Improving Switch Lowering for The LLVM Compiler System

Anton Korobeynikov

asl@math.spbu.ru

Faculty of Mathematics and Mechanics,
Saint Petersburg State University

SYRCoSE 2007

Outline

1. The LLVM Compiler Infrastructure
   - Compiler Infrastructure
   - Targets
   - Link-Time Optimization
   - JIT
   - Target Independent Code Generator
2. Switch Lowering
   - Intro
   - Switches in LLVM
   - Real-World Examples
   - Methods To Emit a Switch
   - Glueing Things Together
3. Results

Outline

1. The LLVM Compiler Infrastructure
   - Compiler Infrastructure
   - Targets
   - Link-Time Optimization
   - JIT
   - Target Independent Code Generator
2. Switch Lowering
   - Intro
   - Switches in LLVM
   - Real-World Examples
   - Methods To Emit a Switch
   - Glueing Things Together
3. Results

Outline

1. The LLVM Compiler Infrastructure
    - Compiler Infrastructure
    - Targets
    - Link-Time Optimization
    - JIT
    - Target Independent Code Generator
2. Switch Lowering
    - Intro
    - Switches in LLVM
    - Real-World Examples
    - Methods To Emit a Switch
    - Glueing Things Together
3. Results

Compiler Infrastructure

- Provides modular and usable components for building compilers of any form.
- Reduces the time and cost to construct a particular compiler.
- Allows components to be shared across different compilers.
- Allows choice of the right component for the job.

LLVM Compiler Infrastructure

- A well-defined Intermediate Representation (IR).
  - Language independent, target independent, easy to use.
- Many high-quality components (libraries) with clean interface.
  - Optimizations, analyses, modular target-independent code generator, JIT compiler, accurate GC, link time optimization, X86/PPC/ARM/IA64/SPARC/Alpha code generators.
- Tools built from the libraries.
  - Aggressive optimizing C/C++/ObjC/Ada compiler.
  - Automated compiler debugger.
  - Modular optimizer.
  - LLVM JIT.

LLVM Compiler Infrastructure

- A well-defined Intermediate Representation (IR).
  - Language independent, target independent, easy to use.
- Many high-quality components (libraries) with clean interface.
  - Optimizations, analyses, modular target-independent code generator, JIT compiler, accurate GC, link time optimization, X86/PPC/ARM/IA64/SPARC/Alpha code generators.
- Tools built from the libraries.
  - Aggressive optimizing C/C++/ObjC/Ada compiler.
  - Automated compiler debugger.
  - Modular optimizer.
  - LLVM JIT.

LLVM Compiler Infrastructure

- A well-defined Intermediate Representation (IR).
  - Language independent, target independent, easy to use.
- Many high-quality components (libraries) with clean interface.
  - Optimizations, analyses, modular target-independent code generator, JIT compiler, accurate GC, link time optimization, X86/PPC/ARM/IA64/SPARC/Alpha code generators.
- Tools built from the libraries.
  - Aggressive optimizing C/C++/ObjC/Ada compiler.
  - Automated compiler debugger.
  - Modular optimizer.
  - LLVM JIT.

LLVM IR

- IR is a typed Virtual Instruction Set.
  - Operations are low-level instructions.
  - Language- & target- independent semantics.
- IR has three representations.
  - In-memory IR - for the compiler to work on.
  - On-disk binary IR - interchange format.
  - On-disk test IR - compiler debugging, inspection.
- IR has a clean & simple design.
  - Small memory footprint, fast to manipulate.
  - Easy to understand, well specified & documented.

LLVM IR Features

- Basic features
  - Lightweight design, efficient and easy to understand.
  - Scalar values are always in SSA form, memory never is.
  - IR is fully typed and types are rigorously checked for consistency.
  - Explicit array/struct accesses, supports alias/dependence analysis.
  - Full support for vector/SIMD datatypes and operations.
  - Full support for GCC-style inline assembly.
- Minor features
  - Exceptions are explicit in CFG, not at on-the-side datastructure.
  - IR is easily extensible with intrinsic functions (debug information and EH information were added this way!).
  - Supports custom calling conventions.

