CERN openlab Summer 2006: Compiler Overview

Martin Swany, Ph.D.

Assistant Professor, Computer and Information Sciences, U. Delaware, USA

Visiting Helsinki Institute of Physics (HIP) at CERN

swany@cis.udel.edu, Martin.Swany@cern.ch

What is a Compiler?

- A **compiler** is a program that translates a program written in one computer language (called the *source code*) into a resulting output in another computer language (often called the *object* or *target code*)
  - Generally, the source code is a high-level language and the object code is machine language
- Compilation entails semantic *understanding* of what is being processed
  - pre-processing does not
- Understanding compilers can help one write better code

Goals and Overview

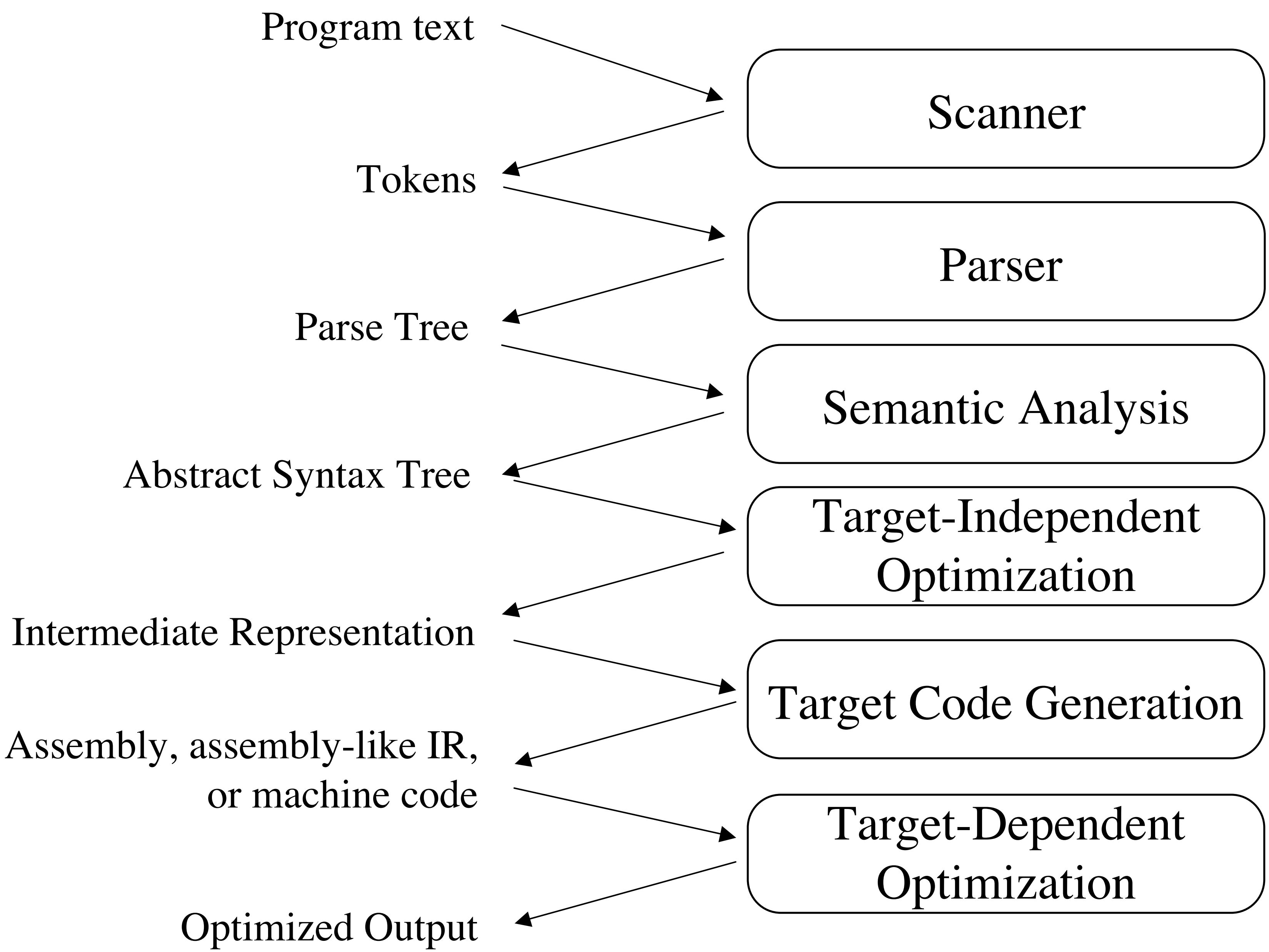
<u>Goals:</u>

- Cover basic terminology and key ideas (without going into too much depth)
- Set the stage for details of optimization in later lectures

