Series Logo
Volume 00, Number 00, Xxxx 19xx

LUCID AND EFFICIENT CASE ANALYSIS

ÚLFAR ERLINGSSON, MUKKAI KRISHNAMOORTHY, AND T. V. RAMAN

ABSTRACT. This paper describes a new scheme for building static search trees, using multiway radix search trees. We present this method for code generation of switch statements in imperative languages. We show that, for sparse case sets, the method produces faster code on average than existing methods, requiring $O(1)$ time with a small constant for the average search. We then apply this method to the problem of code generation for generic functions in object-oriented languages, and find that its use improves clarity as well as efficiency.

Keywords Algorithms, Compilers, Switch statements, Code generation, Code optimization, Object-oriented methods.

1. Introduction

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

In this paper we present a new code generation method for switch statements, which on the average generates faster code for sparse sets of cases than existing methods, and apply it to generic function dispatch in object-oriented languages. The method can be thought of as generating a multiway radix search tree on the case labels, and is referred to as the MRST method throughout this paper. Although the MRST method is appropriate for most searches where a static search tree can be generated, only its application to switch statements is considered here.

There has been considerable work in the past ([8], [5], [2], [3] and [6]) regarding the Pascal case statement and code generation. Generating code for switch statements is discussed in [4] and [9]. A scheme similar to MRST, but restricted to binary radix search trees, can be found in [7].

We begin, in section 2, by describing the MRST method of code generation for switch statements. In section 3, we compare it with four other code generation methods. We analyze the methods in terms of storage requirements and time, providing the worst case and expected case scenarios for each method. We then perform an empirical study of the MRST method with a few representative examples. Finally, in section 5, we present an implementation mechanism for generic functions in object-oriented languages which exploits the properties of our MRST 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 basic 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 sets of cases. The resulting MRST (Multiway Radix Search Tree) is traversed at run-time with the input value to the switch statement.

Case	Binary
0	$0\,0000000_2$
1	$0\,0000001_2$
129	$1\,0000001_2$
131	$1\,0000011_2$

FIGURE 1. A set of cases and their MRST.

In figure 1 we see an example justifying our approach. The cases 0, 1, 129 and 131 are almost uniquely determined by the value of their last two bits. The 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 run-time input by hashing first on the last two bits and then, if necessary, on bit 7. Finally, we compare the input value with the unique remaining candidate case. Note that we jump 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 which distinguishes the cases into several subsets. The algorithm generates code which 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 which grow exponentially in the window length. We therefore use the simple greedy strategy of finding the longest *critical* window which 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 be a set of labels or markers, and m_d be a default label. We 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 $m_i \in M$.

We want to generate code which performs a mapping $\mathcal{F} : Z \to M \cup \{\, m_d \,\}$ 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. The mapping $\mathcal{F}$ thus performs the function we expect from a switch statement.

We denote the bits of our K-bit values by $b_{K-1}, \ldots, b_0$. We define a *window* W to be a sequence of consecutive bits $b_l, \ldots, b_r$ where $K > l \geq r \geq 0$. We also define a special function *val* on windows W and integers $j \in Z$ such that $\text{val}(j, W)$ is the value of the bits of j visible in the window W. Thus if $W = b_5, \ldots, b_3$ we have that $\text{val}(41, W) = \text{val}(101001_2, W) = 101_2 = 5$.

A window $W = b_l, \ldots, b_r$ is *critical* on a subset S of Z if the cardinality of the set $\{\, \mathrm{val}(s, W) \mid s \in S \,\}$ is greater than 2^{l-r}. Thus W is critical if a hash table of size 2^{l-r+1} is more than half full when we hash on it with the set $\{\, \mathrm{val}(s, W) \mid s \in S \,\}$.

2.2. **The MRST Code Generation Algorithm.**

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

```
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 critical window $b_l, \ldots, b_r$ on $C$ (see steps 2.1–2.4).
    Let $n$ be $l - r + 1$, the length of $W_{\max}$.
3.  Generate an assignment of $\mathrm{val}(z, W_{\max})$ 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 $\mathrm{val}(c_i, W_{\max}) = j$.
      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 $P_j$ which is not empty Do
      Generate the entry label $t_j$.
      Call MRST( $m_d$, $P_j$ ).
```

