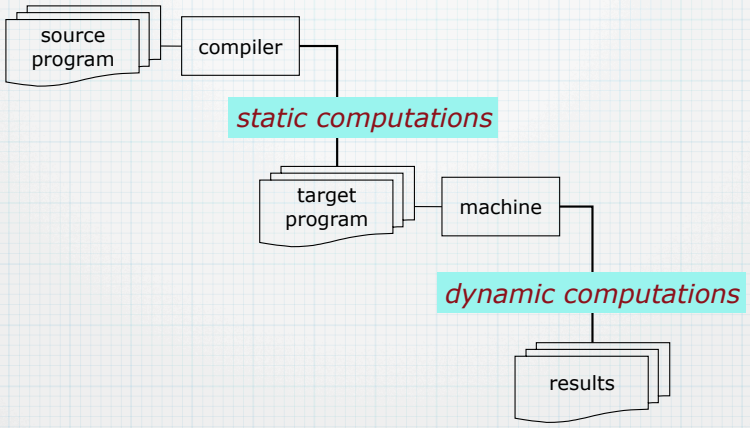
15-745 Optimizing Compilers

Peter Lee

Spring 2006

What is a compiler?

* A compiler translates **source programs** into **target programs**
* Source programs are usually in a high-level, human-readable form that specifies both **static** (compile-time) and **dynamic** (run-time) computations
* Target programs are usually machine executable and specify only run-time computations



Focus of this course

* This course is about **optimizing** compilers
  * methods for automatically improving the quality of the target programs, usually for **faster execution**
  * usually based on a **semantics-based analysis** of the program

Compilers are fundamental

* A very long history of study in CS
* Every new machine architecture defines standard calling conventions and comes with an optimizing compiler
* Chip performance is measured, in large part, by performance on programs written in C
* So as a practical matter, the compiler is an integral part of the machine architecture

"Thompson and Ritchie were among the first to realize that hardware and compiler technology had become good enough that an entire operating system could be written in C, and by 1978 the whole environment had been successfully ported to several machines of different types."

— Eric S. Raymond,
"The Cathedral and the Bazaar"

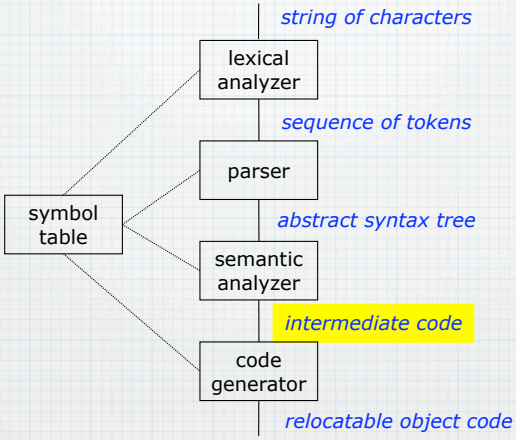
You will probably write a compiler

* Compilers make big, amazing, direct use of theory ideas
* Almost all of the key ideas in their design are important in other problem domains
* You will end up using compiler design principles in almost every research or software development project
* Compilers are also fun!

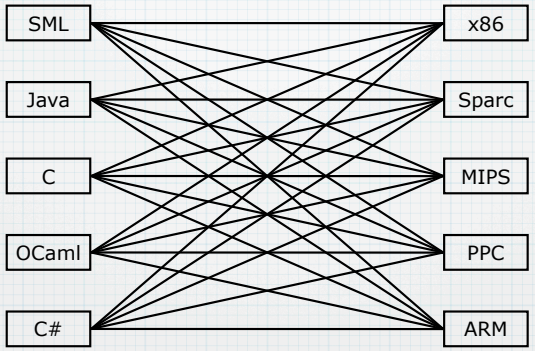
"I'd rather have a search engine or a compiler on a deserted island than a game."