<u>Talk Outline:</u>

- Basic functional overview
- Optimization
- Current Topics

Basic Functional Overview



Functional Overview: Scanning

- ***Scanning***:
  - divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
  - we can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
  - scanning is recognition of a *regular language*, e.g., via deterministic finite automaton (DFA)
- Scanning is lexical analysis
  - Lexical - of or relating to the words or vocabulary of a language

Functional Overview: Parsing

- ***Parsing*** is recognition of a *context-free language*, e.g., via push-down automaton (PDA)
  - Parsing discovers the "context free" structure of the program
  - Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" of a state machine)
- Parsing looks at the syntax
  - <u>syntax</u> - the arrangement of words and phrases to create well-formed sentences in a language

Functional Overview: Semantic Analysis

- ***Semantic analysis*** is the discovery of *meaning* in the program
  - The compiler does what is called static semantic analysis. That's the meaning that can be figured out at compile time
  - Some things (e.g., array subscript bounds errors) can't be figured out until run time. Things like that are part of the program's dynamic semantics
- <u>semantic</u> - related to meaning in language

Functional Overview: Symbol Table

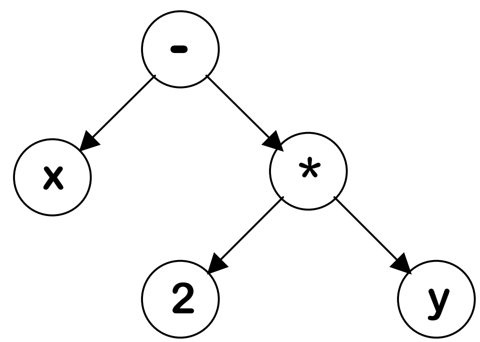
- ***Symbol table***: all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
  - This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

Functional Overview: Intermediate Representation

- ***Intermediate representation*** (IR) is the output of semantic analysis (if the program passes all checks)
  - IRs are often chosen for machine independence, ease of optimization, or compactness (these can be at odds)
- Many compilers actually move the code through more than one IR
- Different sorts of IRs have different properties and strengths
  - Structural
    - Graph oriented
  - Linear
    - Pseudo-code for an abstract machine
    - Easier to rearrange
  - Hybrid
    - Combination of graphs and linear code

Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed



x - 2 * y

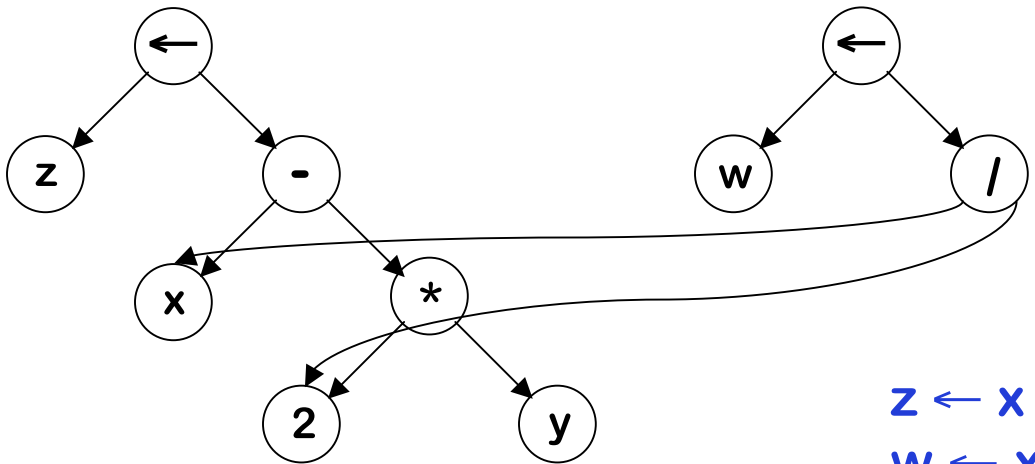
- Can use linearized form of the tree
  - Easier to manipulate than pointers

    `x 2 y * -` in postfix form

    `- * 2 y x` in prefix form

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



z ← x - 2 * y
w ← x / 2

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!

Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:

$$x \leftarrow y \ \underline{op} \ z$$

With 1 operator (*op*) and, at most, 3 names (x, y, & z)

Example:

$z \leftarrow x - 2 * y$ becomes

```
t ← 2 * y
z ← x - t
```

Advantages:

- Resembles many machines
- Introduces a new set of names *
- Compact form

Three Address Code: Quadruples

Naïve representation of three address code
- Table of k * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

RISC assembly code

load	1	Y	
loadi	2	2	
mult	3	2	1
load	4	X	
sub	5	4	2

Quadruples

Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	loadI	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Implicit names take no space

Static Single Assignment Form

- The main idea: each name defined exactly once
- This requires the introduction of ϕ-functions, a conceptual tool to represent the various possible values of a variable

Original

```
x ← …
y ← …
while (x < k)
   x ← x + 1
   y ← y + x
```

SSA-form

$$
\begin{array}{ll}
 & x_0 \leftarrow \ldots \\
 & y_0 \leftarrow \ldots \\
 & \text{if } (x_0 > k) \text{ goto next} \\
\text{loop:} & x_1 \leftarrow \phi(x_0, x_2) \\
 & y_1 \leftarrow \phi(y_0, y_2) \\
 & x_2 \leftarrow x_1 + 1 \\
 & y_2 \leftarrow y_1 + x_2 \\
 & \text{if } (x_2 < k) \text{ goto loop} \\
\text{next:} & \ldots
\end{array}
$$

Strengths of SSA-form

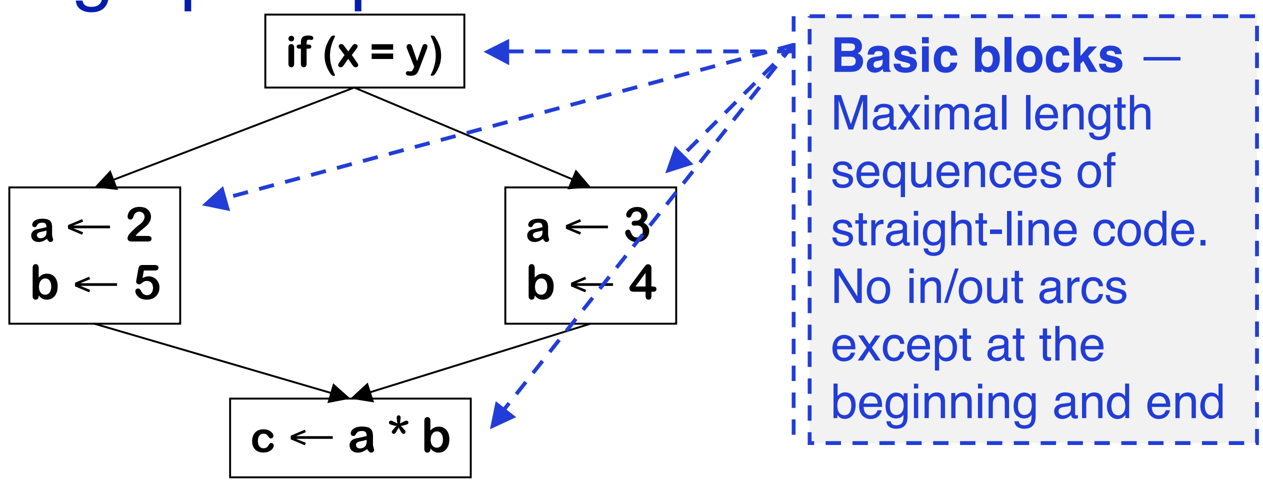
- Sharper analysis
- Hints about placement of invariant code
- (sometimes) faster algorithms

