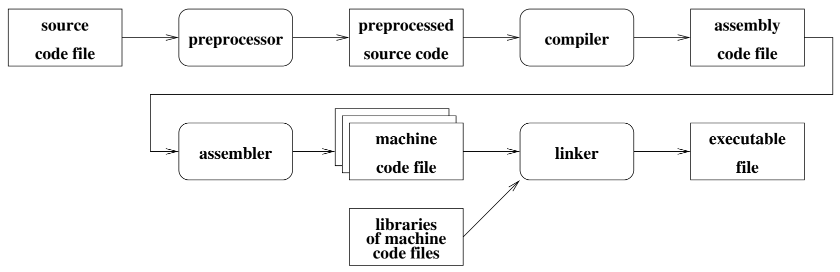
Compilation Process

- There are several steps in the compilation process.



Number of Compilers Needed

- Affected by the number of high-level source languages in which programmers write.
- Affected by the number of machine architectures (different assembly or machine languages) on which programs will execute.

Compiler Phases

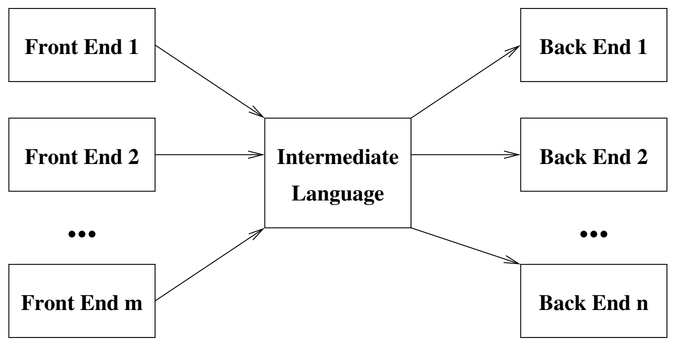
- scanner
- parser
- semantic analyzer
- intermediate code generator
- optimizer
- code generator
- peephole optimizer

Front Ends and Back Ends

- front end
  - takes high-level source code as input
  - produces intermediate code as output
- back end
  - takes intermediate code as input
  - produces assembly or machine code as output

Reducing the Number of Compilers

- With *m* source languages and *n* machines, we need only *m* front ends and *n* back ends (*m*+*n*), as opposed to *m***n* distinct compilers.



Levels for Code Optimization

- statement - high-level optimizations
- basic block - low-level optimiztions (local)
- innermost loop - ex: software pipelining
- perfect loop nest - ex: loop interchange
- general loop nest - ex: loop invariant code motion
- function - applied to entire function (global)
- interprocedural - applied across function boundaries

Optimization Is a Misnomer

- An optimizing compiler typically performs a number of compiler optimizations.
- An optimizing compiler cannot guarantee that it will produce optimal code.
- A *code-improving transformation* is a much more accurate description than a *compiler optimization*.

Phases and Transformations

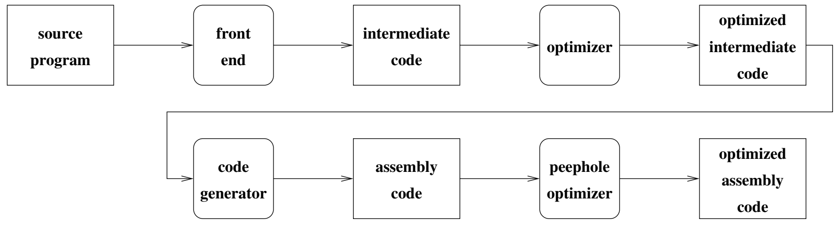
- A compiler phase indicates that the compiler is attempting to apply a number of transformations of the same type.
- A compiler transformation consists of a number of changes.
- Two general types of compiler transformations.
  - improving transformation - program represenation is semantically equivalent before and after the transformation
  - necessary transformation - a sequence of changes that are required to produce correct code for a particular target machine

