SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(10), 1021–1024 (OCTOBER 1985)

Short Communication

PRODUCING GOOD CODE FOR THE CASE STATEMENT

ROBERT L. BERNSTEIN

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, U.S.A.

SUMMARY

After summarizing different ways of generating selection code for the case statement, the execution speed of each type of selection code is described using the RISC machine model. A method for combining the selection code schemes is proposed. The method, part of the PL.8 compiler, is flexible enough to handle case-selector data-types that have a large range of values, such as the character-string or floating-point datatypes, and can be fine-tuned when the probabilities of the case selector taking on particular values are known.

KEY WORDS Case statement Code generation

INTRODUCTION

A *case statement* (also known as a *select statement* or *switch statement*) tests a variable, the *case selector*, against a set of specified constants called *case items*. Non-empty sets of case items may be grouped together into *case clauses* so that the same action is performed if the case selector has the value of any case item in the case clause. If the case selector is not equal to any case item, execution continues at a place specified by an *otherwise clause*. If there is no otherwise clause in the case statement, execution terminates abnormally; a *trap* occurs. The model for the case statement that is used is similar to Ada's[1] with the exception that the case selector and case items may be floating point or string values. The *case density*, which is useful in deciding what kind of code to generate, is a real number greater than 0 and less than or equal to 1. It is the number of values between the highest and lowest valued case items inclusive, divided by the number of case items. Rather than use the case density, the integer approximation of the inverse of the case density can be used just as easily to avoid floating-point operations.

For machine instructions, a model of a reduced-instruction-set computer (RISC), such as the IBM 801[2] having many internal registers for saving computations, and executing all its instructions in roughly the same cycle time, is used. As a rule of thumb, the number of 801 instructions required for an instruction of a conventional machine is the execution time of the complex instruction divided by the execution time of a register-to-register 'add' on the same machine. Thus, some instructions on conventional computers require 2 or 3 RISC instructions. In particular, on an 801 there are no instructions with 'indirect' operands other than 'load' or 'store' instructions; to 'jump indirect', one instruction loads the desired address into a register and then another performs the jump. If instructions are put in their 'reduced' form, the execution time is proportional to the number of instructions, allowing us to blur the distinction between the number of instructions and the time it takes to execute them. Therefore, remarks about instruction length should be viewed as remarks about the *speed* with which a construct can be performed. The relative speeds of constructs given should apply to certain conventional architectures, even though fewer instructions are coded.

In this paper, 'producing code for the case statement' refers only to the time the code needs to determine which alternative to select. Selection code can be carried out in a variety of ways. The most common implementation is a *jump table* (or *branch table*). In this implementation, a *range test* is performed; that is, the case selector is compared against some range of values. If the case selector is in

0038–0644/85/101021–04$01.00

Received 12 June 1984
Revised 16 November 1984 and
24 January 1985

this range, an address is computed and jumped to. The address can be computed in a variety of ways such as looking up a relative address in a table or jumping to a jump instruction contained in a table. Many other variations are used in practice. Another implementation is to compare the case selector against one of the (middle) case items and use the ordering of the comparison to restrict future comparisons to case items that are either higher or lower. If the case item is always the middle one of the remaining set of candidate case items, we get a *binary search*. (Note that this is a binary search to determine membership in a set of case items and not a binary search to determine the value of the case selector, which is already known.) Another possibility is a linear sequence of alternating 'compare' and 'branch equal' instructions, called a *linear search*. Finally, there is the special case, but one that occurs often enough, where some subset of case items in a case clause forms a range with no gaps (the case density is 1). Here a compiler need only test that the case selector is in the appropriate range, omitting the code to modify a jump or index into a jump table. Sale[3] describes these techniques in greater detail.

