Appeared in Information Processing Letters 60, 3 (Nov. 11, 1996), 115-120.

Efficient Multiway Radix Search Trees

Úlfar Erlingsson [a], Mukkai Krishnamoorthy [a], T. V. Raman [b]

[a] *Rensselaer Polytechnic Institute, Troy, NY 12180*

[b] *Advanced Technology Group, Adobe Systems, Mountain View, CA 94039*

We present a new scheme for building static search trees, using multiway radix search. We apply this method to the problem of code generation for switch statements in imperative languages. For sparse case sets, the method has an advantage over existing methods, empirically requiring fewer than three branches for the average search. We give timing results that show that in practice our method runs faster than other methods on large sparse case sets.

Key words: Algorithms, Compilers, Switch statements, Code generation, Code optimization.

1 Introduction

Switch statements in C, like case statements in Pascal and Ada, are useful conditional control constructs. Switch statements represent multiway branching control structures, whereas if statements correspond to binary branching control.

We present a new code generation method for switch statements that on the average generates faster code than existing methods for sparse case sets, i.e., where the cases are widely separated in numeric magnitude. The method can be thought of as generating a Multiway Radix Search Tree (MRST) on the case labels. Although the MRST method is appropriate for most searches where a static search tree can be generated, this paper discusses only its application to switch statements.

There has been considerable work in the past ([2], [3], [5], [6] and [10]) on the Pascal case statement and code generation. The generation of code for switch statements is discussed in [4] and [11]. A scheme similar to MRST, but restricted to binary radix search trees, appears in [9].

Applications for fast sparse switch statements are many and varied. Two examples are:

– Let L be a Common-Lisp-like language with dynamic type dispatch on function arguments. Let F be an n argument generic function in L, with m methods defined on it. Further assume that there are T distinct types in the type-pool of L, each with a unique K-bit identifier. The dispatch of the generic function F is then a sparse switch statement that, for a given input from the T^n possible cases, checks whether one of the m methods applies.
– Consider the problem of searching for any of m substrings of length L in a string S. Assume a hash value is calculated for each of the m substrings, using a special hash function. Now the search can be accomplished in a single scan of S by calculating hash values for each set of L adjacent letters in S, as is done in the Karp-Rabin algorithm [7]. After each hash value is calculated for L adjacent letters from S, it must be compared against the m pre-computed hash values. This is effectively a sparse switch statement.

The remainder of this paper is structured as follows: Section 2 describes the MRST method for code generation for switch statements. Section 3 compares it to four other code generation methods with regard to worst-case and expected time complexity. Finally section 4 presents a comparative study of the empirical running time of MRST, comparing it against the fastest pre-existing method.

2 A New Code Generation Method for Switch Statements

We assume that all case labels, or *cases*, in the switch statement are numeric. Our approach is to determine a critical bit sequence for the input set of cases and hash on it. We do this recursively on non-trivial subsets of cases determined by the hashing. The resulting Multiway Radix Search Tree (MRST) is traversed at run time with the input value to the switch statement.

Case	Binary
0	$0\,00000\,00_2$
1	$0\,00000\,01_2$
129	$1\,00000\,01_2$
131	$1\,00000\,11_2$

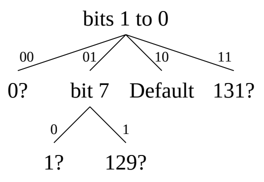

Fig. 1. A set of cases and the corresponding MRST.

Fig. 1 shows an example. The 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 jumps directly to the default handler for any run-time input ending in 10_2, since no case ends in this bit pattern.

The code generation 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 handler; for sets containing one case, to a 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, we must limit the length of the windows, since the branches require hash tables that grow exponentially in the window length. We therefore use 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.

2.1 Definitions

Let Z be the set of all K-bit values, where K is a positive integer. Let $C \subset Z$ be a set of cases, M a set of labels or markers, and m_d an additional default label. Assume as input a set $P = \{ (c_i, m_i) \}$ of ordered pairs, such that every $c_i \in C$ is associated with a single label $m_i \in M$. Thus P defines a total function from C to M.

