Special Feature:

Computer Design to Minimize Memory Requirements

Eric C. R. Hehner
University of Toronto

Matching the instructions and their representations to the distributions of usage can save 75% of the space taken by contemporary machine representations. The gain in space may be accompanied by a reduction in execution time due to more efficient use of data paths. Variable-length codes can also eliminate all forms of overflow from machine-language, and greatly reduce the probability of overflow in data.

Introduction

Language-directed machine design refers to the effort to make computer instructions and structures appropriate for representing and executing programs written in high-level languages. (Extensive bibliographies are available.[1,2]) It has sometimes been taken to mean the design of computers whose machine language is close to, or identical to, a desired high-level language. This is most successful when applied to machine-directed language like Fortran,[3] or a low-redundancy language like APL,[1] the advantage being the elimination of the need for a compiler. But a well-designed high-level language contains redundancy to serve as a check on the programmer's intentions. And a compiler is useful to check the consistency of the redundant information, and to remove it prior to execution, although unchecked redundancy should remain. Declarative information, for example, may be checked and removed, while range checks may remain.

The language that is best for programming is not necessarily the language that is best for execution, but neither are they independent. If the machine is well designed, a compiler can concentrate on analyzing the source program, rather than on synthesizing the object program from inappropriate instructions. And the space occupied by the object program will be as close as possible to the minimum required to represent the execution information contained in the source program. As McKeeman[4] has expressed it, ". . . it is absurd to expect carefully engineered, very fast, automatic desk calculators [i.e., contemporary machines] to be very good for implementing operating systems or compilers. We are forced to manufacture the operations we want out of sequences of hardware operations with the obvious result that the programs become large. When engineers begin to seek more efficient encodings for commonly used sequences of instructions, progress toward the modern computer may begin."

For the most part, language-directed machine designs have come from the experience and intuition of individual designers. Wortman[2] introduced a method to this process by basing the design of a machine on a statistical analysis of the source language for which it was intended. This paper follows Wortman's approach. Although we can suggest no substitute for inspiration in the initial choice of an instruction set, we shall suggest a method of algorithmically improving a given instruction set.

The other purpose of this paper is to present ways of encoding instructions and data to fit the information being represented. It is common knowledge, formalized in 1949 by Shannon,[5] that one can make the best use of memory and data paths by using a variety of instruction and data sizes, encoding frequently-used instructions and data in fewer bits at the expense of longer codes for infrequent ones. We shall find how much variability is appropriate, and how to take advantage of it.

The benefits we expect are, in summary:

1. The compiler-writer's task becomes much easier; he need not spend so much thought on how to synthesize the actions he wants from the operations provided.
2. Space requirements are reduced in two ways: (a) simpler compilers require less space; and (b) the compiled programs require less space. In fact, if a compiler is itself a program compiled from a high-level source language, then savings (a) and (b) above apply independently to it.
3. If execution speed is limited by the speed with which information moves through some data path (e.g., between primary and secondary memories, or between memory and processor), then time will be reduced by packing information more densely, thus making more efficient use of available bandwidth.
4. Overflow is a problem that results from choosing a fixed-length encoding for an infinite, large, or expandable class of values. Variable-length encodings, in addition to saving space and time, have the benefit of eliminating all forms of overflow. With a variable-length operation encoding, one more operation can always be added to the machine. With a variable-length address field, the machine language does not limit the memory that can be added, nor the address space that a program can occupy. Limits will, of course, be imposed externally by economics, but as economic conditions change, these limits are free to be changed without changing the machine language. For any given amount of memory, there is a limit to the size of integers or floating-point numbers that can be represented; available memory is an easily justified limit, but a restrictive encoding is not.

