Overview

• Goals
• Organizational details
• Overview of compilers
Goals

What is the state of the art in compilers and runtimes?

• Focus on how to do research / push boundaries on existing compiler technology
• Mix of classical and cutting edge

Non-goal: How to build a compiler? - see CSE 401

• We won’t cover lexing, parsing, symbol tables, ...
• We won’t capture “everything” in research
How to do research?

Critical analysis on original research
- Focus on original papers: what did they get right? what did they miss?

Background and tools
- Program analysis, optimization frameworks

Getting your hands dirty
- Implementation, performance analysis
Course details

Detailed syllabus / schedule
1-2 research papers / lecture

• Read ahead of lecture
• No books required ... but see syllabus for background reading

Before each lecture, post a question on the reading

3 programming assignments
Programming Assignments

3 assignments: Start compiler

- Control-flow / dominators
- SSA construction / optimization
- Profile-guided optimization

May work in pairs
Grading

Breakdown

- Programming assignments: 80%
- Reading questions: 20%
Links

http://cs.washington.edu/501/
https://catalyst/uw.edu/gopost/board/vsmenon/32536
A Compiler

A language to any language

Contrast with interpreter
As a matter of engineering, it’s a bad idea to try and build a monolithic compiler. Instead, we separate our concerns.

Intermediate language (IL) = Intermediate representation (IR)
The UNCOL Problem
Naturally we might imagine multiple front ends and back ends sharing the same optimizer.

Notice that the optimizer looks just like another compiler.

We might have many optimizers and/or a variety of ILs.
Optimization

An optimizer tries to:

- Eliminate overhead from language abstractions
- Map source program onto target language (e.g., hardware) efficiently
- Equal the efficiency of a good target-language programmer

An optimizer might have any one of several goals:

- Fastest code
- Smallest code
- Code with fewest page/cache misses
**Expectations**

A compiler ought to provide *robust* optimization

- Small changes in the input shouldn’t produce wild changes in the output
- Create an expectation of excellent code quality
- Broaden the set of inputs that produce good code

We’d like to attain a large fraction of peak performance (*not 5%)*

Some inputs have always produced good code

- The 1st Fortran compiler focused on DO loops
- PCC did well on assembly-like programs
Good optimizing compilers are crafted, not just assembled

- Consistent philosophy
- Careful selection of transforms
- Thorough application of those transforms
- Careful use of algorithms and data structures
- Attention to the output code

Compilers are engineered objects

With all of these tradeoffs, results are sometimes unexpected
Safety

The 1st principle of optimization:
*The compiler must preserve the code’s meaning*

When can the compiler transform the code?
- Original & transformed code must have the same final state
- Variables that are visible at exit
- Equality of result, not equality of method

Formally:
For 2 expressions $M$ and $N$, we say that $M$ and $N$ are observationally equivalent if and only if, in any context $C$ where $M$ and $N$ are closed (that is, have no free variables), evaluating $C[M]$ and $C[N]$ either produces identical results or neither terminates - Plotkin, 1975
In practice, compilers use a simpler notion of equivalence:
If, in their actual program context, the result of evaluating $M$
cannot be distinguished from the result of evaluating $N$, the
compiler can substitute $M$ for $N$

This restatement ignores divergence.
If $M$ is faster, the transformation is profitable.

- Compiled code always executes in some context
- Optimization is the art of capitalizing on context
- Lack of context implies fully general (i.e., slow) code

Some compilers employ a lesser standard:
- Correct behavior for standard conforming code
- Undefined behavior for other code
Unsafety

You as a compiler writer must decide if it’s worth the risk of doing this kind of optimization. It’s difficult for the compiler to distinguish between the safe and dangerous cases here. For example, many C compilers perform risky optimizations because the compiler writer has assumed a C programmer can understand the problems and take steps to remedy them at the source level. It’s better to provide the maximum optimization, even if it’s dangerous, than to be conservative at the cost of less efficient code.

No!

- You must not violate the first principle
- Without correctness, you might as well emit a return and be done.
Profitability

The compiler should only transform code when it helps
- Eliminating one or more operations
- Replacing an expensive operation with a cheaper one
- Moving an operation to a place where it won’t execute so often

Sometimes we can prove profitability
- Fold a constant expression into an immediate operation

Sometimes we must guess
- Eliminating a redundant operation in a loop

Sometimes we can’t tell
- Inlining
A couple of slides ago, we mentioned context. The more we know about the context, the better we can make the code.