Targets/Platforms Currently Supported

- Stable
  - X86-32 (ELF, Windows, Darwin), X86-64 (Darwin, ELF)
  - PowerPC-32/64 (Darwin; Linux support is still experimental)
  - ARM (Darwin, ELF)
  - Sparc (V8, V9)
  - Alpha
- Experimental
  - IA64
  - MIPS (in progress, come soon!)
  - Sony/IBM Cell (in progress, come really soon!)
- Other
  - C
  - MSIL

GCC Integration (llvm-gcc4)

- Based on Apple's gcc 4.0.1 branch.
- Uses GCC front-end with LLVM optimizer and code generator
  - Reuses parser, runtime libraries, and some lowering bits.
  - Requires a new GCC "tree-to-llvm" converter.
- Aggressive and fast optimizer built on modern techniques
  - SSA-based optimizer for fast and aggressive transforms.
  - Aggressive loop optimization: unrolling, unswitching, ...
  - Many InterProcedural (cross-function) optimizations: inlining, dead argument elimination, IP constant propagation, global variable optimization, ...

Compilation Example

Original C Source

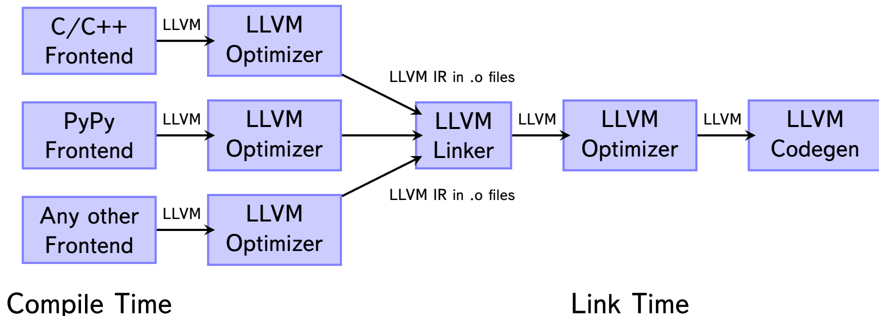
```
unsigned gcd(unsigned a, unsigned b) {
  return (b ? gcd(b, a % b) : a);
}
```

Optimized LLVM IR

```
define i32 @gcd(i32 %a, i32 %b) {
entry:
  %bz  = icmp eq i32 %b, 0
  br i1 %bz, label %return, label %loop
loop:
  %b1  = phi i32 [ %c,  %loop ], [ %b, %entry ]
  %a1  = phi i32 [ %b1, %loop ], [ %a, %entry ]
  %c   = urem i32 %a1, %b1
  %cz  = icmp eq i32 %c, 0
  br i1 %cz, label %return, label %loop
return:
  %res = phi i32 [ %a, %entry ], [ %b1, %loop ]
  ret i32 %res
}
```

Link-Time Optimization (LTO)

- Link-time is a natural place for interprocedural optimizations.
- LLVM is safe with partial programs (dynamically loaded code, libraries, etc).



Just-In Time Compilation

- LLVM IR can be executed directly:
  - Via native execution engine on X86 and PowerPC.
  - Via interpreter everywhere LLVM is running on.
- Execution engine compiles functions lazily.
- External interfaces: one can call external (native) functions and obtain the address of function being JIT'ed.

Existing projects using JIT:

- Present
  - Apple's OpenGL JIT
  - Aldrin (modular music sequencer/tracker)
- Future
  - LLVM-QEMU (GSoC 2007 project)
  - Mesa

Target Independent Code Generator

1. Abstract target description interfaces which capture important properties about various aspects of the machine.
2. Classes used to represent the machine code being generated for a target.
3. Target-independent algorithms used to implement various phases of native code generation (register allocation, scheduling, stack frame representation, etc). Switch lowering is also implemented here
4. Implementations of the abstract target description interfaces for particular targets.
5. The target-independent JIT components.