Code Improving Goals

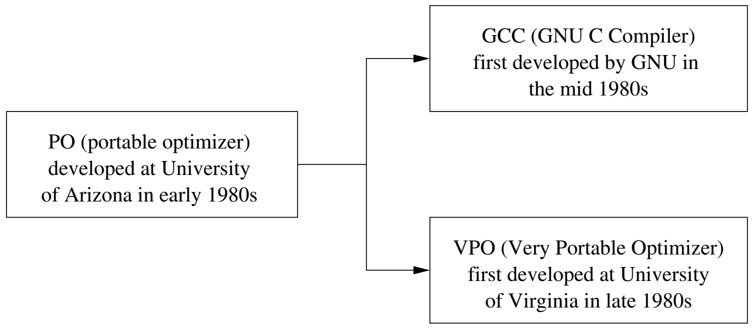
- improve execution time
  - decrease number of instructions executed
  - decrease number of references to memory
  - better exploitation of the architecture
    - pipeline
    - memory hierarchy
    - parallelism (instruction, loop, task)
- decrease size
  - code
  - data
- reduce energy usage

Traditional Approach to Code Generation

- Most compiler optimizations are traditionally performed in a machine-independent fashion.



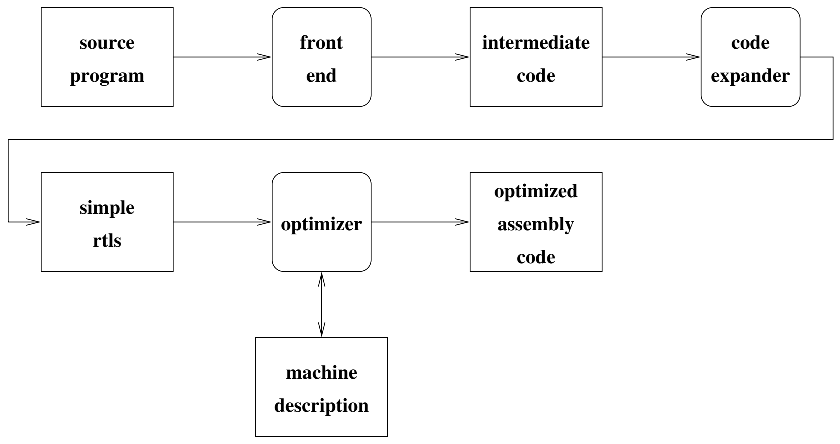
VPO (Very Portable Optimizer)


PO (portable optimizer) developed at University of Arizona in early 1980s
GCC (GNU C Compiler) first developed by GNU in the mid 1980s
VPO (Very Portable Optimizer) first developed at University of Virginia in late 1980s

General VPO Strategies

- Uses a very simple front end that performs no optimizations.
- Uses a very simple code generator to translate each intermediate operation into a sequence of target instructions. The code generator also performs no optimizations.
- Does all optimizations after code generation at the machine level.
- Maintains a consistent representation of the code so that phase ordering problems can be avoided.

Optimizations after Code Generation



Why Perform a Naive Translation of the Source Code?

- A naive front end is easier to write and debug.
- Few optimizations are as effective when applied to source or intermediate-language code as they are at the machine level.

VPCC (Very Portable C Compiler)

- VPCC was the original C front end for VPO.
- Accepts the complete K&R C language.
- Entire front end is a little over 9000 lines of code.
- Produces an intermediate language:
  - postfix (stack code) that is easy to translate to different architectures
  - 46 different simple operations

Example Translation of a C Statement to VPCC Intermediate Code

```
a = a + 1;
```

$\Rightarrow$

```
Name a local
Deref
Con 1
Plus
Name a local
Assign
```

Register Transfer Lists

- RTLs (also used in CHDLs, Computer Hardware Description Languages), are machine-independent representations of machine-dependent operations.
- An RTL describes the effects of a machine instruction:

```
r[2]=r[4];        # move
r[5]=r[6]+r[7];   # add
M[r[2]+4]=r[3];   # store
IC=r[2]?r[3];     # compare
```

Register Transfer Lists (cont.)

