Improving Switch Lowering for The LLVM Compiler System

Anton Korobeynikov
Saint Petersburg State University
Email: asl@math.spbu.ru

***Abstract*—Switch-case statements (or switches) provide a natural way to express multiway branching control flow semantics. They are common in many applications including compilers, parsers, text processing programs, virtual machines. Various optimizations for switches has been studied for many years. This paper presents the description of switch lowering refactoring recently made for the LLVM Compiler System [1].**

I. Introduction

Many imperative high-level programming languages offer a form of multiway branch statement. The examples are *switch* statement in C/C++/Java, *case* in Pascal, etc. Switches can be usually seen in many sophisticated programs like compilers, virtual machines, parsers because these applications often use some value to select an action among a large collection of possible. Almost all finite state machines implementations uses switches in order to translate from one state to another. Such FSMs, for example, are generated on-the-fly in applications doing any sort of pattern matching.

This is compiler task to examine the switch and emit the most effective code for this platfrom to implement multiway branches. The possible issues involved in producing such code are non-trivial. The main tradeof is, as usual, size-speed. However, in most real-world cases small size of generated code for multiway branch facility implies its big speed, since switches are often used in the tight inner loops of the application.

This paper presents description of the switch lowering refactoring made in the LLVM Compiler System recently. Section II introduces briefly to the LLVM itself, its goals, advantages and fields of use. Section III is devoted to the miscellaneous strategies of switch emission and the concrete description of the LLVM switch lowering algorithm.

II. The LLVM Compiler System

The LLVM Compiler System was originally designed as a compiler framework to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time and in the idle time between runs [2].

Currently the key features of LLVM are:

- Has many built-in sophisticated optimization algorithms.
- Can be used as a JIT compiler.
- It is retargettable, supports many targets.
- Has Link Time Optimization (LTO) support.
- Can be successfully used as a drop-in replacement for GCC.

The LLVM code representation describes a program using an abstract RISC-like instruction set but with key higher-level information for effective analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation (using an infinite, typed register set in Static Single Assignment form) [3]. LLVM provides equivalent textual, binary and in-memory code representation.

Let's see, how LLVM code looks like. Consider the naive recursive function, which calculates the Fibonacci sequence:

```
unsigned fib(unsigned n) {
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```

The corresponding LLVM code is:

```
define i32 @fib(i32 %n) {
entry:
  %tmp2 = icmp ult i32 %n, 3
  br i1 %tmp2, label %return, label %next
next:
  %tmp6  = add  i32 %n, -1
  %tmp7  = call i32 @fib(i32 %tmp6)
  %tmp9  = add  i32 %n, -2
  %tmp10 = call i32 @fib(i32 %tmp9)
  %tmp11 = add  i32 %tmp7, %tmp10
  ret i32 %tmp11
return:
  ret i32 1
}
```

One can see here some of LLVM's key features including embedded type information and SSA form of the representation. LLVM's built-in optimization passes can detect tail recursion in this case and optimize it out.

All optimizations can be performed in LLVM in two ways:

1) As LLVM-to-LLVM pass taking LLVM code representation as input and producing LLVM code as output.
2) During codegeneration for target machine.

The second approach is more suitable for our work, since it allows us to use some target machine information, resulting to much better code. We cannot perform well with LLVM-to-LLVM pass, since LLVM representation is *target machine independent*.

The LLVM target-independent code generator is a framework that provides a suite of reusable components for translating the LLVM internal representation to the machine code for a specified target - either in assembly form (suitable for a static compiler) or in binary machine code format (usable for a JIT compiler). The LLVM target-independent code generator consists of five main components:

1) Abstract target description interfaces which capture important properties about various aspects of the machine, independently of how they will be used.
2) Classes used to represent the machine code being generated for a target. These classes are intended to be abstract enough to represent the machine code for any target machine.
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. These machine descriptions make use of the components provided by LLVM, and can optionally provide custom target-specific passes, to build complete code generators for a specific target.
5) The target-independent JIT components. The LLVM JIT is completely target independent (it uses special structure to interface for target-specific issues).

Lowering is the process of turning one instruction (**switch** in our case) into series of more low-level instructions. This process surely depends on the target machine and the switch itself.

III. Switch Lowering Implementation

Switches are represented in the LLVM with the help of **switch** instruction. The syntax for this instruction is:

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

The **switch** instruction uses three parameters: an integer comparison value *value*, a default *label* destination, and an array of pairs of comparison value constants and *label*'s. The table is not allowed to contain duplicate constant entries.

Some examples will follow.