We want to generate code that performs a mapping $\mathcal{F} : Z \rightarrow M \cup \{ m_d \}$ such that an input value $z \in Z$ is mapped onto m_d if $z \notin C$, otherwise, if $z = c_i$ for some $c_i \in C$, onto the label $m_i \in M$ as defined by P. Thus $\mathcal{F}$ performs the function we expect from a switch statement.

Denote the bits of K-bit values by $b_{K-1}, \ldots, b_0$ and define a *window* W to be a sequence of consecutive bits $b_l, \ldots, b_r$, where $K > l \geq r \geq 0$. Let $\text{val}(s, W)$ be the value of the bits of s visible in the window W. Thus, if $W = b_5, \ldots, b_3$, $\text{val}(41, W) = \text{val}(101001_2, W) = 101_2 = 5$.

A window $W = b_l, \ldots, b_r$ is *critical* on a subset S of Z if the cardinality of the set $V_W = \{ \text{val}(s, W) \mid s \in S \}$ is greater than 2^{l-r}. Thus, W is critical if its hash table, of size 2^{l-r+1}, is more than half full. A window W is *most critical* if $|V_W| \geq |V_{W'}|$ for all equally long windows W'.

The initial input to the algorithm is the set $P = \{ (c_i, m_i) \}$ defined above, along with the default label m_d. The output is code that maps a run-time input value $z \in Z$ onto a label $m \in M \cup \{ m_d \}$ according to mapping $\mathcal{F}$.

```
algorithm MRST( $m_d$, $P$ )
  1.  if $P$ contains only one pair $(c, m)$ then
        Generate a jump to $m$ if $z = c$, but a jump to $m_d$ otherwise.
        return.
  2.  Let $C$ be the set $\{ c_i \mid (c_i, m_i) \in P \}$.
      Find $W_{\max}$, the longest and most critical window $b_l, \ldots, b_r$ on $C$.
      Let $n$ be $l - r + 1$, the length of $W_{\max}$.
  3.  Generate an assignment of val$(z, W_{\max})$ to a register $r1$.
      Generate a jump to the label indexed by $r1$ in the table of step 4.
  4.  for $j := 0$ to $2^n - 1$ do
        Create a new entry label $t_j$.
        Create a set $P_j$ of all pairs $(c_i, m_i) \in P$ such that val$(c_i, W_{\max}) = j$.
        if $P_j$ is not empty then Generate $t_j$ as a table entry.
        else Generate $m_d$ as a table entry.
  5.  for each $P_j$ that is not empty do
        Generate the entry label $t_j$.
        call MRST( $m_d$, $P_j$ ).
  return.
```

Finding the longest and most critical window in step two of the algorithm can be achieved with the following simple function. The function makes use of the fact that a prefix of a critical window is also critical to accomplish its task in a single scan. Since any two distinct numbers will differ in at least one bit, the function will always find a window $W_{\max}$ of length at least one.

```
function CriticalWindow( $C$ )
  Let $W$ and $W_{\max}$ be windows, initially empty.
  for $b :=$ bit $b_{K-1}$ to bit $b_0$, one at a time do
    Add $b$ to $W$, extending $W$ one bit to the right.
    if $W$ is critical on $C$ then Assign $W$ to $W_{\max}$.
    else (if $W$ is not critical on $C$)
      Modify $W$ by removing its leftmost bit.
      if $|\{ \text{val}(c_i, W) \mid c_i \in C \}| > |\{ \text{val}(c_i, W_{\max}) \mid c_i \in C \}|$ then
        Assign $W$ to $W_{\max}$.
  return $W_{\max}$.
```

The invariants of the above loop are that the lengths of W and $W_{\max}$ are equal, and that $W_{\max}$ is the longest and most critical window in $b_{K-1}, \ldots, b$.

