RTL-optimization in GCC

Alexander Nutz

Overview

- Background: Syntax Tree & Control Flow Graph
- What is RTL?
  - what is it used for?
  - basic structure
  - history
- RTL-optimizations
  - overview
  - explanation of some
- Examples

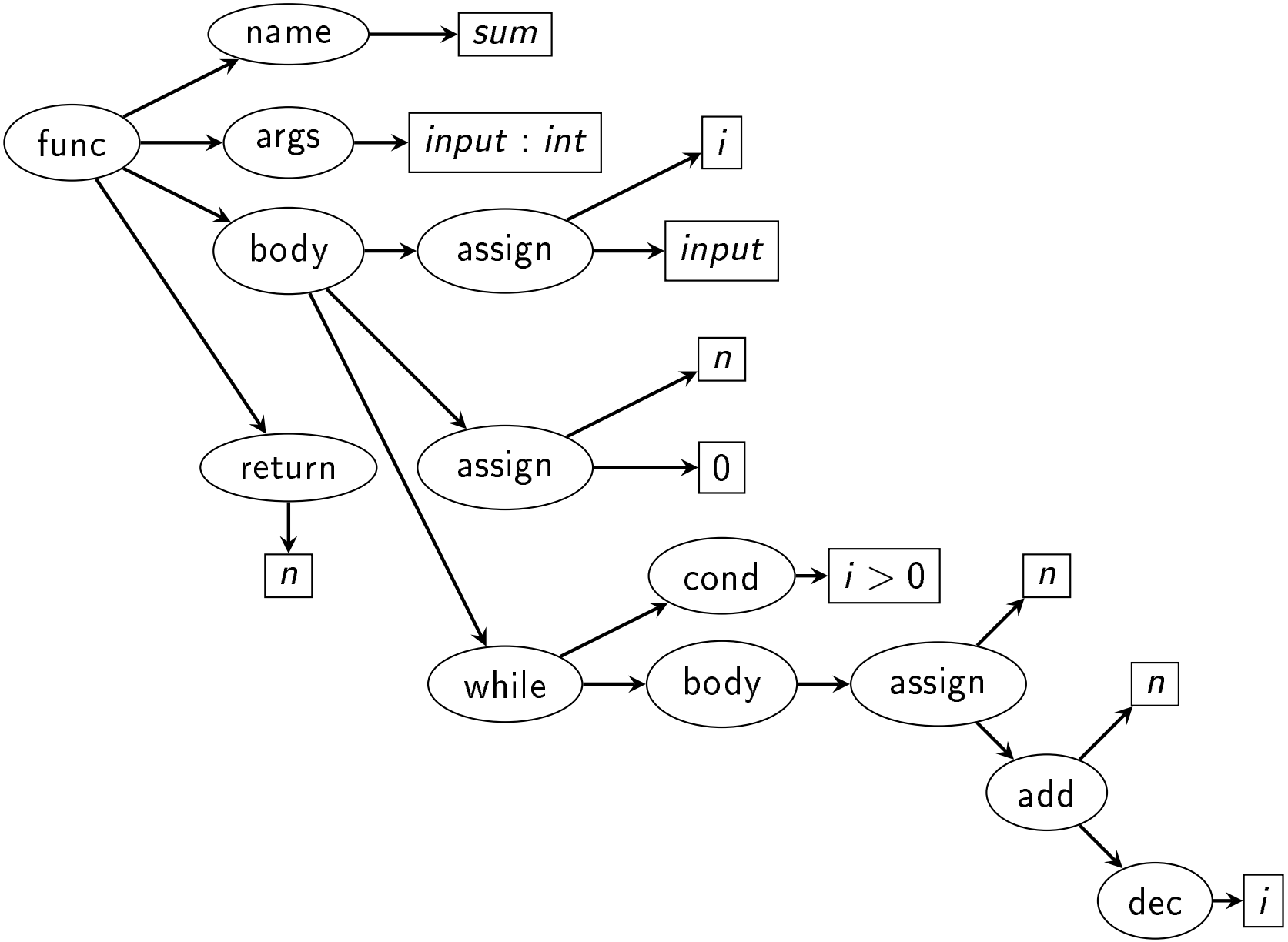
Background: Syntax Trees

- High-level programming languages: highly nested structure (parentheses, loops, complex expressions,...)
  - Parsing naturally gives tree-structured representation (derivation tree of grammar)
- Assembler: “flat”
  - Compliler has to flatten the tree