Step 2 of the algorithm can be performed with the following simple procedure. The effect of the procedure is to scan repeatedly over the bits with windows of increasing length, choosing the longest and most critical window as $W_{\max}$. The fact that the prefix of any critical window is also critical allows us to accomplish this in a single scan. Since any two distinct numbers will differ in at least one bit, we will always find a window W of length at least one.

```
2.1. Let $C$ be the set $\{\, c_i \mid (c_i, m_i) \in P \,\}$.
     Let $W$ and $W_{\max}$ be windows, initially empty.
     Let $\#W$, $\#W_{\max}$, $k$ and $n$ be integers, initially zero.
2.2. 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.
2.3.   Let $k$ be the length of $W$.
       Let $\#W$ be the cardinality of $\{\, \mathrm{val}(c_i, W) \mid c_i \in C \,\}$.
       If $W$ is critical on $C$ Then
         If $k > n$ or $\#W > \#W_{\max}$ Then
           Let $W_{\max}$ become $W$, $\#W_{\max}$ become $\#W$ and $n$ become $k$.
2.4.   Else (if $W$ is non-critical on $C$)
         Remove the leftmost bit from $W$.
         Perform step 2.3 once.
```

2.3. **Potential Optimizations.** Implementations of the algorithm could adopt various strategies for optimizing running time or storage space. Simple optimizations, such as replacing a hash table of size two with a bit test, will provide some speedup. However, the major factor affecting performance is the choice of windows to hash on. By changing the definition of a critical window, so the threshold ratio is not $1/2$ but an arbitrary fraction in $(0, 1]$, storage space and running time can be adjusted according to implementation priorities.

The search tree can also be made shallower by removing the constraint that windows need to be sequences of consecutive bits, and generating additional code which compacts sparse windows. Using that method, we could generate code for the cases in figure 1 which branched directly through a hash table of size 8. Several other optimization schemes are feasible. For instance constructing the MRST using a radix other than 2 may provide speedup for some case sets.

2.4. **Sample Code Generation.** We now show an example of code generated by the MRST algorithm. Below is a small switch statement written in a Pascal-like language. In figure 2 we can see the search tree generated by our algorithm for this switch statement.

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

FIGURE 2. A MRST for the switch statement.

Assembly code generated for the switch statement by our MRST algorithm is shown in figure 3. The input value is assumed to be in a register z and we use a register $r1$ to index into the hash tables.

3. Analysis of Methods

There are several different methods of generating code for switch statements, apart from our MRST. Some common ones are:

1. Skewed Binary Tree (see [1] and [8])
2. Balanced Binary Tree (see [8] and [9])
3. Balanced Binary Tree to Hash Tables (see [3], [6], [5] and [4])
4. Jump Table Method (see [1] and [8])

```
start:
   mov r1, z
   and r1, 0038h
   shr r1, 3
   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
   and r1, 0400h
   shr r1, 10
   jmp [table2+r1]
table2: data case_001_0
        data case_001_1
case_001_0:
   mov r1, z
   and r1, 0100h
   shr r1, 8
   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
   and r1, 0080h
   shr r1, 7
   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
   and r1, 0006h
   shr r1, 1
   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
   and r1, 0800h
   shr r1, 11
   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
```

FIGURE 3. Assembly code generated by the MRST algorithm.

We now compare the time complexity of the above methods with that of our MRST method. We assume that we are running a switch statement that has m cases, and that the statement is called n times with inputs from the set of cases. We show both worst case and expected case running times.

Skewed Binary Tree: This is essentially a linear search. Therefore it takes $O(mn)$ time in all cases.

Balanced Binary Tree: We compare the run-time input with one case at a time in such a way that the search tree is balanced. Hence the running time is $O(n \log m)$ in all cases.

Balanced Binary Tree to Hash Tables: We do a balanced binary search to ranges where the case set is dense, and then hash directly 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: This is a direct hash table lookup, giving us a running time of $O(n)$ in all cases. However, the method 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.

MRST Method: Here the analysis is slightly different, as the tree is no longer balanced or binary. In the worst case, the MRST is skewed with a maximum depth $K/2$, where K is the word size of the machine. Hence the worst case running time is $O(nK)$. However, in the expected case, the search time for an entry is $\log_h(K)$, where h is the expected size of the hash tables. Empirically, we can expect h to be on the order of K, as is shown in the next subsection. Therefore the expected case running time is $O(n)$.

In the first three methods, the internal nodes in the search tree are comparison instructions, while in our MRST method they are hash table lookups. Even though this is likely to increase the size and storage space requirements of the generated code, it is also likely to improve its running time. The results become even more favorable if we do not restrict the run-time input to the set of cases, since early branches to the default handler then improve running time. This justifies our MRST approach.