- RTLs (like instructions) can have multiple effects, which are treated as being accomplished in parallel.
- An RTL describes the effects of a machine instruction:

```
r[6]=r[6]+r[7]; IC=r[6]+r[7]?0;
```

- Transfers of control update the program counter.

```
PC=L12;        # unconditional
PC=IC<0,L12;   # conditional
```

- Complex instructions can be represented using a function call notation.

```
d[2]=sqrt(d[3]);
```

General Form of an RTL

- RTL - Register Transfer List
- {dst = src;}+
- dst
    - register (general-purpose or special)
    - memory reference
- src
    - any expression that is legal for the machine

Why Use RTLs As a Notation for Instruction Representation?

- Since the general RTL form is machine independent, the algorithms that manipulate RTLs are also machine independent.
- RTLs allow optimizations to be performed at the machine level since they represent machine specific instructions.
- Because RTLs are well-defined, it is possible to construct recognizers that can determine if an RTL represents a legal instruction on a target machine.

Why Use RTLs As a Notation for Instruction Representation? (cont.)

- The RTL notation is flexible enough to represent instructions in all optimization phases.
- The notation is easier to understand than other forms:
  - trees
  - stacks
- Easy to understand the effect of an optimization since each RTL represents an instruction on the machine.

Code Expander

- Accepts a sequence of intermediate language operations and translates them into a corresponding sequence of RTLs.
- Code expanders are machine-dependent. Each RTL they emit must denote a valid machine instruction on the target machine.
- Code expanders are dependent on the source language since they must establish the calling conventions of that language.
- With m front ends and n back ends, there would be $m*n$ code expanders.

Pseudo Registers

- The code expander does not assign temporaries to hardware registers.
- Pseudo registers are used instead. They are the same form as hardware registers, except the numbers are higher.
- Any time a new value is needed to be assigned to a unique temporary, the next higher pseudo register is used.

Limiting the Number of Pseudo Registers

- VPO limits the number of pseudo registers at compiler installation time.
- A pseudo register is never live across C statements. A counter is reset at each new statement to allow us to start reusing pseudo registers.
- A pseudo register could be live across a basic block as some C statements contain more than one basic block.

```
a = b <= c;
a = b ? c : d;
```

Dead Registers

- Associated with RTLs are dead register lists.
- A dead register indicates that the RTL contains the last use of the value in this register.
- Example of a dead register:

```
r[17]=r[17]+r[16];  r[16]:
```

Dead Registers (cont.)

- The writer of the code expander explicitly indicates the points where register values become dead. This information could be calculated, but it saves compilation time to have the code expander indicate it.
- One way the dead register information is used is to indicate to the register assignment phase that the hardware register bound to a pseudo register is now free to be bound to another pseudo register.

Calling Conventions

- The code expander also represents the calling conventions for function calls.
  - passing arguments in registers and/or on the run-time stack
  - mechanism to return the function result
  - which registers should have their values preserved by called functions

Example of a Code Expander Routine

```
void cex_plus(int typeid)
{
   int x, y;

   x = *--sp; // gets reg values from CGS
   y = *(sp-1);
   printf("+%c%c=%c%c+%c%c\t%c%c\n",
          RT(y),RT(y),RT(x),RT(x));
}
```

Example of a Code Expander RTLs

- a=a+1; translated into expanded RTLs

```
r[32]=r[30]+1_a;
r[33]=R[r[32]];     r[32]:
r[34]=1;
r[33]=r[33]+r[34]; r[34]:
r[35]=r[30]+1_a;
R[r[35]]=r[33];     r[33]:r[35]:
```

Another Example of a Code Expander RTLs

- There is not always a 1 to 1 mapping between intermediate operations and RTLs.
- For example, global addresses have to be constructed on the SPARC.

```
   Name a global
=>
   r[32]=HI[_a];
   r[32]=r[32]+LO[_a];
```