```
; Emulate a conditional br instruction
%Val = zext i1 %value to i32
switch i32 %Val, label %truedest [
  i32 0, label %falsedest
]

; Emulate an unconditional br instruction
switch i32 0, label %dest [ ]

; Implement a jump table:
switch i32 %val, label %otherwise [
  i32 0, label %onzero
  i32 1, label %onone
  i32 2, label %ontwo
]
```

Consider the following 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
]
```

One can see, that cases cases 0-2 goes to one destination and case 3 – to another. So, we can "compact" the switch writing such LLVM "pseudocode":

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

We call this "pseudocode", because LLVM does not support switches with "compact" case ranges (like gcc *trees*, for example) now. The main advantage of this representation is that we can go to label *dest1* just emitting range comparison (so do only 2 comparisons) and not comparison per each case entry. Internally we will operate on switches in this form. Also we are assuming that cases are sorted withing switch.

This source will be used later as a source switch being lowered. It will be called "example switch" everywhere, otherwise explicitly mentioned.

We will use different metrics to examine switch. Two of them are:

1) Switch density. The definition is straightforward.
2) Number of comparisons needed to emit all cases. This is just sum of number of comparisons needed to emit each switch case. One comparison is needed to emit single case, two – for case range.

A. Different Strategies to Lower Switch

1) Small Switches: All other strategies involve some overhead needed to prepare switch data for future use. This overhead can be significant for small switches. We consider (compacted) switch to be small, if it contains 3 or less cases. In this case we're just emitting series of branches. Also note, even we are saying, that case range uses two comparisons, actually, we need only one *unsigned* comparison plus some additional arithmetical operations. This "trick" is shown below during lowering of our example switch.

```
entry:
  %tmp1 = add i32 %val, 1
  %tmp2 = icmp ule i32 %tmp1, 3
  br i1 %tmp2, label %dest1, label %next
next:
  %tmp3 = icmp eq i32 %val 3
  br i1 %tmp3, label %dest2, label %otherwise
```

This strategy is just a linear search over all cases. So, its runtime is linear on number of cases.

2) Bit Tests: This is actually a "trick" strategy. Consider our target machine has general purpose register of n bits length. Also, let case set span of the switch is less than n and total number of unique destination of switch (excluding default case) is not so big (we are using at most 3 unique destinations). For each unique destination we can compute its own n bit mask. Bit 0 of mask corresponds to the minimal case, bit 1 to the next, etc. The bit of mask is set to '1' if corresponding case goes to the selected destination and is set to '0' otherwise. As the result we will have at most 3 "orthogonal" masks. During runtime we can shift 1 to the left by the value of the input value, perform logical *and* with the mask and shifted 1 and jump to the corresponding destination if the result is not zero. This technique is shown in the code below:

```
entry:
  %tmp1 = add i32 %val, 1
  %tmp2 = icmp ugt i32 %tmp1, 4
  br i1 %tmp2, label %otherwise, label %header
header:
  %tmp3 = shl i32 1, %tmp1
  br label %case1
case1:
  %tmp4 = and i32 %tmp3, 16
  %tmp5 = icmp ne i32 %tmp4, 0
  br i1 %tmp5, label %dest1, label %case2
case2:
  %tmp6 = and i32 %tmp3, 15
  %tmp7 = icmp ne i32 %tmp6, 0
  br i1 %tmp7, label %dest2, label %otherwise
```

We see, that this strategy has some overhead: it should perform case set span comparison and shift before actual checks. Checks itself are cheap: only one 'and' operation and one comparison for *each unique destination*. So, the running time of the strategy is linear on number of unique destination. That is why it is extremely useful, when we have switch with few destinations and sparse case set, like the one shown below.

```
switch i32 %val, label %otherwise [
  i32 1,  label %dest1
  i32 5,  label %dest1
  i32 9,  label %dest1
  i32 12, label %dest1
  i32 30, label %dest1
]
```

This technique performs better on 64-bit targets, than on 32-bit targets, since maximum allowed case set span is more. Such switches can be easily seen in different parsers after dead code elimination.

3) Jump Tables: If switch is dense enough we can just emit a *jump table*. Jump table consists of header, which subtracts the minimal case from the input value, case set span check, an array of addresses of all destinations and a branch instruction which takes index and array of destinations as arguments. Branch instruction jumps to the destination pointed by index argument in the array.

LLVM IR does not contain any jump tables related instructions. Also, not all targets supports jump tables due to lack of needed "branch via index" hardware instructions. We are showing x86 32-bit assembler code as an example.

```
entry:
    movl  val, %eax
    incl  %eax
    cmpl  $4, %eax
    ja    otherwise
    jmpl  *JTI1(,%eax,4)
JTI1:
   .long dest1
   .long dest1
   .long dest1
   .long dest1
   .long dest2
```

4) Balanced Binary Tree: Balanced binary tree emission is widely used technique for switch lowering. Almost every compiler has some variant of this strategy implemented. The idea is well-known: store case values somewhere in the sorted order and use binary search to find needed case in logarithmic of number of cases time. Being implemented at low level (with the use of comparisons and branches) this strategy shows only little overhead. The algorithm itself is:

1) Select middle case from the current case set. Call it the 'pivot'. The pivot splits the whole case set into two subsets: one with all elements less than pivot and other with elements greater or equal than the pivot.
2) Emit signed comparison of input value with the pivot. If less, let current case set be "left" subset and "right" subset otherwise. Emit branches to the "left" subtree checking and the "right" subtree checking.
3) Repeat steps until current case set contains exactly one case.
4) Emit conditional branch: if current input value equals the current case branch to corresponding destination or to default destination otherwise.