3.1. **Empirical Analysis.** We now examine the empirical performance of the MRST method on several representative input case sets. We consider both switch statements with m distinct cases and switch statements with r dense ranges, where a range can contain anywhere from 1 to 20 cases. In figures 4a and 4b we show the average number of branches and storage space for different values of m and r respectively. Storage space is the number of hash table entries required. All values shown are averages from 100 runs on randomly generated sets of input cases, taken from the full range of 32-bit numbers. Our empirical results have very little deviation from the averages shown.

m	Branches	Space
10	2.13	17.1
20	2.23	35.8
100	2.45	175.9
200	2.48	359.0

(a)

r	Branches	Space
10	2.59	183.9
20	2.57	363.3
100	2.64	1769.0
200	2.64	3548.9

(b)

Ex.	Cases	Branches	Space
1	6	2.57	12
2	128	2.00	128
3	15	5.50	30
4	2	2.00	2

(c)

FIGURE 4. Performance of the MRST method on several input case sets

We also consider four other examples, the results of which are shown in figure 4c. The first example converts numbers into Roman numerals. The second arises in the simulation of a finite-state automaton with transition labels in the ASCII character set. The third uses cases that are the distinct powers of two from 2^1 to 2^{15}, and thus has the worst case expected running time for our method. The fourth and final example has only two cases, 2^{14} and -2^{14}.

Obviously, we need to branch at least once in any implementation of a switch statement. We can attain this minimum by using the jump table method, but this is unfeasible for sparse sets of cases. Our MRST method, on the average, requires only two to three branches, using storage space linear in the number of input cases.

4. A Weak Worst-Case Upper Bound on MRST Storage Space

We proceed by induction on the height of the generated MRST. We assume the threshold ratio is a fraction $d \in (0, 1]$, with $D = d^{-1}$. We show that the storage space required for m distinct cases in an MRST of height h is less than $(4/3)^{h-1}Dm$. From the MRST algorithm this is trivially true for an MRST of height one, i.e., with only one hash table, we have $dS_m < m$ and thus $S_m < Dm$.

We now assume our hypothesis holds for MRSTs of heights 1 to $h-1$ and show it holds for MRSTs of height h. We also assume we have a window which distinguishes m input cases into s subsets, with a subsets of cardinality one or two and $b = s - a$ of greater cardinality. There is no extra storage space required for the a small subsets. For each of the larger subsets, however, the space required is less than $(4/3)^{h-2}D$ times the cardinality, and therefore in total $(4/3)^{h-2}D(m-a)$ for the b subsets. The space required for the root level hash table is less than $D(a+b)$.

Therefore the total space is bounded above by $(4/3)^{h-2}D(m-a+a+b)$, which we can simplify to $(4/3)^{h-1}Dm$.

5. A Practical Application Of Switch Optimization

The previous sections outlined and analyzed our MRST techniques for optimizing sparse switch statements in imperative programming languages. We will now look at how we can implement generic function dispatch for object-oriented languages efficiently using switch statements and our MRST method.

Performing by hand the case analysis required for a switch statement based implementation of generic function dispatch is inflexible and error-prone. However, switch statements automatically generated for this purpose are likely to be very sparse. By combining the abstraction mechanisms of object-oriented programming with our MRST method we get an implementation with the desirable qualities of being lucid and clear, while generating near-optimal compiled code.

5.1. Object-oriented Generic Functions.

Let L denote an object-oriented language with single-inheritance. Consider implementing function A, a function of one argument that takes a geometric shape S (e.g., square, triangle or circle) and computes its area. Using conventional case analysis techniques, function A would be implemented as a long if statement, with one conditional clause for each geometric shape that is handled by function A.

The disadvantages of the above are immediately obvious. Extending A to handle new shapes is hard, since it needs to be modified each time and will quickly become unwieldy. Also, as the number of clauses in A increases, the function becomes inefficient. Given a shape S, the implementation simply performs a linear scan of the clauses making up A.

The first of the above disadvantages is, in fact, often used to justify object-oriented programming. Here, A can be implemented as a *generic function*[1]. Extending the implementation of A to handle new shapes only requires the definition of new *methods* on A. Each clause of the non-object-oriented implementation thus becomes a single method, making the semantics of A clear.