- John Carmack,
Co-founder, id Software

Compiler structure

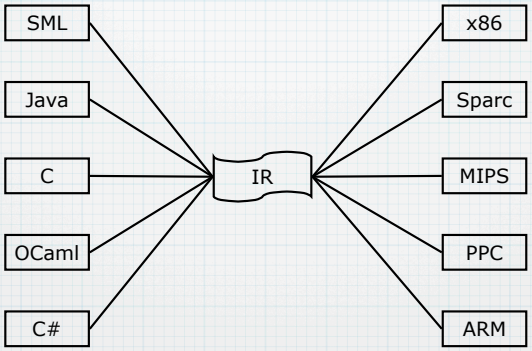


The UNCOL argument



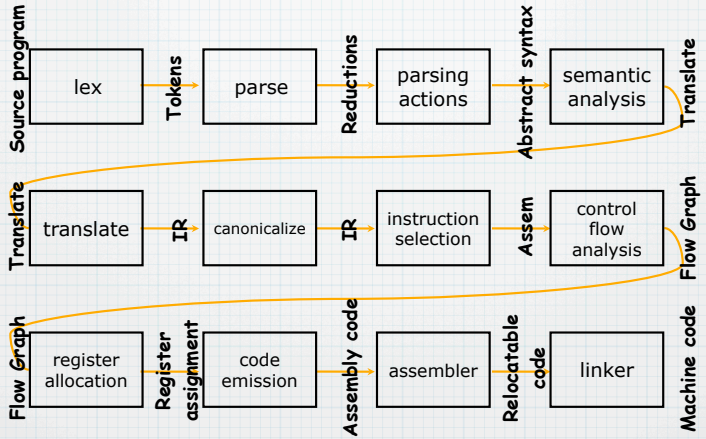
n×m compilers!

The UNCOL argument



vs n+m compilers

More realistically...



This course

* Theory and practice of modern optimizing compilers
  * major focus: analysis and optimizing transformations of the intermediate representation
  * some on target code transformations
  * some on run-time systems
  * no lexing, parsing, typechecking

Optimization

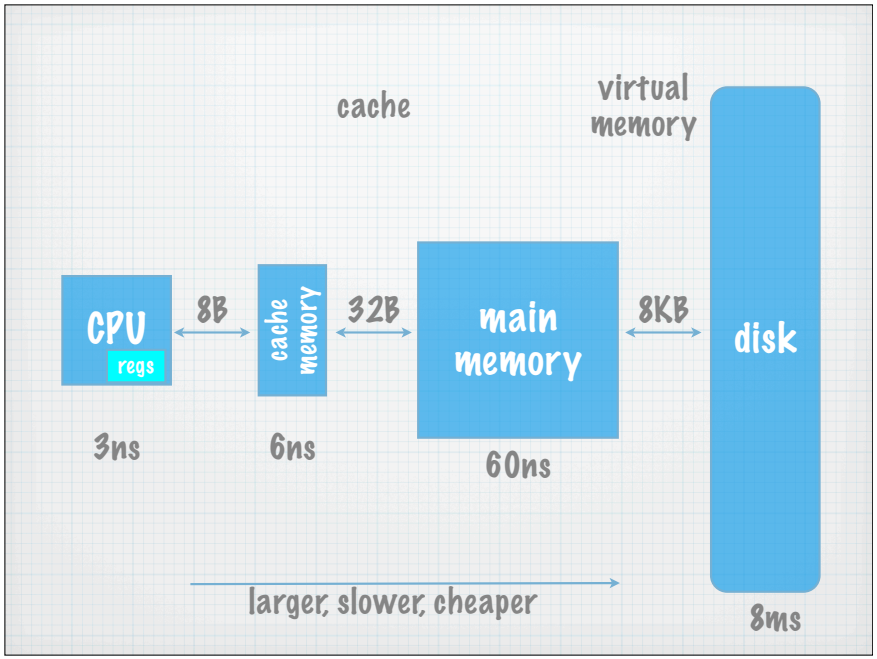
* The **most important function** of a compiler is **static code checking/analysis**
* But **almost as important** is **optimization**
* Optimization was **the driving force** behind the modern RISC microprocessors...
* ...and today the language+compiler is the driving force behind architecture developments such as multi-core processors

