loop body to include parts from multiple iterations
- **Improve cache performance**
- **Increase parallelization opportunities**
- **Loop Reordering**
  - Cache Optimization can be used to tile or block the loops.
  - Loop Distribution/fission/splitting: split one loop into multiple loops
  - Loop Fusion/jamming: combine multiple loops into less loops
- **Loop Dependences**
  - Loop-carried dependencies hold orders of iterations for loops
- **Parallelization**: best to not put parts with dependencies on each other in parallel, unless you take care of synchronization
Register Allocation

- Can be performed for blocks of code
  - A variable can be given a register for a whole subroutine
  - Ex: loop indices, scalar local variables
- Heuristics for register allocation / graph coloring
  - Identify virtual registers that can’t share architectural registers with each other
    - Construct register interference graph, where nodes are virtual registers
    - Minimal coloring of the graph will map virtual registers to the smallest number of architectural registers
- Issues
  - Architectural registers differ by machine (some are specified for data types)
  - Sometimes there aren’t enough architectural registers, coloring the graph doesn’t work to minimize architectural registers. Can use heuristics to split registers and spill data to memory when not being used.
Book used

Programming Language Pragmatics, 3rd AND 4th Editions