5.2. General Case Generic Dispatch.

Overloading of functions and generic dispatch based on argument types are two of the most powerful features of any object-oriented language. We now show how we can implement these features efficiently using switch statements and the MRST method.

Consider a generic function F of n arguments in language 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 execution of the generic function F is, in fact, a switch statement which, for a given input case from the T^n possible cases, checks whether one of the m methods applies.

We can automatically generate efficient code for this switch statement using the following approach. Assume that method i on F requires its jth argument to be of type t_{ij}. Method i is thus executed with run-time arguments $a_1, \ldots, a_n$ if the condition $(\text{type}(a_1) = t_{i1}) \wedge \cdots \wedge (\text{type}(a_n) = t_{in})$ is true. This condition is represented by one nK-bit case in the switch statement, formed by concatenating the K-bit type identifiers of $t_{i1}, \ldots, t_{in}$. At compile-time, we generate executable code by running the MRST algorithm on resulting set of m case/method pairs. At

[1]We use the Common Lisp Object System terminology.

run-time, we concatenate the run-time argument type identifiers into an nK-bit case, and use this case as the input to the switch statement.

It should be pointed out that, in any practical application where such case analysis is performed, the resulting set of cases is very sparse, i.e., m is extremely small relative to 2^{nK}. Since the MRST method is the only method likely to generate an efficient search for such sparse case sets, it is very attractive for this application.

5.3. **Using Interpreted Languages.** Finally, we would like to point out that these techniques apply equally well in the realm of interpreted scripting languages. If the language preprocesses the script into an intermediary representation, a static scheme such as the MRST method can be used. If not, the following approach may be applicable.

TCL is an interpreted scripting language where the primary datatype is the string. A TCL procedure that contains an n-way switch statement can become inefficient as n gets large. However, we can provide an efficient and lucid implementation of generic functions in TCL. We first define a table, implemented as a TCL associative array. We index the table by the result of the various tests equivalent to the cases of the switch statement. The entry in the table is the name of a TCL procedure that executes the corresponding branch of the switch statement. Since associative arrays in TCL are implemented as hash tables, the resulting encoding will exhibit efficiency comparable to that of the MRST method.

6. Conclusion

We have presented the MRST method, a new method of generating static search trees on numeric case sets. We have demonstrated the method in the context of generating code for switch statements in imperative languages. There has been considerable previous work on this code generation. We have found that, for sparse case sets, our MRST method generates code which is empirically faster than previous methods. For most non-contrived case sets, the expected number of branches is less than three. Finally, we have applied the MRST method to generic dispatch in object-oriented languages, and found that its use makes the required case analysis lucid as well as efficient.

7. Acknowledgments

The MRST method originated in work done 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, Snorri Gylfason, Andrew Shapira, David L. Spooner and Kendra Willson for their help in revising this paper.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[2] L. Atkinson, "Optimizing Two-state Case Statements in Pascal," *Software—Practice and Experience*, Vol. 12, 1982, pp. 571–581.

[3] R. Bernstein, "Producing Good Code for the Case Statement," *Software—Practice and Experience*, Vol. 15, 1985, pp. 1021–1024.

[4] C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, The Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1994.

[5] J. Hennessy and N. Mendelsohn, "Compilation of the Pascal Case Statement," *Software—Practice and Experience*, Vol. 12, 1982, pp. 879–882.

[6] S. Kannan and T. Proebsting, "Correction to Producing Good Code for the Case Statement," *Software—Practice and Experience*, Vol. 24, 1994, pp. 233.

[7] D. R. Morrison, "PATRICIA—Practical Algorithm To Retrieve Information Coded In Alphanumeric," *Journal of the Association for Computing Machinery*, Vol. 15, 1968, pp. 514–534.

[8] A. Sale, "The Implementation of Case Statements in Pascal," *Software—Practice and Experience*, Vol. 11, 1981, pp. 929–942.

[9] R. M. Stallman, "Using and Porting GNU CC," Technical Report, Free Software Foundation, Cambridge, MA, 1992.

(Erlingsson) Rensselaer Polytechnic Institute, Troy, NY 12180

(Krishnamoorthy) Rensselaer Polytechnic Institute, Troy, NY 12180

(Raman) Cambridge Research Laboratory, Digital Equipment Corp., Cambridge, MA 02139

E-mail address, Erlingsson: `ulfar@cs.rpi.edu`
E-mail address, Krishnamoorthy: `moorthy@cs.rpi.edu`
E-mail address, Raman: `raman@crl.dec.com`