At every iteration of the loop, window W is extended one bit to the right. If this makes W critical it is stored in $W_{\max}$. Otherwise, the leftmost bit of W is removed, and if the new W has higher cardinality than $W_{\max}$ it is stored in $W_{\max}$. Thus $W_{\max}$ remains the longest critical window and the most critical window of its length.

2.3 Sample Code Generation

Fig. 2 contains the search tree for the following Pascal-like example. Assembly code generated for the switch statement by the MRST algorithm is shown in Fig. 3. The input value is assumed to be found in a register z, and a register $r1$ indexes into the hash tables.

```
Case z Of
   8, 16, 33, 37, 41, 60: Function1;
   144, 264, 291:         Function2;
   1032:                  Function3;
   2048, 2082:            Function4;
   Otherwise              Default;
End;
```

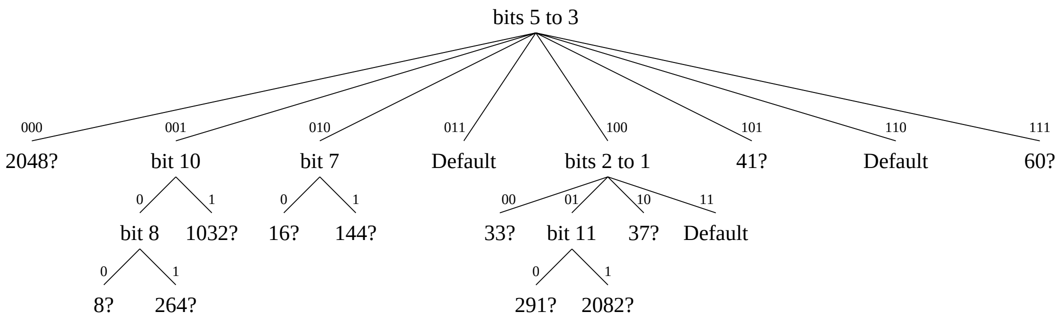

Fig. 2. An MRST for the example switch statement.

3 Analysis of Methods

There are several established methods of generating code for switch statements, apart from MRST. We now compare their worst case and expected time complexity, assuming that we are running a switch statement with m random cases, and that the statement is called n times with inputs from the set of cases. We use the number of branches as the metric for our time complexity measures.

Skewed Binary Tree [1], [10]. This method is essentially a linear search. Therefore it takes $O(mn)$ time.

Balanced Binary Tree [10], [11]. The run-time input is compared with one case at a time in such a way that the search tree is balanced. Hence the

```
start:
   mov  r1, z
   shr  r1, 3
   and  r1, 0007h
   jmp  [table1+r1]
table1: data case_000
        data case_001
        data case_010
        data Default
        data case_100
        data case_101
        data Default
        data case_111
case_000:
   cmp  z, 2048
   jeq  Function4
   jmp  Default
case_001:
   mov  r1, z
   shr  r1, 10
   and  r1, 0001h
   jmp  [table2+r1]
table2: data case_001_0
        data case_001_1
case_001_0:
   mov  r1, z
   shr  r1, 8
   and  r1, 0001h
   jmp  [table3+r1]
table3: data case_001_0_0
        data case_001_1_1
case_001_0_0:
   cmp  z, 8
   jeq  Function1
   jmp  Default
case_001_0_1:
   cmp  z, 264
   jeq  Function2
   jmp  Default
case_001_1:
   cmp  z, 1032
   jeq  Function3
   jmp  Default
case_010:
   mov  r1, z
   shr  r1, 7
   and  r1, 0001h
   jmp  [table4+r1]
table4: data case_010_0
        data case_010_1
case_010_0:
   cmp  z, 16
   jeq  Function1
   jmp  Default
case_010_1:
   cmp  z, 144
   jeq  Function2
   jmp  Default
case_100:
   mov  r1, z
   shr  r1, 1
   and  r1, 0003h
   jmp  [table5+r1]
table5: data case_100_00
        data case_100_01
        data case_100_10
        data Default
case_100_00:
   cmp  z, 33
   jeq  Function1
   jmp  Default
case_100_01:
   mov  r1, z
   shr  r1, 11
   and  r1, 0001h
   jmp  [table6+r1]
table6: data case_100_01_0
        data case_100_01_1
case_100_01_0:
   cmp  z, 291
   jeq  Function2
   jmp  Default
case_100_01_1:
   cmp  z, 2082
   jeq  Function4
   jmp  Default
case_100_10:
   cmp  z, 37
   jeq  Function1
   jmp  Default
case_101:
   cmp  z, 41
   jeq  Function1
   jmp  Default
case_111:
   cmp  z, 60
   jeq  Function1
   jmp  Default
```