Switch Lowering: Intro

- Switches are very common to see:
  - Compilers
  - Virtual Machines
  - Parsers
- Switches are often used in the tight inner loops.

Thus:

- We have to lower switches in optimal way.
- Usual tradeof is size/speed, but often small size implies high speed.
- If we can use target-dependent "tricks", we must use them.

Switches in LLVM

- Switches are represented in LLVM with the help of **switch** instruction:

Syntax of switch instruction

```
switch <intty> <value>, label <defaultdest> [
  <intty> <val>, label <dest> ...
]
```

Emulate a conditional branch

```
%Val = zext i1 %value to i32
switch i32 %Val, label %tdest [
  i32 0, label %fdest
]
```

Implement a jump table

```
switch i32 %val, label %def [
  i32 0, label %onzero
  i32 1, label %onone
  i32 2, label %ontwo
]
```

Switches in LLVM

- Switches are represented in LLVM with the help of **switch** instruction:

Syntax of switch instruction

```
switch <intty> <value>, label <defaultdest> [
  <intty> <val>, label <dest> ...
]
```

Emulate a conditional branch

```
%Val = zext i1 %value to i32
switch i32 %Val, label %tdest [
  i32 0, label %fdest
]
```

Implement a jump table

```
switch i32 %val, label %def [
  i32 0, label %onzero
  i32 1, label %onone
  i32 2, label %ontwo
]
```

Switches in LLVM

Consider this switch

```
switch i32 %val, label %otherwise [
  i32 -1, label %dest1
  i32  0, label %dest1
  i32  1, label %dest1
  i32  2, label %dest1
  i32  3, label %dest2
]
```

Compact it!

```
switch i32 %val, label %otherwise [
  i32 -1..2, label %dest1
  i32     3, label %dest2
]
```

Switches in LLVM

Consider this switch

```
switch i32 %val, label %otherwise [
  i32 -1, label %dest1
  i32  0, label %dest1
  i32  1, label %dest1
  i32  2, label %dest1
  i32  3, label %dest2
]
```

Compact it!

```
switch i32 %val, label %otherwise [
  i32 -1..2, label %dest1
  i32     3, label %dest2
]
```

Real-World Examples

253.perlbmk

```
switch i32 %tmp, label %bb36 [
  i32 0,   label %bb30
  i32 5,   label %bb72
  i32 6,   label %bb55
  i32 7,   label %bb55
  i32 13,  label %bb100
  i32 31,  label %bb100
  i32 32,  label %bb100
  i32 33,  label %bb93
  i32 35,  label %bb88
  i32 126, label %bb55
  i32 173, label %bb62
  i32 174, label %bb62
  i32 184, label %bb80
  i32 185, label %bb80
  i32 186, label %bb80
  i32 188, label %bb80
  i32 310, label %bb33
]
```

Python 2.5

```
switch i32 %c1, label %bb72 [
  i32 33,  label %bb6
  i32 37,  label %bb56
  i32 38,  label %bb61
  i32 42,  label %bb37
  i32 43,  label %bb27
  i32 45,  label %bb32
  i32 47,  label %bb44
  i32 60,  label %bb11
  i32 61,  label %bb
  i32 62,  label %bb20
  i32 94,  label %bb66
  i32 124, label %bb51
]
```

Strategy: Series of Comparisons

Naive approach:

- Just emit series of comparisons.
- Note, we need only 1 *unsigned* comparison to check case range (shown in example).

Naive, but:

- Zero overhead!

