CPSC 510 (Fall 2007, Winter 2008) Compiler Construction II

Robin Cockett

Department of Computer Science
University of Calgary
Alberta, Canada

robin@cpsc.ucalgary.ca

Sept. 2007

Introduction to the course

The course

A first look at general optimization ..

Architecture dependent optimizations

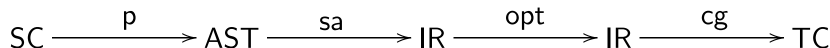
Bibliography

What is a compiler?

- A compiler translates **source programs** into **target programs**
- Usually source programs are a high-level human-readable form while target programs are usually machine executable.
- Programs specify a **dynamic** (run-time) behaviour and are subject to certain **static** (compile-time) requirements.
- Compilers must correctly implement the behaviour specified by programs .... and must do so in a reasonable amount of time producing reasonable efficient target code .

.

Structure of a compiler



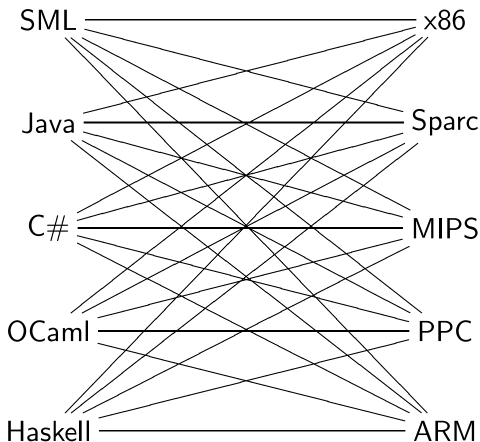
SC	=	Source code
AST	=	Abstract syntax tree
IR	=	Intermediate representation
TC	=	Target Code
p	=	lexing and parsing
sa	=	semantic analysis
opt	=	optimization
cg	=	code generation

This course: the last two steps!

Compilers are fundamental

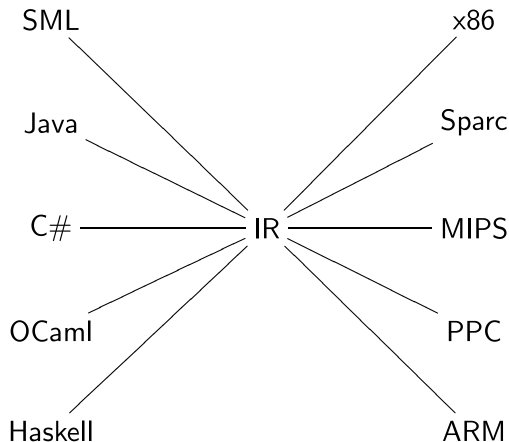
- Compiler technology has a long history.
- Theoretical ideas are heavily used.
- Modern machine architectures are unusable without an (optimizing) compiler.
- Architecture and compiler technology interacts
  - RISC/CISC micro-processor architecture
  - Pipeline architectures ...
  - Multi-core processors ...
- Ideas from compilers are used everywhere ...

Organizing compiler development



.... n times m compilers.

Organizing compiler development



.... *n* plus *m* compilers.

Organizing compiler development

The strategy:

- Have good intermediate representations!
- Do generally applicable optimizations at the level of the intermediate representation.
- Do architecture specific optimizations when generating code ...

Focus of the course ...

This course is about **optimizing** compilers (parsing, lexing, and type checking is assumed ...)

- Main emphasis on general optimization techniques:
  - Control flow analysis
  - Code transformations
  - Instruction selection and scheduling.
  - Register allocation
- Secondary concerns:
  - Compiling should not take forever
  - The code footprint should be reasonable ...
  - Architectural features pipelining, parallelism.

Focus of the course ...

Compilers directly use theoretical ideas ...

- Algorithms are everywhere ...
- Data structures are everywhere ...
- Data-flow analysis uses graph theory
- Transformations are based on the **semantics** of the computation.

The course

1. First semester: the theory and practise of optimization
   - Basic register allocation and instruction selection
   - Control flow graphs (CFG), dominators, loop structure, and reducibility
   - Data-flow analysis, solving fixed point equations on CFGs
   - Static single assignment form (ψ-functions), translation into a control flow language, program reduction.
   - Arrays, garbage, functions, ...
   - Code generation issues: instruction scheduling ...

The course

II. Second semester: implementing a substantial component of a compiler

- Project initiation: project description and website (25
- Four project milestones (25
- Final presentation and project submission (50

Evaluation of the course

First semester:

- Midterm exam (25%)
- Final exam: take-home exam (35%)

Assignments:

- Register allocation and instruction selection (10%)
- Data-flow analysis (10%)
- Code reducing in a control flow language (10%)

What is better?

Focus on execution speed ...

- Reduce number of instructions
- Decrease memory usage
- Increase locality
- Replace expensive instructions with cheap ones
- Increase pipelining and parallelism
- Use special architectural features

WANT: optimized program performs no worse!

Architecture independent optimizations ...