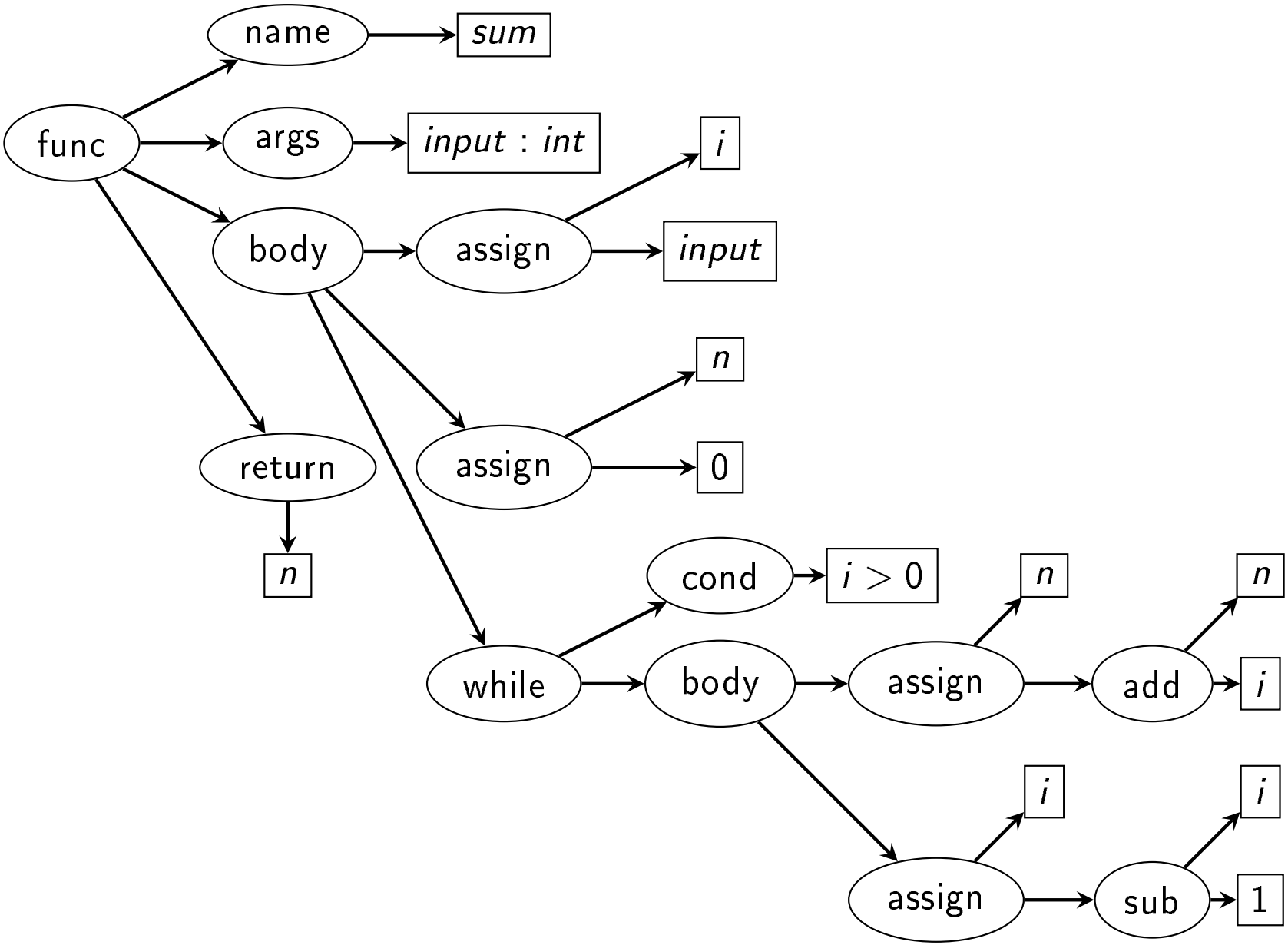
Background: Syntax Trees - Example Program

```
int sum(int input) {
  i = input;
  n = 0;
  while(i > 0)
    n = n + (i--);
  return n;
}
```