EFFICIENT RANGE TESTS

A straightforward way to carry out a range test is to perform the following four pseudo–801 instructions:

compare *case selector* with *lower bound*
if less goto *out of range*
compare *case selector* with *upper bound*
if greater goto *out of range*
assertion: *lower bound* ≤ *case selector* ≤ *upper bound*

But there is another way to do this test in two or three instructions on computers that allow unsigned or 'logical' (in IBM System/370 terminology) comparisons. For an unsigned comparison, the contents of a register is interpreted as a positive value in the range 0 to 2^n-1 where n is the word size, rather than say a two's complement value in the range -2^{n-1} to $+2^{n-1}-1$. The important point here is that negative numbers act like very large positive numbers. Most computers either have 'unsigned compare' or 'branch on unsigned condition' instructions. To test that a register is in a desired range, the following three instructions suffice:

k ← *case selector – lower bound*
compare *k* with (*upper bound – lower bound*)
  – –this is an integer constant
if unsigned greater goto *out of range*
assertion: *lower bound* ≤ *case selector* ≤ *upper bound*

If the lower bound is zero, the subtraction is unnecessary. To force a trap when the value is out of range, the 801 has a 'trap on condition' instruction, which can be used instead of the last branch. On many machines a trap can be forced by jumping to an illegal address. For example, on the IBM System/370 or Motorola MC68000 a branch to an odd address forces a trap.

DECIDING WHICH METHODS TO USE

As indicated by Sale, both binary search selection and jump table selection are asymptotically faster than linear search selection. However, it is as important to handle situations where the case statement has a few case items as it is to handle the situations where the case statement has an 'asymptotic' number of case items. (Atkinson[4] believes a two case-item case statement to be important enough to discuss how one might generate code for this specially.) Therefore, some care should be used to decide where a jump table, range test, binary search or linear search is appropriate.

The range test as previously given requires two or three 801 instructions. Another two or three 801 instructions load the jump address from the jump table and two instructions perform the jump indirect. So, 6–9 instructions select some alternative in a jump table. When there is no otherwise clause, all instructions are expected to be executed. The opposite is true for a linear or binary search where the maximum number of instructions get executed only when there is no equal case item. (It is perfectly reasonable to make the paths that lead to traps require more instructions, since these are the ones that would not get repeatedly executed.) If there are four scalar case items, the selection takes 2 instructions if the case selector is equal to the first case item, 4 instructions if equal to the second case item, ..., 8 instructions if equal to the fourth case item, and 9 instructions if the case selector is not equal to some case item. Assuming these events equally likely, fewer than 6 instructions are executed on the average. If some of the compare and branch instructions can be combined to form a range test, all the better. When there are four or fewer scalar case items, a linear search is faster than a jump table.

Binary searching performed as a sequence of three in-line 'compare', 'branch greater' and 'branch equal' instructions executes one more branch instruction than a linear search for each case item tested. Of course, there is never benefit in binary searching two case items. The average number of instructions executed in binary searching 4 case items is about 5. Thus, binary searching is faster than linear searching when there are at least 4 case items. When the comparison is more costly than one instruction, such as when floating-point or string comparisons are

required, or when a range test lumps together several case items, binary searching is preferred even if there are only 3 case items. It is better to put 'branch (not) greater' tests before the 'branch equal' tests when the probability that the case selector is (not) greater than the current case item is not less than the probability that the two are equal. If case alternatives are equally likely, this is usually true.

Although a jump table is the fastest method of selection when there are many case items, Hennessey and Mendelsohn[5] suggest splitting a single jump table into several smaller ones if the overall case density is low (say less than 1/2), as a space/time trade-off. Binary *or* linear searching is then used to select the appropriate jump table. Selecting the smallest number of clusters so that each cluster meets a minimum-density requirement is NP-complete. This is easily seen as this problem is a generalization of the clustering problem given as MS9 in Reference 6. In MS9, densities are restricted to integers, whereas our problem allows real-valued densities.

A natural heuristic for forming clusters is the greedy approach, which always splits at the next largest gap between sorted case items. For example, if the integer-valued case items in sorted order are (1, 2, 3, 5, 10), having case density 1/2, we would first split between 5 and 10 with gap 5 yielding densities 4/5 and 1 for the two sub-ranges (1, 2, 3, 5) and (10). If case density 4/5 is not deemed sufficiently dense, we would then split between 3 and 5 with gap 2. If a heap sort[7] is used to sort the case items initially, the 'extract maximum' function of the sorting routine can be reused to find the next largest gap.