Variability of instruction lengths

Let us assume that all instructions are addressable, and that the addressable unit of instruction storage is b bits. To avoid wasting space by alignment of instructions to the addressable unit, the various instruction sizes must all be multiples of b bits. Let us assume that each instruction is composed of independently encoded fields (e.g., operation, register, offset, immediate operand), and that the average number of fields per instruction is f (typically $f = 2$-4). This means that the various code lengths of a given field must differ from one another by multiples of b bits (otherwise one could compose an instruction whose length is not a multiple of b bits). Essentially, b bits is the basic character or unit of encoding. The smaller b is, the more able we are to choose encodings that match the distributions of usage, and therefore save space. On the other hand, addressing to a finer unit of storage costs bits on each branch address. This tradeoff should determine the optimum addressable unit of storage for instructions, and optimum variability in instruction lengths.

The following argument, though rough, gives us an indication of the optimum value of b. The waste (or redundancy) in an encoding is the difference between the number of bits taken by the encoding, and the information content being encoded (for mathematical definitions, see Hehner[6]). Huffman[7] has proved that in a minimum-space encoding the waste is limited to one character of encoding per message. In our context, this means that the waste, in a "best" encoding, is limited to b bits per field. Let us assume that the actual waste is proportional to b, with proportionality factor w. (For a variety of frequency distributions, we found typically $0.1 \leq w \leq 0.5$.) Then the wasted space in an average instruction is fbw bits. Changing the unit of variability from $2b$ bits to b bits changes the waste per instruction from $2fbw$ to fbw bits, a saving of fbw bits.

Let the relative frequency of branch instructions be x (in one sample[8] $x = 0.1$). Changing the addressable unit from $2b$ bits to b bits adds an average of 1 bit per branch address, or x bits per instruction. Therefore, in the instruction stream, an addressable unit of b bits is preferable to an addressable unit of $2b$ bits if $fbw > x$, i.e., if $b > x/(fw)$. Typically, the right side of this inequality has a value in the range 0.05-0.5, indicating that, for instruction lengths, bit-variability is preferable to any coarser unit.

The above conclusion is based on the assumption that the waste in each field of an instruction is proportional to the upper bound on the waste in a minimum-space encoding of the field. But for some fields, such as addresses and immediate data, a minimum-space encoding is either impossible to find, or impractical, and we are left to our coding ingenuity. When that is inconsistent for changing b, w will not be constant. When our ingenuity is weak, w will be large (> 1) strengthening the conclusion that b should be as small as 1 bit.

To confirm this conclusion, we chose a large, well-known compiler[9] (4420 lines of source, 60K bytes of object code), and allowed ourselves the freedom of bit-variability in its encoding. Some of the results of our experiments[10] are reported in the following sections.

Operations

Given a set of operations, and a sample of programs compiled into sequences of these operations, we can find the frequency of use of each operation, and hence encode them according to an algorithm given by Huffman.[7] This encoding will minimize the space required for operations over all possible encodings, under the assumption that the operations appear with independent probabilities. By including a zero-frequency nonoperation in the set, we obtain an open-ended code, allowing future expansion, at a cost of one bit on the least frequent operation.

It has been observed, however, that in sequences of operations compiled from high-level source languages, operations do not appear with independent probabilities.[2,8,11] There are several ways to take advantage of the dependencies; the two presented here are iterative pairing and conditional coding.

Iterative pairing works as follows: one pair of operations is chosen, and a new operation is invented to replace that pair wherever it occurs in a sequence of operations (except where the second is the target of some branch). This process is repeated some number of times; at each stage the pair chosen may include operations invented at previous stages. Finally, we use an open-ended minimum-space encoding for the resulting operation set. The object is to choose pairs such that the new operation set can be encoded much more densely than the original. We could choose, at each stage, the pair that makes the most improvement at that stage; but this computation is quite complex, and it may not result in as much improvement over several iterations as some other sequence of choices. In practice, choosing the most frequent pair is easy, and it seems to work well. In our sample, increasing the operation set from 47 to 178 operations decreased the space required for operations to less than half of the space required before pairing, and to less than one quarter of the original 8-bit fixed-length encoding.