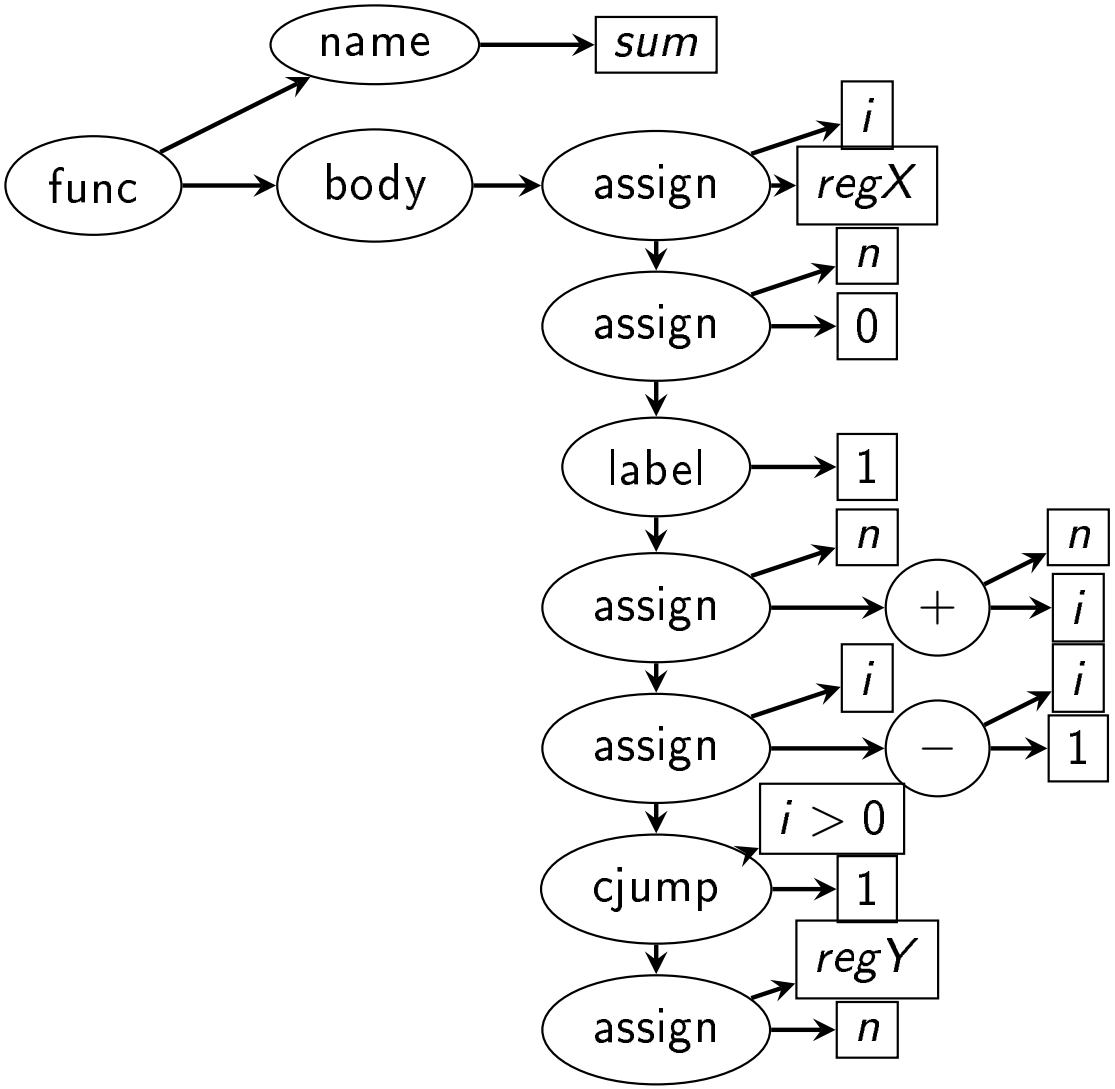
Abstract Syntax Tree - Example



AST-lowering Example (1)