Before

```
switch i32 %v, label %bb1 [
  i32 -1..2, label %bb2
  i32     3, label %bb3
]
```

After

```
entry:
  %t1 = add i32 %v, 1
  %t2 = icmp ule i32 %t1, 3
  br i1 %t2, label %bb2, label %next
next:
  %t3 = icmp eq i32 %v 3
  br i1 %t3, label %bb3, label %bb1
```

Strategy: Series of Comparisons

Naive approach:

- Just emit series of comparisons.
- Note, we need only 1 *unsigned* comparison to check case range (shown in example).

Naive, but:

- Zero overhead!

Before

```
switch i32 %v, label %bb1 [
  i32 -1..2, label %bb2
  i32     3, label %bb3
]
```

After

```
entry:
  %t1 = add i32 %v, 1
  %t2 = icmp ule i32 %t1, 3
   br i1 %t2, label %bb2, label %next
next:
  %t3 = icmp eq i32 %v 3
   br i1 %t3, label %bb3, label %bb1
```

Strategy: Jump Table

If switch is dense enough we can emit *jump table*:

- Header, which subtracts the minimal case from the input value.
- Case set span check.
- An array of addresses of all destinations.
- Branch instruction which takes index and array of destinations as arguments.

Possible problems:

- Not all targets support jump tables.
- Big overhead due to pipeline stalls.
- We need to represent *every possible case* (even default). Thus - big size of data generated.

Strategy: Jump Table

If switch is dense enough we can emit *jump table*:

- Header, which subtracts the minimal case from the input value.
- Case set span check.
- An array of addresses of all destinations.
- Branch instruction which takes index and array of destinations as arguments.

Possible problems:

- Not all targets support jump tables.
- Big overhead due to pipeline stalls.
- We need to represent *every possible case* (even default). Thus - big size of data generated.

Jump Table Example

Before

```
switch i32 %v, label %bb1 [
  i32 -1..2, label %bb2
  i32     3, label %bb3
]
```

After

```
entry:
    movl  val, %eax
    incl  %eax
    cmpl  $4, %eax
    ja    bb1
    jmpl  *JTI1(,%eax,4)
JTI1:
   .long  bb2
   .long  bb2
   .long  bb2
   .long  bb2
   .long  bb3
```

Strategy: Balanced Binary Tree

Idea: store case values somewhere in the sorted order and use binary search to find needed case in logarithmic of number of cases time.

Implementation:

1. Select middle case from the current case set ('pivot'). The pivot splits the whole case set into 'left' and 'right' subsets.
2. Emit signed comparison of input value with the pivot.
3. Repeat steps 1-2 for 'left' and 'right' subtrees, until current case set is exactly one case.
4. Emit comparison of input value with remaining case. Branch to case destination if equals, to default destination otherwise.

Balanced Binary Tree Example

Before

```
switch i32 %v, label %bb1 [
  i32 -1..2, label %bb2
  i32     3, label %bb3
]
```

After

```
entry:
  %p   = icmp slt i32 %v, 3
   br i1 %p,  label %l1,  label %l2
l2: ; leaf 2
  %t2  = icmp eq i32 %v, 3
   br i1 %t2, label %bb3, label %bb1
l1: ; leaf 1
  %off = add i32 %v, 1
  %t1  = icmp ule i32 %off, 3
   br i1 %t1, label %bb2, label %bb1
```

Strategy: Shift-And

Consider we have the following switch (usually a result of if-series conversion):

Another Example from Python 2.5

```
switch i32 %c, label %bb [
  i32 91,  label %bb2
  i32 93,  label %bb3
  i32 100, label %bb2
  i32 101, label %bb3
  i32 113, label %bb2
  i32 115, label %bb3
]
```

Can we emit it better than via ugly binary tree?

Shift-And Continued

Yes, surely! Consider these conditions are met:

1. Total number of unique destinations of switch is not so big.
2. Case set span of the switch is less than bitwidth of general purpose register of our target machine.