The above optimizations, except the first, depend on the machine architecture. BUT there are many general optimizations steps which **reduce** the number of instructions ...

1. Constant propagation, constant folding
2. Dead-code elimination, common subexpression elimination
3. Store/load elimination
4. Strength reduction and algebraic simplification
5. Loop invariant code motion, loop unrolling
6. Loop fusion loop, and loop splitting
7. Induction variable elimination

These types of optimization are the main focus of the course ...

Constant propagation ...

Replace variables by constants if they are known ...

```
a = 5; b= 27;                      a = 5; b= 27;
...                          ⇒     ...
n = a + b;                         n = 5 + 27;
for (i=0; i<n; ++i)                for (i=0; i<n; ++i)
...                                ...
```

Constant folding ...

Evaluate expressions involving constants ...

```
a = 5; b= 27;
...
n= a + b;
for (i=0; i,n; ++i)
...
```

⇒

```
a = 5; b= 27;
...
n = 32;
for (i=0; i<n; ++i)
...
```

Common subexpression elimination ..

Replace expressions which have already been evaluated by a variable which holds the value of the computation ...

a = c * d;		a = c+d;
...	⇒	...
d = (c*d +t);		d = a+t;
...		...

Eliminating dead code is less easy: how do we know that some code will *never* be used – a **global** analysis is needed.

Algebraic simplification ...

Replace and expression with a simpler expressions which known to be equal using algebraic identities:

```
a = c * 0;      a = 0;
b = c+0;        b = c;
c = d*1         c = d;
m = 2*n         m = shift(n 1);
```

The last is a strength reduction as shifting is cheaper than multiplying ...

Code motion ..

Move expressions which are invariant in loops out of loops:

```
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; ++j)
            t1[k] = t2 * k;
```

This is a very important optimization but it is tricky!!

So what is the difficulty?

These optimizations may look easy ...

.. but they are not. The big question is how to do them automatically! They also interact: optimizing in one area can make available an optimization in another etc.

ORGANIZING them so that one optimization does not get in the way of another is crucial! ... and so, in the end, everything has been done which is (reasonably) possible.

REMEMBER: a small optimization in a loop can save hours of CPU time!

Using the machine architecture to advantage is important too ...

Locality ...

Why is memory usage important?

Location	Type of memory	Quantity	Access time
CPU	Registers	32B	3ns
CPU	Cache	2KB	6ns
CPU	Main memory	2MB	60ns
Hard drive	Disk	2GB	8ms

Memory hierarchy – the importance of locality

Register allocation

$$
x = (x+y)*z \quad \Rightarrow \quad
\begin{aligned}
r1 &= M[x]; \\
r2 &= M[y]; \\
r1 &= r1+r2; \\
r2 &= M[z]; \\
x &\leftarrow r1*r2;
\end{aligned}
$$

Register allocation, in general, is an NP-complete problem.
However, many of the register allocation problems particular to compilers can be solved in near linear time.
Poor register allocation leads to **spilling** and loss of locality .... a big hit!

Instruction selection

Most architectures provide a variety of instructions which cam be used in different ways to perform a given computation.
There are dynamic algorithms for instruction selection techniques which minimize the number/cost of the instructions.

Pipelining..

Consider the loop

```
for (1=1; i ≤ m; i++)
  for (1=1; j ≤ n; j++)
    x[i] = x[i] + y[i,j];
```

Pipelined execution for $m = 4$ and $n = 3$:

Time	Processor 1	Processor 2	Processor 3
1	x[1] += y[1,1]		
2	x[2] += y[2,1]	x[1] += y[2,1]	
3	x[3] += y[3,1]	x[2] += y[2,2]	x[1] += y[1,3]
4	x[4] += y[4,1]	x[3] += y[3,2]	x[2] += y[2,3]
5		x[4] += y[4,2]	x[3] += y[3,3]
6			x[4] += y[4,3]

Instruction scheduling

Instruction should be scheduled to make use of pipelining and parallelism.

Texts ...

Modern Compiler Implementation	Appel
Building Optimizing Compilers	Morgan
Compiler Design and Optimization	Muchnik
Compilers: Principals, Techniques, and Tools	Aho Sethi Ullman

Journals ...

IEEE	Computer
	Transactions on Computers
	Concurrency
	Transactions on Parallel and Distributed Systems
ACM	TOPLAS - Transactions on Programming Languages and Systems
	Transactions on Computer Systems
JPDS	Journal of Parallel and Distributed Computing
JSC	Journal of Supercomputing
JPP	International Journal of Parallel Programming
PC	Parallel Computing
JPL	Journal of Programming Languages

Conferences ...

PLDI	ACM Symposium on Programming Language Design and Implementation
POPL	ACM Symposium on Principles of Programming Languages
ASPLOS	ACM Symposium on Arcitecture Support for Programming Languages and Operating Systems
PPOPP	ACM Symposium on Principles and Practise of Parallel Programming
ICPP	International Conference on Parallel Processing
ICS	International Conference on Super Computing
PACT	Parallel Architectures an Compilation Techniques