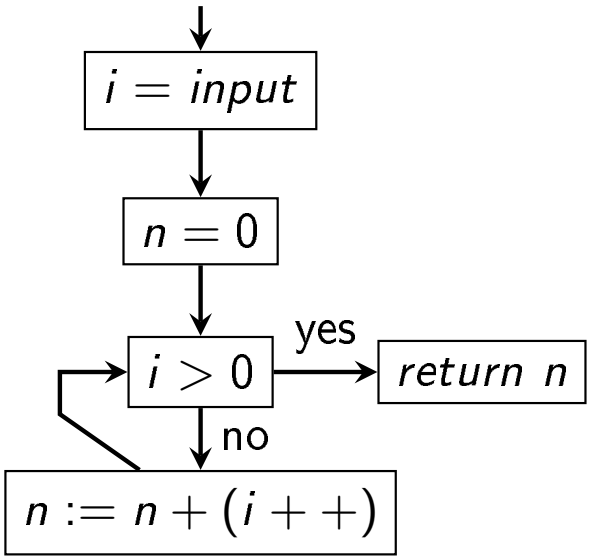
AST-lowering Example (2)



Control Flow Graph

Graph with

- Statements/Basic blocks as Nodes
- Edges according to control flow



ASTs and CFGs: Conclusion

- AST
  - focus on syntactic structure
  - transformations are made on it
  - generated from source code/other AST
- CFG
  - focus on control flow
  - many analyses are based on it (f.i. dataflow analyses)
  - generated from AST

Both are representations of the program (part) but different aspects are made explicit.

Both can have annotations containing additional high-level information.

RTL is...

- "Register Transfer Language"
- GCC's traditional intermediate representation
  - flat – sequence of instructions
  - LISP-like syntax
  - (lots of) additional information (like control flow, data dependencies,...)
  - can be close to hardware or "not-so-close" (f.i. handles pseudo-registers as well as hard registers)
- Also used for machine descriptions in GCC (not in scope here)

Basic RTL Syntax

The basic RTL units are called *insns* – roughly equivalent to statements/assembler lines

Example insns:

```
(insn 11 10 12 4 test.cpp:4
 (set (reg:SI 59 [ n ]) (const_int 0 [0x0]))
 -1 (nil))

(jump_insn 12 11 13 4 test.cpp:4
 (set (pc) (label_ref 24))
 -1 (nil))

Shape of insn, jump_insn, call_insn:
(<insn-type> <Id> <prevId> <nextId>
  <insn-code in machine description> <program locati
  <side effect pattern>
  <register dependencies> <misc. notes on regs>)
```