Types of Code Expander Records

char	value	description
'+'	RTLINE	register transfer list
'-'	ASMLINE	line passed to the assembler
'_'	DELAYLINE	line passed to assembler after function
't'	TRASHLINE	tells CSE phase what values are no longer available
'u'	USELINE	a hardware register has been implicitly used
'r'	RESLINE	a hardware register has been implicitly assigned a value
's'	SIDEFFECT	also represents implicit uses
'c'	CASELINE	used to build a table of addresses for a C switch stmt
'f'	FUNCNAME	name of the function
'#'	COMMENT	comment
'S'	SETLINE	indicates that a list of registers could be set at this point
'C'	CSLINE	indicates callersave registers at point of call
'P'	PARMLINE	contains list of register arguments read by callee
'm'		maps register types to characters
'n'		name of the source file
'd'		declaration of a variable
'L'		label for a basic block
'*'		function separator
'l'		source file line number

Necessary Transformations in VPO

- register assignment
  - Assigns pseudo registers (temporary values) to hardware registers.
- fixing function entry and exit
  - Adds code at the function prologue and epilogue to allocate/deallocate space on the runtime stack for the activation record, save and restore callee-save registers, etc.

Optimization Phases in VPO

- branch chaining
- useless jump elimination
- unreachable code elimination
- basic block reordering
- instruction selection (local and global)
- constant folding
- evaluation order determination
- branch reversal
- dead assignment elimination
- loop inversion
- cross jumping

Optimization Phases in VPO (cont.)

- register allocation
- common subexpression elimination
- loop-invariant code motion
- recurrence elimination
- loop strength reduction
- induction variable elimination
- strength reduction
- loop unrolling
- instruction scheduling
- filling delay slots

VPO Optimizations That Typically Decrease Code Size and Execution Time

- branch chaining
- useless jump elimination
- basic block reordering
- instruction selection
- branch reversal
- dead assignment elimination
- common subexpression elimination
- induction variable elimination
- filling delay slots

VPO Optimizations That Typically Just Decrease Execution Time

- loop-invariant code motion
- strength reduction
- instruction scheduling

VPO Optimizations That Decrease the Number of Memory References

- register allocation
- recurrence elimination

VPO Optimizations That Typically Just Decrease Code Size

- unreachable code elimination
- cross jumping

VPO Optimizations That Typically Just Enable Other Optimizations

- constant folding
- evaluation order determination
- loop strength reduction

VPO Optimizations That Decrease Execution Time and Increase Code Size

- loop inversion
- loop unrolling

Basic Blocks

- A basic block is a sequence of instructions (operations or statements) having only one entry point and one exit point.
- Basic blocks are determined by encountering:
  - labels
  - transfers of control
    - unconditional jumps (direct and indirect)
    - conditional branches
    - calls (direct and indirect)

Calculating Basic Blocks

- Determine the set of leaders (first instruction in each basic block).
  - the first instruction in a function
  - any instruction that is the target of a transfer of control
  - any instruction immediately following a transfer of control
- For each leader, its basic block consists of the leader and the statements up to but not including the next leader or the end of the function.

Phase Ordering Problem

- typical ordering of optimization phases attempted
  - statically determined
  - order cannot be easily changed (different representations)
- The problem is that there are situations where one phase creates new opportunities for optimizations already attempted by a previously executed phase.

VPO Approach to the Phase Ordering Problem

- VPO uses an iterative approach
  - maintains RTL representation through all optimization phases
  - allows previous phases to be reinvoked when new opportunities for optimizations are introduced

Order of Optimizations in VPO

branch chaining
useless jump elimination
unreachable code elimination
branch reversal (eliminates unconditional jumps)
basic block reordering (eliminates unconditional jumps)
merge basic blocks
local instruction selection
evaluation order determination
global instruction selection
register assignment
instruction selection
loop inversion

Order of Optimizations in VPO (cont.)

```
do {
    dead assignment elimination
    register allocation
    instruction selection
    common subexpression elimination
    dead assignment elimination
    for each loop, innermost first do
        cross jumping
        loop-invariant code motion
        recurrence elimination
        loop strength reduction
        induction variable elimination
    various branch optimizations seen before
    strength reduction
    instruction selection
} while (changes);
```

Order of Optimizations in VPO (cont.)

various branch optimizations seen before
fixing function entry and exit
instruction scheduling
filling delay slots
useless jump elimination
branch chaining

Why Reapply Code-Improving Transformations?