Control-flow Graph

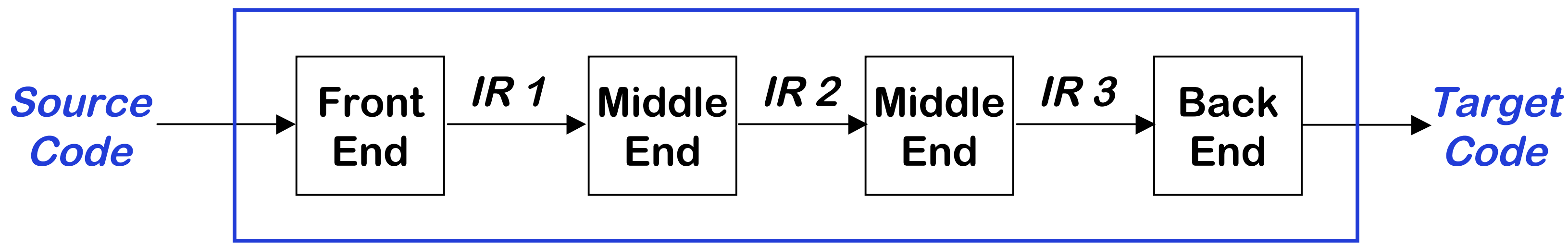
Models the transfer of control in the procedure

- Nodes in the graph are <u>basic blocks</u>
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

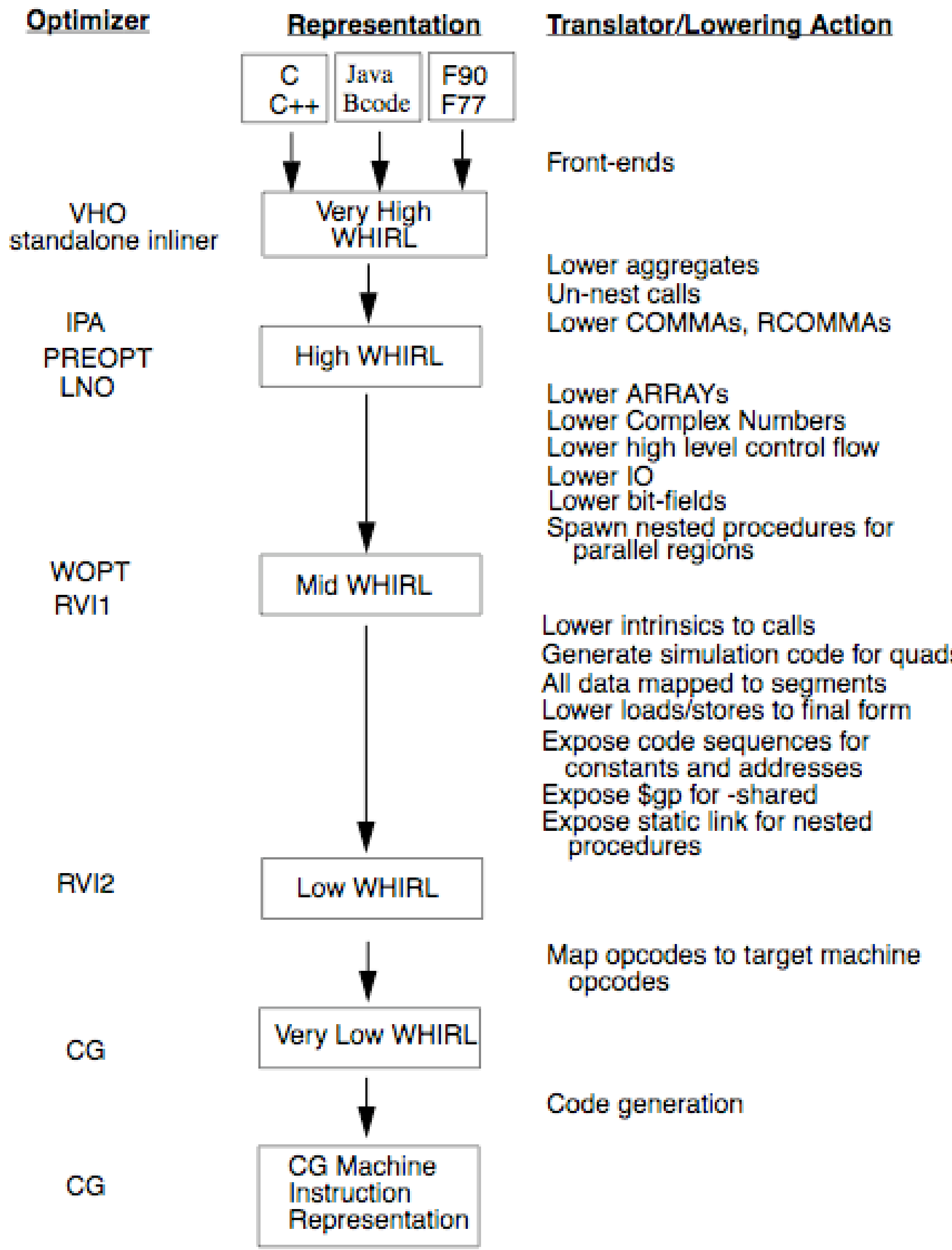
Example



Using Multiple Representations



- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - WHIRL intermediate format
    - Consists of 5 different *IR*s that are progressively more detailed



