Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Switch Statement Case Reordering FDO

Edmar Wienskoski
Freescale
edmar@freescale.com

Abstract

When gcc parses a switch statement, it uses some criteria to decide between generating a jump table or a binary search tree. The jump table has a fixed significant cost, and for large switch statements the sequence of compare-branches from the root to the leaves can also be costly.

The optimization described here collects a histogram of a switch statement condition expression and uses it to balance the binary search tree. We also implement a default statement promotion: Single values in the histogram that maps to default in the switch statement, are candidates to become a new node in the binary search tree, which gives a chance to that particular value to have its path on the tree optimized.

1 Motivation

In looking at the perl benchmark in Spec2k for optimization opportunities, it was noticed that the basic blocks more often executed where part of one big switch statement inside the main interpreter loop.

From the switch condition expression evaluation to those basic blocks, there were many instruction cycles. A feedback directed optimization that moves the most often executed case statements out of the switch statement could improve the performance of this benchmark considerably.

To test the effectiveness of pursuing this optimization: I changed the original source code of function `regmatch` on file `regexec.c` such that the two case statements more often executed were hoisted outside the switch statement. Figure 1 illustrates the idea, where em x and *y* represents the two case statements in question. The result was a surprising 17% improvement.[1]

```
                          if (x){
  switch (c){               :
    case a:               } else
      :                   if (y){
    case x:                 :
      :             =>    } else
    case y:               switch (c){
      :                     case a:
    case b:                   :
      :                     case b:
                              :
```

Figure 1: Hoisting case statements out of the switch

Gcc infrastructure has developed quite fast re-

[1]This was obtained with the Motorola research compiler and a G3 Linux PowerPC machine. The same experiment with gcc development branch *tree-profiling-branch* on a G5 Linux PowerPC machine yields 12.6% improvement.

cently: SSA based optimizations, auto vectorization, and feedback directed optimizations among them. All that activity motivated us to apply the above idea into gcc.

2 The implementation

Currently, gcc has two strategies to generate code for a switch statement: a jump table, or a binary search tree.

The jump table offers uniform access time for all case statements, but that time is usually much slower than a single compare-and-branch sequence. On the other hand, the binary search tree requires several compare-and-branch sequences to reach the leaves of the tree. As the number of case statements increases, the length of the compare-branches sequences from the root of the binary search tree to their leaves also increases.

Considering that in a binary search tree 50% of all nodes are leaves,[2] the chance of one of the most executed case statements ending up either in a leaf or in a node next to a leaf is quite high. That of course, would be the worst possible performance outcome. Thus, whenever the number of case statement is above some threshold, gcc will give preference to generate the jump table instead of the binary search tree in order to avoid risking that performance penalty.

But in fact, the gcc infrastructure supports a weight[3] attribute to each node of binary search tree. That effectively allows of the implementation of balanced binary search tree algorithm. Unfortunately, this feature is not used effectively to avoid the performance penalty described in the previous paragraph. Currently, gcc attributes weights of 1 to all nodes that represents one single value of the switch statement condition expression, and weight of 2 to nodes that represents a range of values.[4]

Considering the infrastructure available in gcc to balance a binary search tree, it is a better idea to tweak the weight of the binary search tree nodes, instead of implementing all the possible basic block manipulations and condition expression generations to achieve the code transformation depicted in Figure 1.[5] The result of this strategy could be as effective as the proposed solution, as Figure 2 illustrates.

Another consideration in the implementation of this optimization is how gcc collects feedback information.

Gcc can instrument basic blocks to measure basic block frequency, but can also instrument the target application to compute a histogram of run time values of any expression in the application. That includes the computation of histograms of switch statement condition expressions.

The basic block counters could had be used to weigh the binary search tree, but the histogram gives us one extra optimization opportunity: To promote values that map to the default statement into individual nodes in the binary search tree.

In general, all run time values of the condition expression that maps to the default statement, causes the execution of a sequence of compare-branch instructions that goes from the root of

[2] Except for some odd shaped trees, the observation is quite accurate for balanced or not trees as long as each internal node has 2 siblings.

[3] Gcc sources name this attribute as `cost`. We will now refer to weight or cost interchangeably.

[4] Incidentally, the switch statement in function `regmatch` on file `regexec.c` of Spec2k perl benchmark consists of single values only.

[5] It would also be much easier to debug and prove correctness of the optimization.