- A small change may introduce the potential for new code-improving transformations to be applied.
- Instruction selection is commonly reinvoked after other code-improving transformations since a change in the form of an instruction may allow it to be combined with other instructions.

Instruction Selection after CSE

```
r[1]=F[r[13]+i.];
...
r[2]=F[r[13]+i.];
r[3]=r[3]+r[2];   r[2]:
=>
r[1]=F[r[13]+i.];
...
r[2]=r[1];
r[3]=r[3]+r[2];   r[2]:
=>
r[1]=F[r[13]+i.];
...
r[3]=r[3]+r[1];
```

Instruction Selection after Register Allocation

```
d[3]=R[a[6]+i.];
a[2]=_down;
R[d[3]<<2+a[2]]=0; d[3]:
=>
d[3]=d[4];
a[2]=_down;
R[d[3]<<2+a[2]]=0; d[3]:
=>
a[2]=_down;
R[d[4]<<2+a[2]]=0;
```

Control Flow

- Why does an optimizer need to know the flow of control between basic blocks?
  - eliminate unreachable code
  - perform various branch optimizations
  - calculate data-flow information
  - locating loops

Branch Chaining

- If a basic block contains a single instruction that is an unconditional jump, then each block that is a predecessor that jumps or branches to this block can jump instead to the target of the unconditional jump.
- L19 in the conditional branch below can be replaced by L20:

```
    PC=IC<0,L19;
    ...
L19:
    PC=L20;
```

Useless Jump Elimination

- If a block ends with a jump (conditional or unconditional) and the target of the jump is the block that positionally follows the jump, then the jump can be deleted.

```
        ...
        PC=L5;
    L5:
        ...
```

Unreachable Code Elimination

- Unreachable code elimination eliminates basic blocks that can never be reached from the entry point of the function.

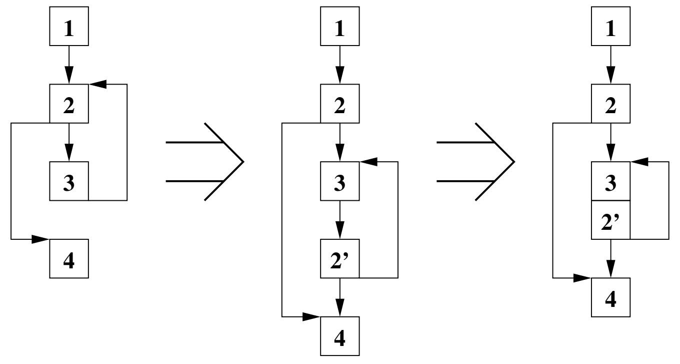
```
        PC=L2;
        r[1]=R[r[13]+i.];
        ...
    L3:
=>
        PC=L2;
    L3:
```

Merging Basic Blocks

- If a basic block does not end with a transfer of control, has only one successor, and its successor has only this block as a predecessor, then the two blocks can be merged.
- Why merge blocks?
  - Allows some global code-improving transformations to be applied as local transformations.
  - Reduces the number of basic blocks, which will reduce the time and space required for subsequent data and control-flow analysis.

Merging Basic Blocks Example

- The opportunity to merge basic blocks can often occur after other code-improving transformations, such as loop inversion.