Example LLVM code is

```
entry:
  %pivot = icmp slt i32 %val, 3
  br i1 %pivot, label %leaf1, label %leaf2
leaf2:
  %tmp2  = icmp eq i32 %val, 3
  br i1 %tmp2, label %dest2, label %otherwise
leaf1:
  %off   = add i32 %val, 1
  %tmp1  = icmp ule i32 %off, 3
  br i1 %tmp1, label %dest1, label %otherwise
```

5) Binary Tree With Custom Pivot Selection: When we try to stack different strategies the usual binary tree is not smart enough. For example, if we are combining binary tree and jump tables emission we are interested in the best pivot, which maximizes densities of right and left subtrees. This, heuristically, leads to denser subtrees, thus such pivot will allow us to emit jump table for the whole subtree at some step.

However, we should not split tree in trivial cases, for example, in the middle of the dense case set, even if sum of

TABLE I
EXAMPLE CASE SET

Case	Binary
0	0 00000 00_2
1	0 00000 01_2
129	1 00000 01_2
131	1 00000 11_2

densitites attends maximum at this pivot. Also, it will be better to split switch maximizing the distance between subtrees.

Taking into account all mentioned arguments, we are assigning each pivot its own *quality*. Let l_i and r_i denote the densities of the left and right subtrees being split at pivot i. Let a_i denote 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

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

We are selecting pivot with maximum quality. All other details are the same as for ordinary binary tree. Such pivot selection procedure will surely "throw out" trivial cases (splitting in the middle of the dense block) and will prefer more far subtrees, when sum of the densities for two different pivots are close.

Surely, resulting tree is not balanced any more, thus this strategy should be used only in connection with other strategies like jump tables emission.

6) Multiway Radix Search Trees: Multiway radix search trees (or MRST for short) is a novel technique first presented in [4]. In general it is a sort of non-perfect hashing but designed specially for static sparse search trees. This is really important method, since it can turn sparse case set into series of much more denser sets suitable for applying of other strategies listed here. We will cite the description from [4].

The algorithm looks at the input set of cases and finds a short sequence, or *window*, of consecutive bits that distinguishes the cases into several subsets. The algorithm generates code that branches on the value of the window in the run-time input thus selecting a specific set of candidate cases based on the input value. For empty sets, the branches lead to the default handle; for sets containing one case, to simple comparison of the case and the input value; and for larger sets, to recursive invocations of the algorithm.

It is desirable to find long windows, in order to make the search tree wide and shallow. However, the length of the windows should be limited, since the branches require hash tables that grow exponentially in the window length. The simple greedy strategy of finding the longest *critical* window that distinguishes the cases into more than a threshold number of subsets relative to the window length is used. In [4] it is proven, that average number of branches for MRST algorithm is at most three.

Consider the table I. We see, that cases 0, 1, 129 and 131 are almost uniquely determined by their last two bits. The one exception is the pair 1 and 129, where bit 7 can determine the case. Therefore we can quickly narrow down the candidate cases for a given run-time input by hashing first on the last two bits and then, if necessary, on bit 7, the leftmost bit. Finally, we can compare the input value with the unique remaining candidate case. Note that the code should branch directly to the default handler for any run-time input ending in 10_2 since no case ends in this bit pattern.

MRST tends to be global switch rewriting strategy. That is why it is hard to use it "stacked" with other methods. Consider the switch with two dense case sets separated far away. In most cases, it will be better to split such switch into two case ranges and emit two jump tables, not to apply MRST.

B. Switch Lowering

Before our work LLVM's switch lowering algorithm was simple: at first it tried to emit jump table, if the switch had been dense enough. Otherwise the algorithm just emitted balanced binary search tree. Also, algorithm operated on the switch itself, not its "compacted" form. This leads to inefficient code in many cases.

Current LLVM switch lowering algorithm is simple worklist-driven algorithm trying to apply different strategies for case set popped from the worklist.