RTL past vs. today

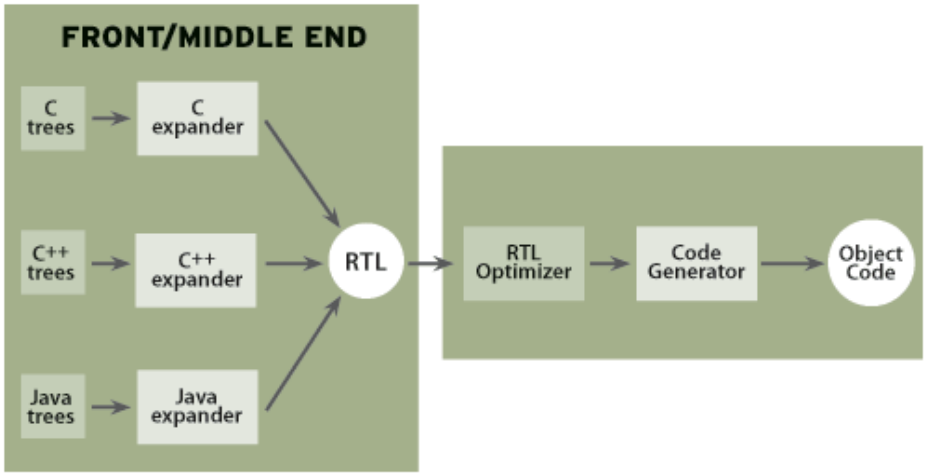


Figure: Old GCC architecture

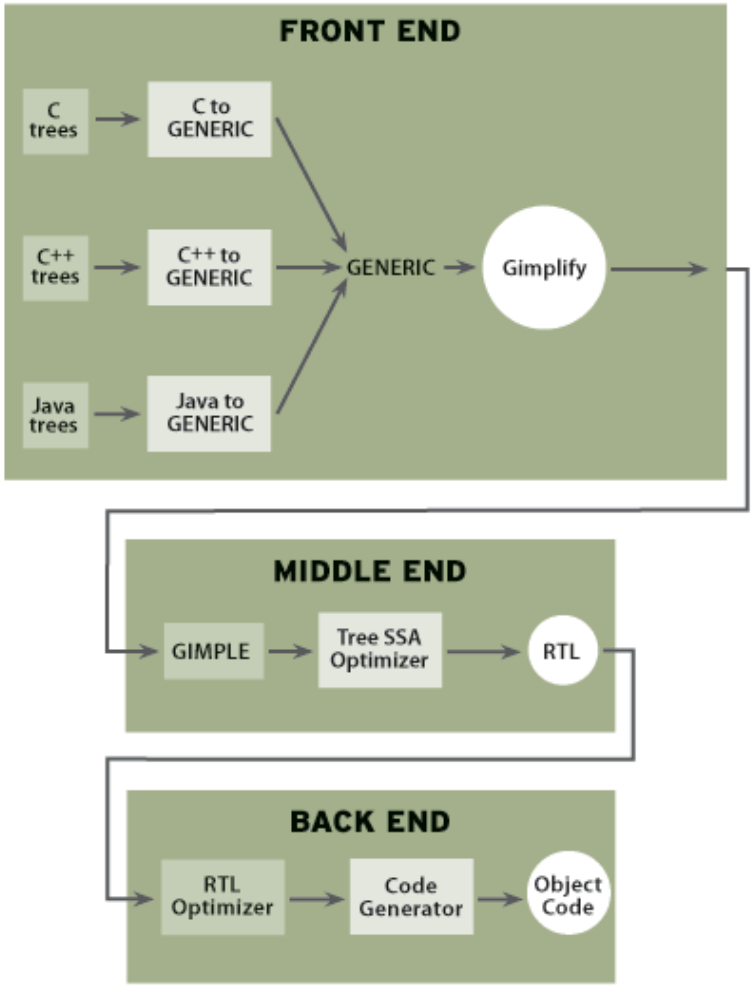


Figure: GCC 4.0 (April 2005): Integration of Tree-SSA

pictures taken from [1]

RTL vs. Tree-SSA

Tree-SSA	RTL
better for optimizations	
closer to programmer	closer to assembler
machine independent	possibly machine dependent
middle-end	middle-to-back-end

Machine independet optimizations are moved to Tree-SSA – still in progress.

RTL-passes in GCC

- Generation of exception landing pads
- CFG cleanup
- Forward propagation of single-def values
- Common subexpression elimination
- Global common subexpression elimination
- Loop optimization
- Jump bypassing
- If conversion
- Web construction
- Instruction combination
- Register movement
- Mode switching optimization
- Modulo scheduling
- Instruction scheduling
- Register allocation
- Basic block reordering
- Variable tracking
- Delayed branch scheduling
- Branch shortening
- Register-to-stack conversion

RTL-passes in GCC

- Generation of exception landing pads
- **CFG cleanup**
- **Forward propagation of single-def values**
- **Common subexpression elimination**
- **Global common subexpression elimination**
- **Loop optimization**
- **Jump bypassing**
- **If conversion**
- Web construction
- **Instruction combination**
- Register movement
- Mode switching optimization
- Modulo scheduling
- **Instruction scheduling**
- **Register allocation**
- Basic block reordering
- Variable tracking
- Delayed branch scheduling
- Branch shortening
- Register-to-stack conversion