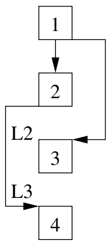


Basic Block Reorganization to Eliminate Unconditional Jumps

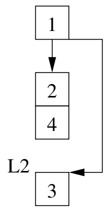
- If a basic block ends with an unconditional jump and the target block is not reached by falling into it from another block, then the unconditional jump can be eliminated by moving the target block and its positional successors to follow the block with the unconditional jump.

Example of Basic Block Reorganization from an If-Then-Else

```
    PC=IC!=0,L2;
    R[r[14]+i.]=0;
    PC=L3;
L2:
    PC=RT;
L3:
    ...
=>
    PC=IC!=0,L2;
    R[r[14]+i.]=0;
    ...
L2:
    PC=RT;
```



=>



Switch Statement Example

- Consider the code that would be generated for a C switch statement.

```
switch (expr) {
   case e1: ...
   case e2: ...
   case e3: ...
   ...
   default: ...
}
```

Code Generated for a Switch Statement

```
      code to evaluate expr;
      jump to test;
L1:   code for statements at e1
L2:   code for statements at e2
L3:   code for statements at e3
      ...
LD:   code for default case
      goto next;
test: linear search, binary search, or indirect jump
next: ...
```

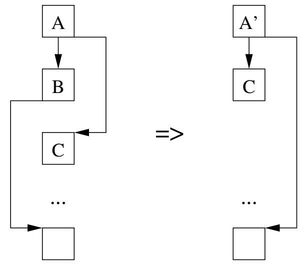
Jump Elimination by Reversing Conditional Branches

- An unconditional jump can be eliminated when the following conditions hold.
  - Block A contains a conditional branch.
  - Block B contains only an unconditional jump and positionally follows block A.
  - Block C is target of branch in block A and positionally follows block B.
  - There are no jumps to block B.

Jump Elimination by Reversing Conditional Branches

- Can eliminate the unconditional jump in block B by:
  - reversing the condition in branch in block A,
  - replacing the target of the branch in block A with the target of the unconditional jump in block B, and
  - deleting block B.

```
      PC=IC<=0,L1;
      PC=L2;
   L1:
=>
      PC=IC>0,L2;
   L1:
```



Why Do Jump Elimination in the Back End of a Compiler

- May be able to exploit opportunities after other code-improving transformations have been applied.
- The range of conditional branches on some machines is very limited.

Instruction Selection

- Instruction selection in VPO involves combining two or three instructions together into a single instruction that is valid for that machine.
- A recognizer developed from a machine description verifies if the merged effects of the combined instructions is valid.

```
   r[17]=5;
   r[16]=r[16]+r[17]; r[17]:
=>
   r[17]=5; r[16]=r[16]+5; r[17]:
=>
   r[16]=r[16]+5;
```

Instruction Selection (cont.)

- Instruction selection was often accomplished as a peephole optimization. Pairs or triples of instructions are replaced by a single instruction according to a predefined template of patterns.
- Unlike most peephole optimizers, VPO allows instructions to be combined that are not contiguous.

Links for Instruction Selection

- VPO uses data-flow analysis information to define links based on where a register is set and is first used.

```
   1       r[17]=5;
           r[16]=R[_base];
   3 {1,2} r[16]=r[16]+r[17]; r[17]:
=>
   2       r[16]=R[_base];
   3 {2}   r[16]=r[16]+5;
```

Instruction Simplification

- An instruction is simplified before checking the machine description to see if it is legal.
- constant folding

```
   r[16]=r[16]+5;
   r[17]=r[16]+6;    r[16]:
=>
   r[17]=r[16]+5+6; r[16]:
=>
   r[17]=r[16]+11;  r[16]:
```

Instruction Simplification (cont.)

- algebraic simplification

```
r[5]=r[5]+0; => r[5]=r[5];
r[5]=r[5]-0; => r[5]=r[5];
r[5]=r[5]*1; => r[5]=r[5];
r[5]=r[5]/1; => r[5]=r[5];
```

- multiple unary operators

```
r[18]=~~r[15]; => r[18]=r[15];
r[18]=--r[15]; => r[18]=r[15];
```