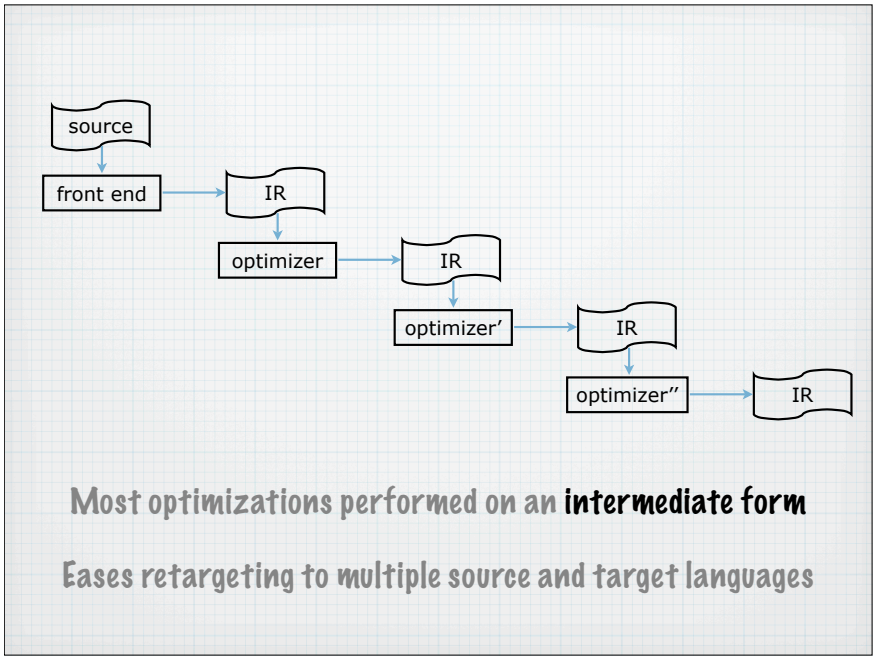
What is optimization?

* Informally: Transform a program into an **equivalent** but **better** form
  * [Note: **Red** denotes dangerous hand-waving!]
* "Optimize" is a bit of a misnomer
  * the results are almost never optimal
  * the Full Employment Theorem
  * anyway, what is meant by "better"?

Our focus: Run-time performance

* How to improve run-time performance?
  * Reduce the number of instructions
  * Replace "expensive" instructions with "cheap" ones
  * Reduce memory costs
  * Increase parallelism





Most optimizations performed on an **intermediate form**

Eases retargeting to multiple source and target languages

Ingredients for an optimization

* Identify an opportunity
  * applicable to many programs
  * affects key parts of programs
  * amenable to "efficient enough" algorithm
* Formulate the optimization problem
* Pick a representation
* Develop a static analysis
  * detect when legal and desirable
* Implement the code transformation
* Evaluate experimentally (and repeat!)

* Most optimizations we can imagine or desire are not susceptible to this recipe

```
SumFrom1toN (int max) {
  sum = 0;
  for (i=1; i<max; i++) sum += i;
  return sum;
}
```

optimizer

```
inline SumFrom1toN (int max) {
  return max > 0 ?
         ((max+max*max)>>1) : 0;
}
```

Some important optimizations

- machine independent
  - algebraic simplification
  - constant propagation
  - constant folding
  - common subexpression elimination
  - dead-code elimination
  - loop-invariant code motion
  - induction-variable elimination

Some important optimizations

- machine dependent
  - jump optimization
  - register allocation
  - instruction scheduling
  - strength reduction