Fig. 3. Assembly code generated by the MRST algorithm for the example.

running time is $O(n \log m)$.

Balanced Binary Tree to Hash Tables [3], [4], [5], [6]. This is a balanced binary search to ranges where the case set is dense and then a direct indexing on those ranges. Let the number of leaf nodes with hash tables be r. The worst-case running time occurs when $r = m$ and is $O(n \log m)$, since $\log m$ is the search time for the tree. The expected case running time is $O(n \log r)$.

Jump Table Method [1], [10]. This is a direct hash table lookup and has a running time of $O(n)$ in all cases. It requires space linear in the *range* of input cases and is therefore impractical for sparse sets of input cases. Even so, this is usually the optimal method for dense case sets.

The analysis of the MRST method is somewhat different from that of the methods above, since the tree is no longer balanced or binary. The worst case for MRST occurs when the set of cases is the distinct powers of two. Then the MRST is skewed with a maximum depth of $K/2$, where K is the architecture word size. Hence the worst-case running time is $O(nK)$. This is still linear in n, albeit with a large constant.

We can find the expected-case behaviour of MRST under the assumption that the m input cases are random equi-distributed K-bit numbers. Given a window W of length L, the probability that there are r distinct values $\mathrm{val}(c_i, W)$, where c_i is in the set of m input cases and $0 < r < 2^L$, is given by the following (see [8], page 62):

$$P(r, m, L) = \frac{2^L!}{(2^L - r)!\, 2^{mL}} \left\{ {m \atop r} \right\}.$$

Thus, if we sample m elements with replacement from a random source of values in the range $[0; 2^L)$, $P(r, m, L)$ is the probability that the sample will contain r distinct different values. Given this we can define the probability that W is critical as

$$P_{critical}(m, L) = \sum_{r=2^{L-1}+1}^{2^L} P(r, m, L).$$

When m becomes large (e.g., > 10) $P_{critical}(m, L)$ approaches unity for $L = \lfloor \lg m \rfloor$. We therefore expect the top-level multiway branch of a MRST to have a fan out of order m, with $m/2$ of the branches leading to non-trivial case sets, and each of these sets containing roughly two cases. Thus the average number of branches in an MRST with a uniformly random set of cases is less than three: the top-level branch, plus an average of one and one-half branches for the resulting case subsets. Our experimental results in the next section confirm this expectation.

4 Empirical Timing Results

We integrated MRST into the lcc C compiler [4]. The lcc compiler has a built-in implementation of the balanced binary tree to hash table (BBTH) method for switch statements.

We performed timing experiments on two representative platforms: the CISC architecture Intel 486 and the recent RISC architecture UltraSPARC. The relative performance of the two methods was similar on both platforms, so we only show the UltraSPARC results.

We examined the empirical performance of the two methods on several representative input case sets. We considered switch statements with m distinct cases and with r dense ranges, where a range could contain anywhere from 1 to 20 cases. We performed our experiments on randomly-generated sets of input cases, taken from the full range of 32-bit numbers. In Figs. 4 and 5 we show the average number of branches, along with storage space and running time resulting from our experiments. The values shown are averages taken from 100 runs. In general our results have very little deviation from these averages. As can be seen, MRST has far fewer branches, and runs faster than BBTH.