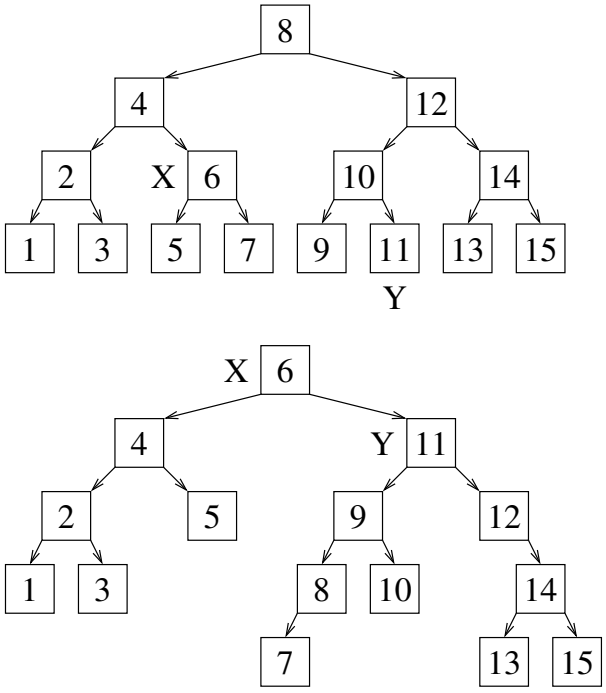

Figure 2: A proper cost function can cause the tree balancing algorithm to move the nodes *X* and *Y* closer to the root

the binary search tree to one of its leaves, before it can be identified as one default value. That means the execution of the default statement will always incur in the highest performance penalty. But what if the default statement is among the ones with highest execution frequency?

Default value promotion is the answer. The idea is to identify one particular value in the histogram that maps to the default statement, and whose execution frequency is high, (This can be easily done by inspection of the histogram and some heuristic to define "high execution frequency"), and then create a new node in the binary search tree for that value. This is semantically equivalent to the code transformation illustrated in Figure 3.

After balancing the tree, the path between the root and the newly created node can be optimized, leaving the general performance penalty of the default statement to the less frequent occurring values.

```
switch (c){                    switch (c){
  case a:                        case a:
    :                              :
  default:          =>           case x:
    :                            default:
}                                  :
                               }
```

Figure 3: The value *x* on the histogram of condition expression *c* satisfies the "high execution frequency" criteria.

Examples of heuristics that could be used are: A fixed threshold (e.g.: 10% of execution frequency); and a relative parameter (e.g.: Highest five values in the histogram).

One last implementation detail concerns the case when the case statement labels are too sparse, and some case labels are below the histogram range and / or some are above it. If that happens, the execution frequency of values below and / or above the histogram range are equally distributed among the corresponding labels.

3 Results

The optimization described in this paper was implemented and tested on a snapshot of the *tree-profile-branch* development branch from May 24, 2005 [TPB].

All of Spec2k was validated on a G5 running Linux PowerPC [YDL, YHPC]. Except for perl, all other benchmarks had no variation in performance and are omitted from this discussion.

	Individual parts							Σ	%
Original gcc	21.71	1.23	18.76	63.60	36.16	32.22	59.93	233.61	—
Case stmt hoisted	21.43	1.24	18.72	53.80	31.82	27.71	52.63	207.35	12.6
Balanced - train	18.99	1.19	17.94	56.20	32.77	28.71	54.32	210.12	11.2
Balanced - validation	19.29	1.23	17.95	51.51	30.42	26.44	50.30	197.04	18.5
Balanced & hoisted	18.74	1.20	17.84	52.21	31.03	26.96	51.57	198.35	17.7

Table 1: Spec2k perl benchmark execution times. Individual parts and total time shown in seconds.

On table 1 we show all Spec2k perl results. The first line of data, was obtained with the original compiler and is used as base reference for relative comparison. The same level of optimization was used across all executions, including other profile feedback optimizations. The second line, was obtained with the benchmark manually optimized as depicted in Figure 1. The third and fourth lines, were obtained with the switch statement case reordering enabled. On the third, the benchmark was trained with the *train* data set. On the next one the *validation* data set was used instead. Finally, the last line has both manually hoisted case statements and the switch statement case reordering.