Optimizer
Representation
Translator/Lowering Action
C
C++
Java
Bcode
F90
F77
Front-ends
VHO
standalone inliner
Very High
WHIRL
Lower aggregates
Un-nest calls
Lower COMMAs, RCOMMAs
IPA
PREOPT
LNO
High WHIRL
Lower ARRAYs
Lower Complex Numbers
Lower high level control flow
Lower IO
Lower bit-fields
Spawn nested procedures for
parallel regions
WOPT
RVI1
Mid WHIRL
Lower intrinsics to calls
Generate simulation code for quads
All data mapped to segments
Lower loads/stores to final form
Expose code sequences for
constants and addresses
Expose $gp for -shared
Expose static link for nested
procedures
RVI2
Low WHIRL
Map opcodes to target machine
opcodes
CG
Very Low WHIRL
Code generation
CG
CG Machine
Instruction
Representation

WHIRL Example

Optimization Overview

- Simple target code generation gives us correct but highly suboptimal code
  - redundant computations
  - inefficient use of the registers, multiple functional units, and cache
- Next we turn to *optimization* (really *code improvement)*: the phases of compilation devoted to generating good code
  - Here we interpret “good” to mean *fast*
  - Some also consider program transformations to decrease memory requirements

Optimization Overview

- What makes code run faster?
- Make the code shorter
  - Shorter sequences of instructions with the same effect take less time to run
  - Reduce redundant operations
- Hide latency
  - Begin operations that take time as soon as possible and perform other independent tasks in the meantime
    - Loads and stores
    - Branches
    - Expensive operations
- Threaded through both is efficient use of machine resources
  - Registers in particular

Peephole Optimization

- A relatively simple way to significantly improve the quality of naive code is to run a *peephole optimizer* over the target code
  - Slide over the target code considering a several instruction window (a peephole), looking for suboptimal patterns of instructions
  - the patterns to look for are heuristic
    - patterns to match common suboptimal idioms produced by a particular front end
    - patterns to exploit special instructions available on a given machine
- These techniques are extended to wider scopes for more advanced optimizations

Peephole Optimization

- ***Elimination of redundant loads and stores***
  - The peephole optimizer can often recognize that the value produced by a load instruction is already available in a register

```
r2 ← r1 + 5
i ← r2
r3 ← i
r3 ← r3 × 3
```

becomes

```
r2 ← r1 + 5
i ← r2
r3 ← r2 × 3
```

Peephole Optimization

- ***Constant folding***
- A code generator may produce code that performs calculations at run time that could actually be performed at compile time
  - A peephole optimizer can often recognize such code

```
r2 ← 3 × 2
```

becomes

```
r2 ← 6
```

Peephole Optimization

- ***Constant propagation***
  - Sometimes we can tell that a variable will have a constant value at a particular point in a program
  - We can then replace occurrences of the variable with occurrences of the constant

```
r2 ← 4
r3 ← r1 + r2
r2 ←. . .
```

becomes

```
r2 ← 4
r3 ← r1 + 4
r2 ←. . .
```

and then

```
r3 ← r1 + 4
r2 ←. . .
```

Peephole Optimization

- ***Common subexpression elimination***
  - When the same calculation occurs twice within the peephole of the optimizer, we can often eliminate the second calculation:

```
r2 ← r1 × 5
r2 ← r2 + r3
r3 ← r1 × 5
```

  becomes

```
r4 ← r1 × 5
r2 ← r4 + r3
r3 ← r4
```

  - Often, as shown here, an extra register will be needed to hold the common value

Peephole Optimization

- ***Copy propagation***
  - Even when we cannot tell that the contents of register *b* will be constant, we may sometimes be able to tell that register *b* will contain the same value as register *a*
    - replace uses of *b* with uses of *a*, so long as neither *a* nor *b* is modified

```
r2 ← r1
r3 ← r1 + r2
r2 ← 5
```

becomes

```
r2 ← r1
r3 ← r1 + r1
r2 ← 5
```

and then

```
r3 ← r1 + r1
r2 ← 5
```

Peephole Optimization

- ***Strength reduction***
  - Numeric identities can sometimes be used to replace a comparatively expensive instruction with a cheaper one
    - In particular, multiplication or division by powers of two can be replaced with adds or shifts:

```
r1 ← r2 × 2
```

becomes

```
r1 ← r2 + r2 or r1 ← r2 << 1
```

```
r1 ← r2 / 2
```

becomes

```
r1 ← r2 >> 1
```

Peephole Optimization

- ***Elimination of useless instructions***
  - Instructions like the following can be dropped entirely:

    `r1 ← r1 + 0`

    `r1 ← r1 × 1`
- ***Filling of load and branch delays***
  - Loads and branches take a few instructions to complete and they can be started earlier while unconditional instructions execute
- ***Exploitation of the instruction set***
  - Particularly on CISC machines, sequences of simple instructions can often be replaced by a smaller number of more complex instructions

Loop Improvement

- Many codes spend much of their time in loops so those are a key focus of optimization
- Consider two classes of loop improvements:
  - those that move *invariant* computations out of the body of a loop and into its header, and
  - those that reduce the amount of time spent maintaining *induction variables*

Loop Improvement

- A *loop invariant* is an instruction (i.e., a load or calculation) in a loop whose result is guaranteed to be the same in every iteration
  - If a loop is executed *n* times and we are able to move an invariant instruction out of the body and into the header (saving its result in a register for use within the body), then we will eliminate *n* -1 calculations from the program
    - a potentially significant savings
- In order to tell whether an instruction is invariant, we need to identify the bodies of loops, and we need to track the locations at which operand values are defined

Loop Improvement

- An *induction variable* (or register) is one that takes on a simple progression of values in successive iterations of a loop.
  - We confine our attention to arithmetic progressions
  - Induction variables appear as loop indices, subscript computations, or variables incremented or decremented explicitly within the body of the loop
- Induction variables are important for two reasons:
  - They commonly provide opportunities for strength reduction, replacing multiplication with addition
  - They are commonly redundant: instead of keeping several induction variables in registers, we can often keep a smaller number and calculate the remainder from those when needed

Compiling for Modern Processors

- ***pipelining*** is probably the most important performance critical feature
  - It works like this: TIME →

fetch inst	decode inst	fetch data	execute	store data		
	fetch inst	decode inst	fetch data	execute	store data	
		fetch inst	decode inst	fetch data	execute	store data
			fetch inst	decode inst	fetch data	execute

Compiling for Modern Processors

- The processor has to be careful not to execute an instruction that depends on a *previous* instruction that hasn't finished yet
  - The compiler can improve the achievable performance by generating code in which the number of dependencies that would *stall* the pipeline is minimized
- This is called ***instruction scheduling***; it is one of the most important optimizations for modern compilers

Compiling for Modern Processors

- Loads and load delays are influenced by
  - Dependences
    - Flow dependence (read after write)
    - Anti-dependence (write after read)
    - Output dependence (write after write)
- Branches (control dependencies)
  - since control can go both ways, branches create delays

Compiling for Modern Processors

- Goal for performance: *minimize pipeline stalls*
- Loads and branches take longer than *ordinary* instructions
- The instruction scheduler tries to find instructions that can executed during the delay
- Loads have to go to memory, which is slow
  - the instruction in a load delay slot can't use the loaded value
- Branches disrupt the pipeline
- A branch instruction generally takes 2 cycles to evaluate the branch decision
  - the instruction in a “branch delay slot” gets executed whether the branch occurs or not
  - A final “store” is a good candidate
  - Alternatively, an instruction can be run and nullified

Register Rotation

- Some modern processors feature rotating registers, which “rotate” or are renumbered with each iteration of a loop
- If the load instruction has a 4 cycle latency, then the store cannot begin until 4 cycles after the load begins

- Consider this pseudo-code

```
DO I = 1,N
    load X(I) into register 1
    store register into Y(I)
ENDDO
```

Register Rotation

- If the registers rotate, then that example can be implemented as:

```
load X(1) into register 5
load X(2) into register 4
load X(3) into register 3
load X(4) into register 2
DO I = 1, N-4
   load X(1+4) into register 1
   store register 5 to Y(i)
ENDDO
store register 4 to y(n-3)
store register 4 to y(n-2)
store register 4 to y(n-1)
store register 4 to y(n)
```