When performing our empirical timing experiments, we resolved the following two issues to MRST's disadvantage:

– The run-time input iterated through the members of the case set. If the input had not been restricted to the set of cases, MRST would have had an

	Branches		Space		Time	
m	BBTH	MRST	BBTH	MRST	BBTH	MRST
10	3.58	2.05	10.00	25.92	19.01	19.69
100	6.67	2.29	100.00	228.64	32.06	20.39
1000	9.97	2.44	1000.00	2043.84	55.56	25.57

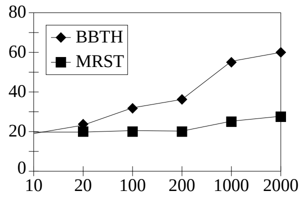

Fig. 4. Empirical results for m random cases (time graphed).

	Branches		Space		Time	
r	BBTH	MRST	BBTH	MRST	BBTH	MRST
10	3.58	2.43	105.64	222.20	25.53	20.95
100	6.67	2.49	1049.59	2103.67	38.16	25.81
1000	9.97	2.59	10519.50	19343.70	54.50	34.86

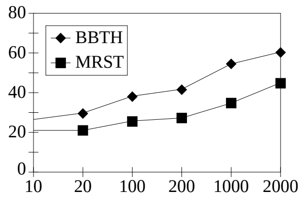

Fig. 5. Empirical results for r random dense ranges of cases (time graphed).

advantage, since early branchings to the default handler would then have provided speedup.
- The generated code for the switch statements was placed inside a tight inner loop and therefore could make full use of the instruction cache. If the code had not been in the cache, MRST again would have had an advantage, since the shallowness of the generated tree means that very few cache lines have to be brought in from memory.

We also ran a set of tests in which the cache was flushed clean after every iteration and where the run-time input was not restricted to the case set. The results from those experiments were as expected, viz., MRST was even faster relative to BBTH.

We also timed and compared the two methods using a set of 1000 switch statements found in the source code of the lcc compiler, the GNU gcc compiler and the GNU GhostScript PostScript interpreter. The two methods were on the average almost equally fast on these case sets. The total running time for one million iterations of the 1000 switch statements differed by less than 5% for the two methods, with MRST being slightly faster.

This small difference in running time is due to the fact that our example set included only "hand-coded" switch statements. The statements therefore tended to be very small on average and the constants had usually been chosen to form dense ranges. Under these circumstances we do not expect MRST to perform any better than BBTH.

Even so, the above timing results show that MRST can be significantly faster than other methods and is rarely, if ever, slower for any but the smallest case sets. MRST is especially appropriate for large and sparse sets of cases, such as may be automatically generated in modern compilers and operating systems.

Acknowledgments MRST originated in the first author's undergraduate studies at the University of Iceland. Snorri Agnarsson and Andrew Shapira contributed significantly to the development of the method. The authors would like to thank Ed Green, David Gries, Snorri Gylfason, Andrew Shapira, David L. Spooner, Kendra Willson and the anonymous reviewer for their help in revising this paper.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1988.

[2] L. Atkinson. Optimizing two-state case statements in Pascal. *Software—Practice and Experience*, 12:571–581, 1982.

[3] R. Bernstein. Producing good code for the case statement. *Software—Practice and Experience*, 15:1021–1024, 1985.

[4] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1994.

[5] J. Hennessy and N. Mendelsohn. Compilation of the Pascal case statement. *Software—Practice and Experience*, 12:879–882, 1982.

[6] S. Kannan and T. Proebsting. Correction to Producing good code for the case statement. *Software—Practice and Experience*, 24:233, 1994.

[7] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. Technical report 31-81, Aiken Compu. Lab., Harvard University, Cambridge, MA, 1981.

[8] D. E. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms.* Addison-Wesley, 2 edition, 1981.

[9] D. R. Morrison. PATRICIA—Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.

[10] A. Sale. The implementation of case statements in Pascal. *Software—Practice and Experience*, 11, 1981.

[11] R. M. Stallman. Using and porting GNU CC. Technical report, Free Software Foundation, 1992.