Algorithm

- Create *destination mask* for each unique destination. The bit of mask is set to '1' if corresponding case goes to the selected destination and is set to '0' otherwise.
- Use switch input value to create *input mask* at run-time.
- 'And' *input mask* with pre-computed *destination masks*. Branch to corresponding destination BB if not zero.

Shift-And Example

Before

```
switch i32 %c, label %bb [
  i32 91,  label %bb2
  i32 93,  label %bb3
  i32 100, label %bb2
  i32 101, label %bb3
  i32 113, label %bb2
  i32 115, label %bb3
]
```

After

```
entry:  ; Perform range comparison
  %t1 = sub i32 %c, 91
  %t2 = icmp ugt i32 %t1, 24
   br i1 %t2, label %bb, label %header
header: ; Compute runtime mask
  %t3 = shl i32 1, %t1
   br label %case1
case1:  ; Compare with first mask
  %t4 = and i32 %t3, u0x400201
  %t5 = icmp ne i32 %t4, 0
   br i1 %t5, label %bb2, label %case2
case2:  ; Compare with second mask
  %t6 = and i32 %t3, u0x1000404
  %t7 = icmp ne i32 %t6, 0
   br i1 %t7, label %bb3, label %bb
```

Strategy: Binary Tree With Custom Pivot Selection

How we should operate, if we want to stack different strategies to emit switch?

- Idea: always emit binary tree and try to apply different strategies during emission of subtrees.
- Problem: usual binary tree is not smart enough. For example, if we're emitting jump tables we are interested in the best pivot maximizing densities of right and left subtree.

Best pivot criteria:

- Should not split tree in trivial cases, for example, in the middle of the dense case set.
- Split switch maximizing the distance between subtrees.

Strategy: Binary Tree With Custom Pivot Selection

How we should operate, if we want to stack different strategies to emit switch?

- Idea: always emit binary tree and try to apply different strategies during emission of subtrees.
- Problem: usual binary tree is not smart enough. For example, if we're emitting jump tables we are interested in the best pivot maximizing densities of right and left subtree.

Best pivot criteria:

- Should not split tree in trivial cases, for example, in the middle of the dense case set.
- Split switch maximizing the distance between subtrees.

Strategy: Binary Tree With Custom Pivot Selection

How we should operate, if we want to stack different strategies to emit switch?

- Idea: always emit binary tree and try to apply different strategies during emission of subtrees.
- Problem: usual binary tree is not smart enough. For example, if we're emitting jump tables we are interested in the best pivot maximizing densities of right and left subtree.

Best pivot criteria:

- Should not split tree in trivial cases, for example, in the middle of the dense case set.
- Split switch maximizing the distance between subtrees.

Switch Split Metric – Pivot Quality

We can now introduce metric used to split a switch.

- Denote l_i and r_i the densities of the left and right subtrees being split at pivot i.
- Denote a_i the maximal case of the left subtree and b_i – minimal case of the right subtree.

The *quality* q_i of pivot i is defined as:

$$q_i = (l_i + r_i) \log(b_i - a_i)$$

Features:

- Trivial cases (splitting in the middle of the dense block) are thrown.
- More far subtrees are selected, when sum of the densities for two different pivots are close.

Glueing Things Together

Worklist-driven Algorithm

1. Push original switch into worklist.
2. Pop current case set from worklist.
3. Try to apply BitTest (Shift-And) strategy
4. Try to apply SmallSwitch (series of comparisons) strategy
5. Try to apply JumpTable strategy
6. Emit binary tree:
   1. Select pivot of maximal quality
   2. Split case set at this pivot.
   3. Push two new case sets into worklist.
7. Iterate 2-6 until worklist is not empty.

Results

Testsuite

- 253.perlbmk from SPEC'2006
- llvm-gcc4 itself (bootsrapped and used to compile huge source sets)

Speed Gain

- 5-7% on compilation of huge source sets (Mozilla and Qt library were used)
- 5% on 253.perlbmk