Local optimizations

- Some optimizations are **local**, meaning that the legality and desirability for a statement or expression can be determined in isolation from the rest of the program

Algebraic simplifications

a*1	+	a
a/1	+	a
a*0	+	0
a+0	+	a
a-0	+	a
a = b+1 c = a-1	+	c=b

Use algebraic identities to simplify arithmetic

Global optimizations

- The most important optimizations are **global**
- They typically require a semantics-based analysis of the entire procedure (or program)
  - **dataflow analysis**
  - **abstract interpretation**

Dead-code elimination

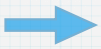
```
debug = false;
  ...

if (debug) { ... }
 ...
```

If code will never be executed or its result and effect will never be used, eliminate it

Constant propagation

```
a = 5;
b = 3;
 ...
n = a + b;
for (i=0; i<n; ++i)
  ..
```

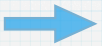


```
a = 5;
b = 3;
 ...
n = 5 + 3;
for (i=0; i<n; ++i)
  ..
```

If a and b can be determined to be constants, then replace them

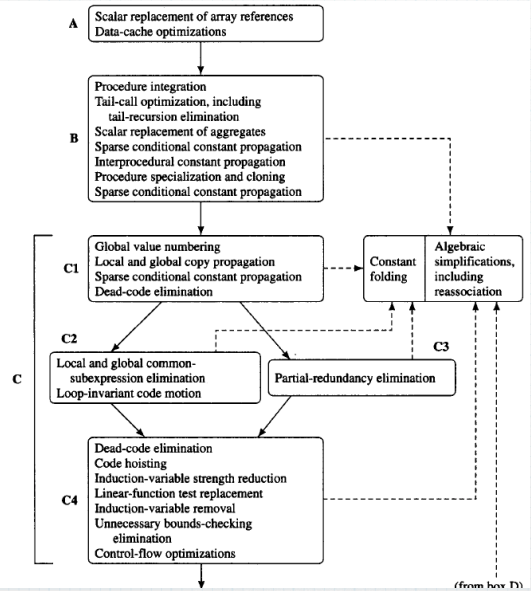
Constant folding

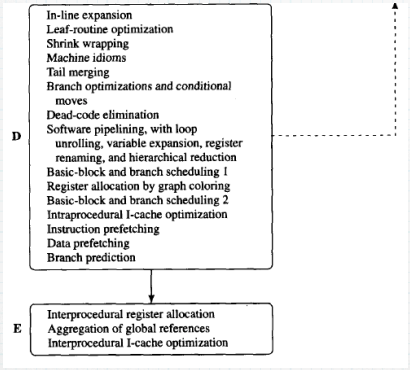
```
a = 5;
b = 3;
 ...
n = 5 + 3;
for (i=0; i<n; ++i)
  ..
```



```
 ...
n = 8;
for (i=0; i<n; ++i)
  ..
```

Local and global optimizations can trigger additional local and global optimization opportunities



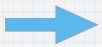


Redundant computations

* The detection and **elimination of** (fully or partially) **redundant computations** is perhaps the major goal of an optimizer

Common subexpression elimination

```
a = c*d;
  ...
  ...
d = (c*d + t) * u;
```



```
a = c*d;
  ...
  ...
d = (a + t) * u;
```

If an expression's evaluation is redundant, then reuse the previous result

Loop-invariant code motion

```
for (i=0; i<100; ++i)
  for (j=0; j<100; ++j)
    for (k=0; k<100; ++k)
      a[i][j][k] = i*j*k;
```

```
for (i=0; i<100; ++i)
  for (j=0; j<100; ++j) {
    t1 = a[i][j];
    t2 = i*j;
    for (k=0; k<100; ++k)
      t1[k] = t2*k;
```

Very important because loop bodies execute frequently

Code motion is tricky

```
int *a;
int n;
...
scanf("%d", &n);
for (i=0; i<n; ++i)
  for (j=0; j<n; ++j)
    for (k=0; k<n; ++k) {
      f = q/p;
      a[i][j][k] = f*i*j*k;
```