Predicate Registers

- Predicate registers are single-bit registers that allow the conditional execution of instructions
- Y = 2.0 -> (p1) Y = 2.0
  - If p1 is “1”, then the operation is performed, otherwise it is treated as a nop
- This allows some control dependencies to turn into data dependencies

Predicate and Rotating Registers

```
DO I = 1, N+4
If (I <= N) set p1=true; else p1=false;
If (I >= 4) set p2=true; else p2=false;
(p1) load X(I) into register 1
(p2) store register 5 to Y(I-4)
ENDDO
```

Conversion of if statements

- Replace conditional branches with predicated operations.
- For example, the code generated for:

```
if (a < b)
    c = a;
else
    c = b;
if (d < e)
    f = d;
else
    f = e;
```

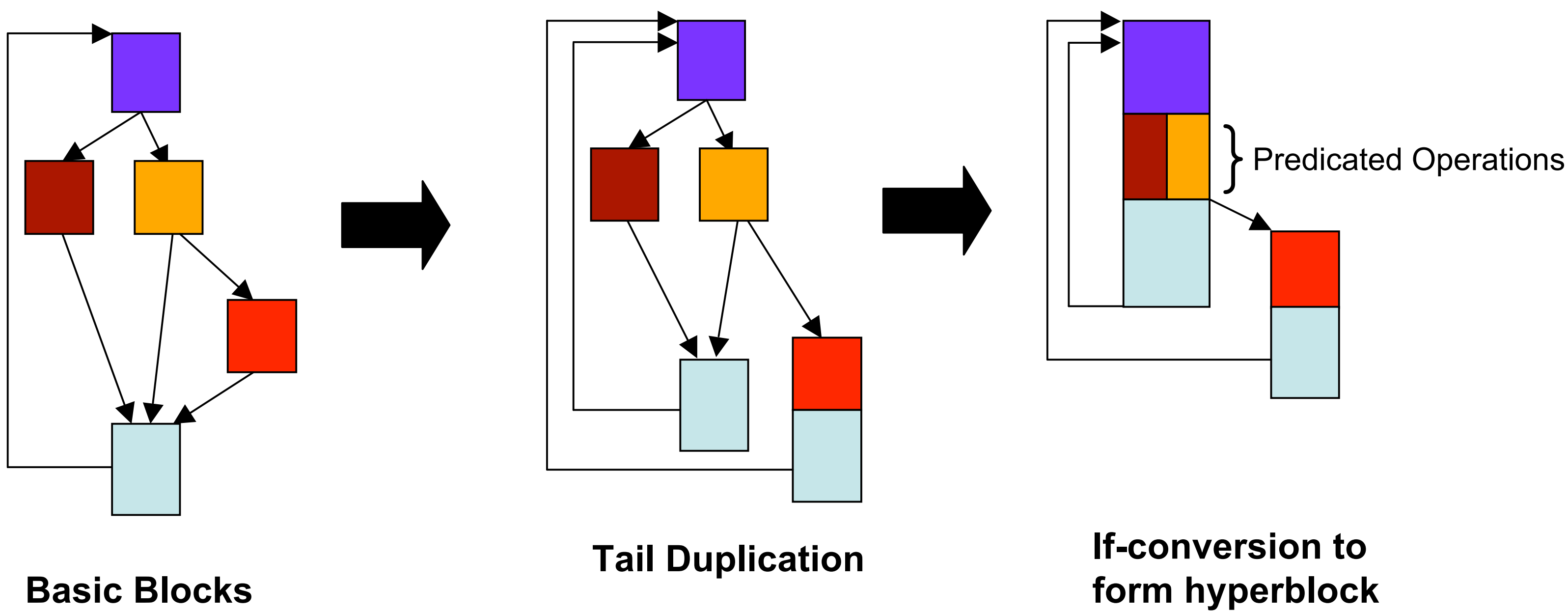
might be these two EPIC instructions:

`P1 = CMPP.< a,b`	`P2 = CMPP.>= a,b`	`P3 = CMPP.< d,e`	`P4 = CMPP.>= d,e`

`c = a   if p1`	`c = b   if p2`	`f = d   if p3`	`f = e   if p4`

Hyperblocks

- In hyperblock formation, if-conversion is used to form larger blocks of operations than the usual basic blocks
  – tail duplication used to remove some incoming edges in middle of block
  – if-conversion applied after tail duplication
  – larger blocks provide a greater opportunity for code motion to increase instruction-level parallelism