Leads us to consider:
- global optimization (entire routine)
- interprocedural analysis (many routines)
- link-time optimization
- run-time optimization

All attempts to enlarge the context
Our History *(according to Keith Cooper)*

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1955</td>
<td><em>Commercial optimizing compilers</em></td>
</tr>
<tr>
<td></td>
<td>Fortran</td>
</tr>
<tr>
<td></td>
<td>Cobol</td>
</tr>
<tr>
<td>1960</td>
<td><em>Academics try to catch up with industry</em></td>
</tr>
<tr>
<td></td>
<td>Algol 60</td>
</tr>
<tr>
<td></td>
<td>Early algorithms for code generation</td>
</tr>
<tr>
<td></td>
<td>Relating theory to practice (Lavrov)</td>
</tr>
<tr>
<td></td>
<td>Alpha project in Novosibirsk (Ershov)</td>
</tr>
<tr>
<td>1965</td>
<td><em>Technology begins to spread</em></td>
</tr>
<tr>
<td></td>
<td>PL/1</td>
</tr>
<tr>
<td></td>
<td>Fortran H (Medlock &amp; Lowry)</td>
</tr>
<tr>
<td></td>
<td>Algol 68</td>
</tr>
<tr>
<td></td>
<td>Value numbering (Balke, Ershov)</td>
</tr>
<tr>
<td></td>
<td>Simula 67</td>
</tr>
<tr>
<td></td>
<td>Literature begins to emerge (Allen)</td>
</tr>
<tr>
<td>Year</td>
<td>Event</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>1970</td>
<td>The literature explodes and optimization grows up</td>
</tr>
<tr>
<td></td>
<td>Cocke &amp; Schwartz, Allen-Cocke catalogue</td>
</tr>
<tr>
<td></td>
<td>Theory of analysis (Kildall, Allen &amp; Cocke)</td>
</tr>
<tr>
<td></td>
<td>Interprocedural analysis (Spillman)</td>
</tr>
<tr>
<td></td>
<td>Strength reduction, dead-code elimination (SETL)</td>
</tr>
<tr>
<td></td>
<td>Expression-tree algorithms (Aho, Sethi, Ullman)</td>
</tr>
<tr>
<td>1975</td>
<td>Academics try to catch up with industry</td>
</tr>
<tr>
<td>Pascal</td>
<td>Full literature on data-flow analysis</td>
</tr>
<tr>
<td>CLU</td>
<td>Strength reduction (Cocke &amp; Kennedy)</td>
</tr>
<tr>
<td>Alphard</td>
<td>Partial redundancy elimination (Morel &amp; Renvoise)</td>
</tr>
<tr>
<td>Lisp</td>
<td>Inline-substitution studies (Scheiffler, Ball)</td>
</tr>
<tr>
<td></td>
<td>Tail-recursion elimination (Steele)</td>
</tr>
</tbody>
</table>
## History

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
</table>
| 1980 | *Programming environments & new architectures*
|      | Smalltalk: Incremental analysis (Reps, Ryder, Zadeck) |
|      | Ada: Incremental compilation (Schwartz) |
|      | Scheme: Interprocedural analysis (Myers, Cooper) |
|      | RISC compilers (PL.8, MIPS) |
|      | Graph-coloring register allocation (Chaitin, Chow) |
|      | Vectorization (Wolfe, Allen) |
| 1985 | *Resurgence of interest in classical optimization* |
|      | C++: SSA form (Cytron, et al.) |
|      | ML: Constant prop (Wegman & Zadeck, Torczon) |
|      | Modula-3: Software pipelining (Lam) |
|      | Pointer analysis (Ruggeri) |
## History

**1990**  
*Architects (and memory speed) drive the process*

**F90**  
Hierarchical allocation (Callahan & Koblenz)  
Scalar replacement (Carr)  
Cache blocking (Wolf)  
Prefetch placement (Mowry)  
Commercial interprocedural optimizers (Convex)

**1995**  
*The internet and SSA come of age*

**Java**  
JIT compilers (everyone)  

**Perl**  
Code compression (Franz)  
SSA formulations of old algorithms  
Compile to VM (Java)  
Memory-layout optimizations (Smith)
2000
Javascript
Lua
Ruby
...

Dynamic compilers are everywhere

It’s hard to get much perspective on the past decade, but it’s clear that evolution continues at an astonishing rate.