ALGORITHM

The following is an abstract description of the algorithm used by the PL.8 compiler[8] to generate selection code for the case statement. The PL.8 compiler generates code for machines including the IBM 801, IBM System/370, and Motorola MC68000. The following is a list of parameters used in the algorithm and values that PL.8 uses when generating code for various machines.

MinCaseDensity: minimum case density allowable for a jump table. (1/2 for all machines)

MinForJumpTable: minimum number of case items allowed in a jump table (5 for 801, 6 for MC68000 and S/370)

MinForBinSearch: minimum number of case items allowed in a binary search. (4 for all machines where the case selector is scalar, 3 for non-scalar case selectors.)

Initially the case-statement selection code is generated as a linear search (i.e. a sequence of 'compare' and 'branch equal' instructions) surrounded by markers to delimit the beginning and end of the code. From this code, case items are extracted from the immediate values of the compare instructions. The process then proceeds as follows:

1. Sort case items (giving error messages concerning duplicate case items in the case statement).
2. Starting with all the case items as one cluster, break up cluster(s) using the greedy approach, until each has case density greater than MinCaseDensity. (This does not necessarily mean that the cluster will be carried out as a jump table.)
3. For each cluster where the number of case items is greater than MinForJumpTable replace the code that performs 'compare' and 'branch equal' tests on these case items with a code fragment for a jump table, or simply use a range test if all case items of the cluster belong to the same clause and the case density is 1.
4. For all remaining clusters not handled by the previous step because there were too few case items in the cluster, combine as many case items as possible into range tests.
5. Combine range tests generated by the previous two steps (recall that a jump table contains a range test) and the remaining single case items not covered by some range test into a binary search if the number of such tests is greater than MinForBinSearch, or a linear search otherwise.

The initial code generation phases of the PL.8 compiler try to produce strength-reduced code. Thus, a comparison of a string variable with a string constant will be changed into an unsigned register-to-integer constant comparison when the string is of fixed length and can fit into a register. The integer value used corresponds to the bit pattern of the string reinterpreted as an integer. The same is done for comparisons of short floating-point quantities. Since it is rare to find the values of case items densely packed under this new interpretation, floating point and string case items will invariably be assigned one case item per cluster.

Faster executing code can be produced if the probabilities that the case selector takes on the case item values are known. These may be known as a result of trace information that is automatically supplied to the compiler, or perhaps as an extra-lingual mechanism pertaining to the case statement. In step 5, the linear search can be arranged in decreasing probabilities, and a Huffman search rather than a binary search can be used. When combining case items into range tests (possibly used in a jump table), the probabilities should be added together.

ACKNOWLEDGEMENTS

Dick Goldberg initially wrote the programs that process case statements for the PL.8 compiler; his ideas have greatly influenced the given method. I would like to thank Peter Markstein who suggested writing this paper and Hank Warren for discussions that have improved my understanding and presentation of the problem. Finally, I thank the editor and reviewers for concentrating my diffuse verbiage.

REFERENCES

1. *Reference Manual for The Ada Programming Language*, U.S. Department of Defense, July 1980.
2. G. Radin, 'The 801 minicomputer', *Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, 1–3 March, 1982, pp. 39–47.
3. A. Sale, 'The implementation of case statements in Pascal', *Software—Practice and Experience*, **11**, 929–942 (1981).
4. L. Atkinson, 'Optimizing two-state case statements in Pascal', *Software—Practice and Experience*, **12**, 571–582 (1982).
5. J. Hennessey and N. Mendelsohn, 'Compilation of the Pascal case statement', *Software—Practice and Experience*, **12**, 879–882 (1982).
6. M. Garey and D. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., San Francisco, 1979, p. 281.
7. A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, Mass., 1974.
8. M. Auslander and M. Hopkins, 'An overview of the PL.8 compiler', *SIGPLAN Symp. on Compiler Construction*, Boston, 23–25 June 1982, pp. 22–31.