Oops!

```
int *a;
int n;
...
scanf("%d", &n);
f = q/p;
for (i=0; i<n; ++i)
  for (j=0; j<n; ++j) {
    t1 = a[i][j];
    t2 = i*j;
    for (k=0; k<n; ++k)
      t1[k] = f*t2*k;
  }
```

Semantics-based analyses

* We will spend considerable time on the theory and practice of dataflow analysis
* This is the major semantics-based analysis used by most optimizing compilers
* Some particularly difficult problems, such as predicting aliasing or heap pointer structure, seem less susceptible to dataflow analysis and often use other methods

Code-level optimizations

* Some optimizations are **code-level**, i.e., performed on the target code (or some machine-dependent intermediate form)

Jump optimizations

```
    cmp d0,d1
    beq L1
    br  L2
L1:  ...
  ...
L2:  ...
  ...
```

```
    cmp d0,d1
    bne L2
L1:  ...
  ...
L2:  ...
  ...
```

Simplify jump and branch instructions

Strength reduction

b * 2 + b + b + lsh(b)

-1 * b + -b

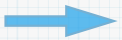
On some processors, some operations are significantly less expensive than others

Memory optimizations and parallelization

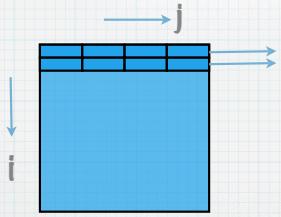
* Some optimizations are fully or partially machine-dependent and yet are performed on a machine-independent representation

Cache optimizations

```
for (j=0; j<n; ++j)
  for (i=0; i<n; ++i)
    x += a[i][j];
```



```
for (i=0; i<n; ++i)
  for (j=0; j<n; ++j)
    x += a[i][j];
```



Loop permutation can sometimes improve the spatial locality of memory accesses

How this course works

Course staff

* Peter Lee
  * http://www.cs.cmu.edu/~petel
* Mike DeRosa
  * http://www.cs.cmu.edu/~mderosa
* Angie Miller
  * amiller@cs.cmu.edu

Prerequisites

* undergraduate architecture course
  * e.g., 15-213
* undergraduate compiler design course
  * e.g., 15-411
* proficiency in SML, OCaml, or Java programming
* basic understanding of architecture, especially x86

Major activities (tentative)

* attendance at lectures
* ~3 optimizer tasks (teams of 2)
* significant project (teams of 2), with proposal
* textbook readings
* research paper readings and in-class presentations
* in-class proposal and project presentations
* either an exam or take-home worksheet

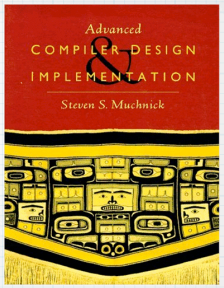
Tentative grading scheme

* Tasks: 30%
* Project: 30%
* Exam: 20%
* Participation: 20%
  * in class; paper presentations; project presentations

Tasks

* We will assign tasks to implement specific optimizations in a compiler
  * this guarantees that you will definitely understand and gain experience with some basics
* We will give you a basic compiler
  * a toy compiler, but the optimization tasks will be realistic
* You will evaluate your own compiler, submit the results, and this is your grade

Readings



* Textbook:
  * Steven Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufmann, 1997
* Reading list assignments:
  * In-class presentations, as time allows
  * Reading list is forthcoming

Getting the textbook

* It may be possible to get through the course without the Muchnick textbook, or with a different textbook
* But we recommend that you buy a copy
  * a few copies in the bookstore
  * also on sale on-line

Course schedule

* Web site forthcoming
  * will contain schedule of lectures, tasks, etc
  * also lots of support material and writeups, including task information

Communication

* Please send email to Mike (mderosa@cs) to get put onto the class mailing list