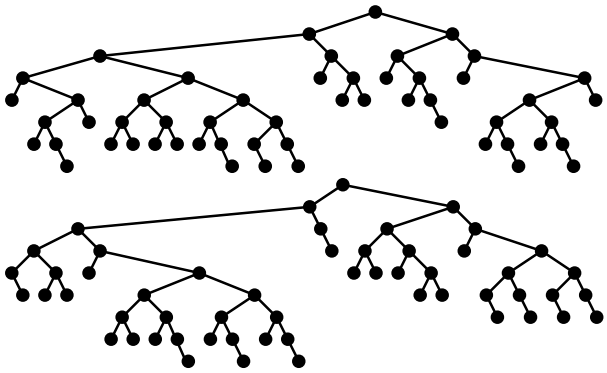

Figure 5: Binary search tree of the switch statement condition expression of function `regmove` on file `regexec.c` obtained with the train data set and validation data set respectively

The reader should note that, in order to manually apply the case statement hoisting as described in Section 1 of this paper, one must have the knowledge of which two case statements were more often executed in the validation execution of the benchmark. But that knowledge is not available when the Spec test harness is used. Because the data set used to train the application may not represent adequately the data set used for validation, a direct comparison of the second and third lines would not be fair. That is the reason for having the fourth line on Table 1. By training the application with the same data set used for validation, we ensure that both the switch statement case reordering fdo and the manually applied case hoisting have the same knowledge base. We don't claim any official Spec results, per Spec rules.

For illustration, Figure 4 shows the histogram of the switch statement condition expression of function `regmove` on file `regexec.c`. The left columns are the values of the histogram obtained using the train data set, the right columns are the values of the histogram obtained using the validation data set. Figure 5 shows the actual binary search tree after balancing it. The one in the top was obtained with the training data set, the one on the bottom was obtained with the validation data set.

On Section 2 it was argued that rebalancing the binary search tree would be enough to obtain the same results as hoisting case statements out of the switch. The fifth line on Table 1 confirms that: in the presence of switch statement

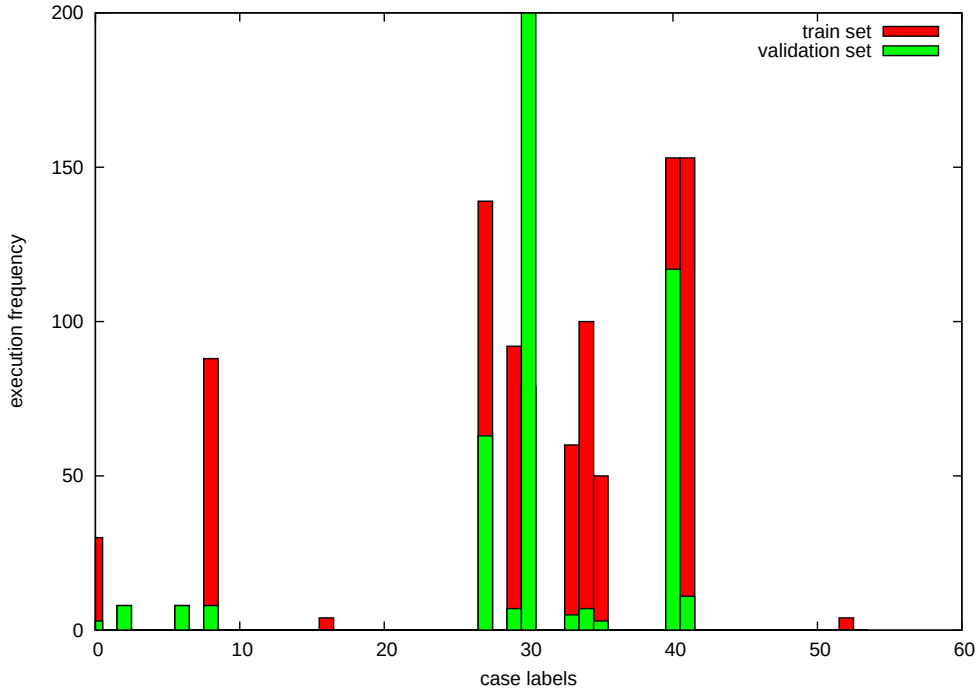

Figure 4: Histograms of the switch statement condition expression of function `regmove` on file `regexec.c` obtained with the train data set (left columns) and validation data set (right columns).

case reordering, hoisting the two most often executed case statements is ineffective.

	Time	%
train data set	202.42	—
validation data set	204.58	-1.0
Case stmt hoisted	193.52	4.6

Table 2: Spec2k perl benchmark execution time in seconds, using xlc 7.0 compiler.