(Global) Common Subexpression Elimination

1. Detect common subexpressions (dataflow analysis)
2. Compute cost of replacement
3. Replace if cheaper

Local:

- within basic blocks
- simple analysis

Global:

- whole procedure
- more complex analysis needed
- does *partial redundancy elimination*

Common Subexpression Elimination – Example

```
i = a + b * c
j = i + b * c
b = b + 4
k = b * c
```

is optimized to:

```
newvar = b * c
i = a + newvar
j = i + newvar
b = b + 4
k = b * c
```

Partial Redundancy Elimination – Example

```
if(b)
{
  x = 4
}
else
{
  x = 5
  i = x * y
}
j = x * y
```

is optimized to:

```
if(b) {
  x = 4
  newvar = x * y
}
else {
  x = 5
  newvar = x * y
  i = newvar
}
j = newvar
```

Loop Optimization

- Loop invariant motion
  - move statements that are not changed in the loop outside of it
- Loop unrolling
  - reduce Loop condition checking-overhead by copying the body
- Loop peeling
  - copy first or last few iterations to the outside of the loop
- Loop unswitching
  - replace if-then-else in loop-body by top-level if-then else and two copies of the loop

Loop Unrolling – Example

```
int x;
for(x=0;x<100;x++)
{
  remove(x);
}
```

is optimized to:

```
int x;
for (x=0;x<100;x+=5)
{
  remove(x);
  remove(x+1);
  remove(x+2);
  remove(x+3);
  remove(x+4);
}
```

Loop Peeling – Example

```
int p = 10;
for(int i=0;i<10;++i)
{
  y[i] = x[i] + x[p];
  p = i;
}
```

is optimized to:

```
y[0] = x[0] + x[10];
for(int i=1;i<10;++i)
{
  y[i] = x[i] + x[i-1];
}
```

Loop Unswitching – Example

```
int i, w;
int[] x[100], y[100];
for(i=0;i<100;i++) {
  x[i] = x[i] + y[i];
  if (w)
    y[i] = 0;
}
```

is optimized to:

```
int i, w, x[100], y[100];
if (w) {
  for (i=0;i<100;i++) {
    x[i] = x[i] + y[i];
    y[i] = 0;
  }
} else {
  for (i=0;i<100;i++) {
    x[i] = x[i] + y[i];
  }
}
```

Register Allocation

- Liveness-Analysis
  $\Longrightarrow$ a variable is dead at a program point if is not read until the program stops, live otherwise
- Interference/preference graph
- in GCC/IRA

Register Allocation - Example

```
a = 1
b = 2
c = 3
d = 4
e = a + b
f = e + c
g = f + d
r = g
return r
```

Register Allocation - Example - Result

```
reg1 = 1
reg2 = 2
store memLoc1 3
store memLoc2 4
reg1 = reg1 + reg2
reg1 += load memLoc1
reg1 += load memLoc2
return reg1
```

Register Allocation (3)

Things are further complicated by:

- different kinds of registers
- optimizing across regions/interprocedural optimizations
- different costs for register operations/spilling
- coalescing may increase outdegree

- *From Source to Binary: The Inner Workings of GCC*: http://www.redhat.com/magazine/002dec04/features/gcc/
- Wikipedia (english) articles on: *Loop Unrolling, Loop Peeling, Loop Unswitching, Register Allocation, Partial Redundancy Elimination, Common Subexpression Elimination, GNU Compiler Collection*
- *GCC Internals – RTL*: http://gcc.gnu.org/onlinedocs/gccint/RTL.html
- *GCC Internals – RTL-passes*: http://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html
- *GCC Internals – Debugging Options*: http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
- *GCC Internals – Optimize Options*: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- Vladimir N. Makarov, *The Integrated Register Allocator for GCC*, in: Proceedings of the GCC Developers' Summit 2007