Loop Improvement II

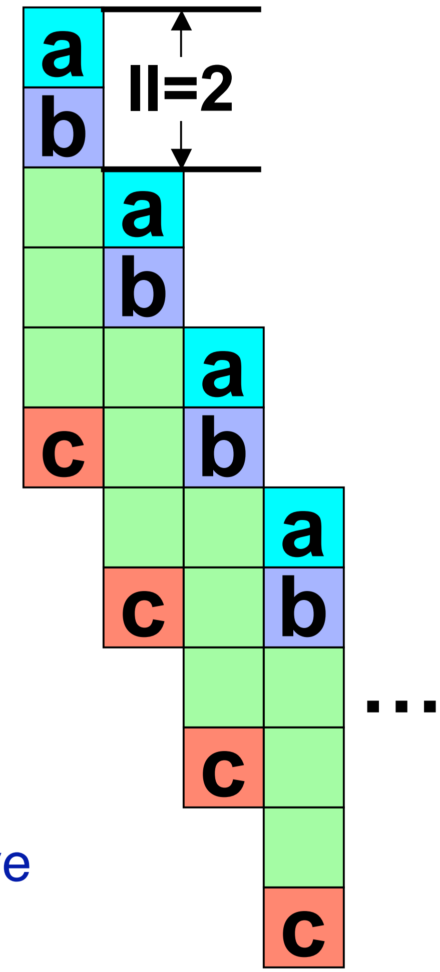
Loop Unrolling and Software Pipelining

- Loop *unrolling* is a transformation that embeds two or more iterations of a source-level loop in a single iteration of a new, longer loop, and allowing the instruction scheduler to intermingle the instructions of the original iterations
- Loop unrolling gives more instructions between branches (increases the size of the basic block)
  - This provides more opportunities for instruction scheduling improvements

Software Pipelining

Software pipelining is a loop scheduling technique that overlaps the execution of successive loop iterations.

```
for (i = 0; i < N; i++) {
    a: x ← y{1} + …;
    b: y ← …;
    c: … ← x;
}
```



Modulo scheduling: overlaps the execution of successive iterations in a fixed Initiation Interval (II).

Current Topics

- Inter-procedural optimization (IPO)
  - We've talked about optimization within basic blocks
  - With dataflow analysis, these sorts of optimizations can be extended to an entire function
  - Given that object files center around functions, there is traditionally no good time to perform whole-program or inter-procedural optimizations
  - One way to proceed is to carry the IR in the object file and look for optimization opportunities when objects are linked
  - Function inlining is a good example of what can be done
    - That can also expose more opportunities for instruction scheduling

Current Topics

- Profile-guided Optimization (PGO)
- We talked about branches from the perspective of basic block scheduling
- Another important topic is branch prediction
- To attempt to keep the pipeline filled, the compiler can predict whether a branch will be taken or not and continue to fetch instructions and data
- There is not much information available at compile time to make this an informed guess
- With PGO, the compiler can insert instrumentation to record branch behavior
- Subsequent profile-guided compilation can improve the predictions and thus, the performance

Current Topics

- Modern processors sometimes execute instructions out of order
  - Think of it as dynamic instruction scheduling
- Due to this, there are many cases where determining the best code and data layout is extremely difficult (if not intractable)
- There is growing tendency toward “empirical optimization”
  - Propelled by the Automatic Tuning of Linear Algebra Software (ATLAS) work by Whaley and Dongarra
- Simply allow the compiler (or compiler harness) to try variants of code structure and data layout and choose the one that works best

Current Topics

- ASPhALT - Automatic System for Parallel AppLication Transformation
  - Work in my group at U. Delaware
- Using the Open64 compiler infrastructure, we transform MPI codes to optimize communication performance
  - Source to source by un-parsing the WHIRL after transformation
- Goals:
  - Take advantage of data dependence in the compiler
  - Implement an "empirical optimization" harness

Sources

- Scott, Programming Language Pragmatics
- Kennedy, Allen, Optimizing Compilers for Modern Architectures
- Cooper, Torczon, Engineering a Compiler
- Muchnick, Advanced Compiler Design
- Text and slides from
  - Texts above
  - Amaral
    - http://www.cs.ualberta.ca/~amaral/
  - Rong and Gao
    - http://www.capsl.udel.edu
  - Goldberg, NYU
    - http://cs.nyu.edu/goldberg/