To contrast, on Table 2 we have a set of Spec2k perl results for the IBM compiler [XLC] running in the same machine.[6]

On table 3 we show all Spec95 m88ksim results. Those results were obtained in the same G5 Linux PowerPC machine. The first line of data was obtained with the original compiler. The second and third lines, were obtained with the switch statement case reordering enabled and training with the train and validation data set respectively.

This benchmark has a total of 66 switch statements, 56 of them were never executed. Among the other 10, the two largest switch statements have 38 and 16 case labels respectively. Figure 6 shows the binary search tree for those two switch statements after balancing them. Note the heavy effect of the balancing in the symmetry of the trees.

As discussed in Section 2, gcc makes a tradeoff between generating a jump table and a binary search tree. As the memory latency of the target architecture decreases, one can expect the importance of this tradeoff to be less significant or non-existing altogether. This trend is visible on Table 4, it summarizes all perl and m88ksim results for two other PowerPC parts that has re-

[6]The benchmark was compiled with flags -O5, and -qpdf1 / -qpdf2.

	Time	%
Original gcc	30.25	—
Balanced - train	27.25	11.0
Balanced - validation	26.83	12.7

Table 3: Spec95 m88ksim benchmark execution time in seconds.

	7450				8548			
	perl		m88ksim		perl		m88ksim	
	Time	%	Time	%	Time	%	Time	%
Original gcc	454.8	—	46.0	—	501.0	—	60.0	—
Balanced - train	443.2	2.6	46.0	0.0	505.3	-0.8	59.8	0.3
Balanced - validation	428.7	6.1	46.0	0.0	469.7	6.8	59.9	0.2

Table 4: Summary of results for two other PowerPC parts: the 7450 and the 8548.

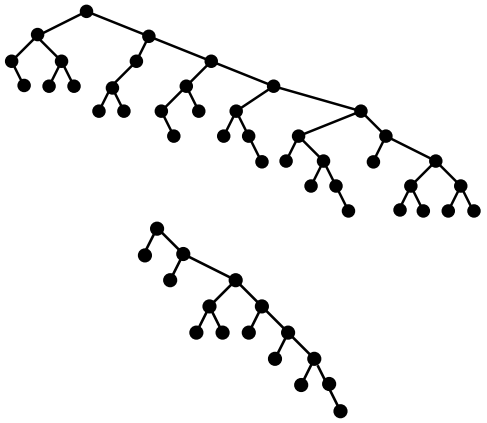

Figure 6: Binary search trees of two large switch statement on Spec95 m88ksim benchmark.

spectively smaller memory latencies: the 7450, and the 8548.

4 Conclusion and Future work

We showed how feedback directed optimization can be used to improve large switch statements performance, for both explicit case statements and default statements.

Futhermore, that this optimization can be implemented in gcc by leveraging existing infrastructure, namely the capacity to generate balanced binary search trees for switch statements, and the capacity to create run time value histograms of the switch statement condition expressions of the target application.

In conclusion, for such small code change in gcc, the gain in performance of certain applications is quite significant.

Minor improvements to the current implementation are planned:

- At the present, all the histograms have the same fixed range. We could use this range as a minimum range, and have the compiler to scan the case labels and enlarge the range if necessary. We could also provide command line parameters to override the minimum range values, and establish maximum range values.
- The default value promotion heuristic used in this implementation was a fixed 10% threshold. There should be a set of param-

eters to change this to some other value, or to some other heuristic.

Another idea, also related to switch statement optimization, is to explore the application of feedback directed information on the register allocator. In the perl benchmark, there is one long case statement that causes a large amount of register pressure, but is seldom executed. That could cause spill to be inserted elsewhere, perhaps into the main path of execution. The switch statement feedback information could be used to guide the register allocator to spill registers inside the seldom executed path instead.

5 Acknowledgments

I would like to thank Kate Stewart for her unlimited support during all the stages of this project. Without her, this work would still be forgotten in some cabinet.

I also want to thank Kristi Morton for her helpful comments and encouragement, Jan Hubicka for his candid feedback on the first gcc patch, and Freescale for making this work possible.

References

[TPB] FSF, `http://gcc.gnu.org/projects/tree-profiling.html`

[YDL] Terra Soft Solutions, *Yellow Dog Linux Version 4.0 for PowerPC*, 2005.

[YHPC] Terra Soft Solutions, *Y-HPC User's Manual*, January 13, 2005.

[XLC] Absoft, *XL C/C++ IBM Compiler for Y-HPC Linux 7.0*, Reference XLC4CSS70, 2005.