Algorithm 1 Current LLVM switch lowering algorithm

Require: Switch SI in "compacted" form.
Ensure: Switch SI lowered with different strategies.
  push SI to $WorkList$;
  repeat
    pop cSI out of $WorkList$;
    if (case set span of cSI is small) & (number of unique destinations of cSI is also small) **then**
      emit bit tests;
    else if (number of cases in $cSI \leq 3$) **then**
      emit series of branches; (*"Small switch" strategy*)
    else if cSI is not less than 40% dense **then**
      emit jump table;
    else
      let P be the pivot of maximum quality;
      emit comparison with P;
      split cSI into two subswitches lSI and rSI at P;
      push lSI to $WorkList$;
      push rSI to $WorkList$;
    end if
  until $WorkList$ is not empty

IV. RELATED WORK

There have been a few papers discussing the issues arising in optimizations of switches.

Different methods were studied by Bernstein in [5] including linear search, binary search and jump tables. He found, that linear search is faster than the jump table for small switches (when number of cases is 3 or less) and that binary search is faster than linear search for big switches. The methods of breaking big switch into smaller and denser clusters were also examined. He proposed, that optimal solution of the

TABLE II
COMPILATION TIMES

Case	Old switch lowering	New switch lowering
Mozilla Suite	37^m47^s	35^m21^s
Qt Library	28^m54^s	27^m01^s

clustering problem is NP-complete. However, in [6] Kannan and Proebsting presented optimal clustering algorithm, which runs in $O(n^2)$ time.

In [7] switch lowering problem was studied from the position of a static search problem. A summary of the various methods were presented, and these methods were analyzed in terms of time and space complexity of the executable code, and also compilation time. A general algorithm for code generation was presented and was shown to improve Bernstein's algorithm in a number of ways.

MRST algorithm was firstly presented in [4]. Authors discussed other possible switch lowering strategies including linear search, balanced binary tree, combination of binary trees and hash tables, and jump tables in comparison with MRST. Also the implementation of MRST algorithm for *lcc* compiler was presented.

In [8] the switch lowering problem was examined when there are additional information about the case hit frequency. Authors presented different strategies such as *hot case hoisting* and *switch case partitioning* which lead to significant reduction of running time. So, actually, they are studied *profile-driven* switch lowering.

Some approaches assume that switch cases have been transformed to a series of conditional branches and try to reorder these branches according to their importance [9] or convert them into indirect jumps [10].

V. Results, Conclusion and Future Work

Unfortunately, SPEC testsuite was unavailable for author. That is why we perform comparison in the following way:

- Perform llvm-gcc compiler bootstrap with and without improving switch lowering algorithm.
- Build Mozilla Suite and Trolltech' Qt library with both compilers
- Compare build times.

All tests were performed on dual-core AMD Opteron 2210 workstation with 2Gb RAM. Only one core was used for tests. The results are shown in the table II.

Possible future work is summarized below.

- We haven't added MRST algorithm yet due to problems noted in the section III-A6. Actually, we need to develop some *local sparseness* metric, which can be used to decide, whether MRST algorithm or other techniques should be run on switch.
- We haven't studied any profile-driven switch lowering. We suppose, that impact from such optimization can be significant.
- It can be interesting to try to implement switch case clustering algorithm from [6] and compare it with another strategies.

Acknowledgment

Author would like to thank Chris Lattner for the help and many interesting ideas.

References

[1] The LLVM Compiler System. [Online]. Available: http://llvm.org
[2] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. [Online]. Available: http://llvm.org/pubs/2004-01-30-CGO-LLVM.html
[3] C. Lattner, "Macroscopic Data Structure Analysis and Optimization," Ph.D. dissertation, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. [Online]. Available: http://llvm.org/pubs/2005-05-04-LattnerPHDThesis.html
[4] U. Erlingsson, M. S. Krishnamoorthy, and T. V. Raman, "Efficient Multiway Radix Search Trees," *Information Processing Letters*, vol. 60, no. 3, pp. 115–120, 1996. [Online]. Available: http://citeseer.ist.psu.edu/erlingsson96efficient.html
[5] R. L. Bernstein, "Producing good code for the case statement," *Softw. Pract. Exper.*, vol. 15, no. 10, pp. 1021–1024, 1985.
[6] S. Kannan and T. Proebsting, "Correction to Producing Good Code for the Case Statement," 1994. [Online]. Available: http://citeseer.ist.psu.edu/kannan94correction.html
[7] D. A. Spuler, "Compiler code generation for multiway branch statements as a static search problem," Department of Computer Science, James Cook University, Australia, Tech. Rep., January 1994.
[8] P. Zhao and J. N. Amaral, "Feedback-directed switch-case statement optimization," in *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 295–302. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1488708
[9] M. Yang, G.-R. Uh, and D. B. Whalley, "Efficient and effective branch reordering using profile data," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, pp. 667–697, 2002.
[10] G.-R. Uh and D. B. Whalley, "Effectively exploiting indirect jumps," *Software-Practice and Experience*, vol. 29, no. 12, pp. 1061–1101, 1999. [Online]. Available: http://citeseer.ist.psu.edu/article/uh97effectively.html