Conditional coding is a generalization of a technique introduced by Foster and Gonter.[11] For a given operation, say LOAD, every operation has a certain probability of following it, so the operation set can be given minimum-space encoding in the context following LOAD. Similary, in the context following each of the other operations, the operation set can be given a minimum-space encoding. In general, the encodings following different operations will be different, so that the interpretation of an operation code will depend on the interpretation of the preceding operation code. (Once again, the target of a branch requires special consideration.) For even better results, the encoding of an operation can depend on the preceding pair, or n-tuple, of operations.

The mechanism needed to implement this scheme is remarkably simple. It consists mainly of a shift register that contains an unconditional representation of the context. This context and the current operation code together specify both the current operation and the length of the current code. Then the new context is shifted into the context register, and the proper amount of the instruction stream is consumed. The target of a branch must be encoded in a standard context, rather than in the context of the preceding operations, and the branch instruction must cause this standard context to be placed in the context register. A branch of distance zero can be used to establish a standard context whenever needed.

When we tried this on our sample, the results were as follows: encoding operations in the context of 1, 2, and 3 preceding operations reduced the space required for operations by 43, 53, and 56% respectively, or to 2.1, 1.7, and 1.6 bits per operation code, on the average.

The former technique, iterative pairing, is a method of improving the set of operations, whereas the latter, conditional coding, is just a coding technique. Both take advantage of interinstruction dependencies to match the machine-language representation of a program more closely to its information content. They relied on the fact that there is only a small number of different operations on a computer. The other portions of instructions—addresses and immediate data—are large, conceptually infinite, classes, so other techniques are needed.

Constants

Constants in the source program may take the form of immediate data in the machine-language program. This is common for small integers but uncommon for other numbers (large integers, floating-point numbers) or for character strings, which do not fit into the fixed-size space allotted to immediate data in most instruction formats. But with variable-length instructions, any constants may sit in the instruction stream, avoiding the need for indirection.

The use of variable-length encodings for integers is not new; the encodings used, however, have been decimal sign-and-magnitude, which is neither very compact nor very good for arithmetic. And the arithmetic algorithms used have been digit-serial, which is rather slow. But the encodings need not be decimal, nor sign-and-magnitude, and the arithmetic unit can be designed to handle as many bits in parallel for variable-length as for fixed-length encodings. The Burroughs B1700, for example, has an arithmetic unit that is 24 bits wide, and can be used iteratively (for long operands), and fractionally (for short operands).[12]

To test the effect of variable-length encodings on the space required for the integer constants in our sample, we tried several schemes, including a two's complement scheme,[13] and the binary sign-and-magnitude scheme of the appendix. These encodings gave similar results: integer constants required, on the average, 5.5 bits each. The success of these encodings depends on the fact that small integer constants, especially 0 and 1, are more common than large ones.

Floating-point numbers can be represented simply by pairs of integers, and can be added to a machine in either of two ways: by adding new operations to operate on the new number type, or by representing all numbers as the more complicated type. The latter alternative becomes very attractive when only two extra bits are required to represent a zero exponent.

Data addresses

A form of data address that is generally recognized as being suitable for programs compiled from block-structured languages is called "lexic-level, order-number pairs." To design a suitable encoding, we must discern the patterns of use. One pattern is that small lexic-levels and order-numbers are more common than large ones. Even if the use of lexic-levels within a program is uniform, and even if the use of order-numbers within each block is uniform, the above observation will still hold for the following reason: Since level- and order-numbering begin at 0, every program has a lexic-level 0, and every block that contains local storage has in it an item with order-number 0, but progressively fewer programs and blocks have progressively higher levels and order numbers. Therefore, the encoding given above for integer constants is suitable for this form of data address. The result for our sample was an average address length of 7.3 bits.

Branch addresses

The regularity in branch addresses that allows us to take advantage of more than one address length has been observed by several people.[8,14] It is simply that short jumps are more common than long ones. Therefore, by expressing branch addresses relative to the instruction pointer, the encoding used above for integer constants becomes appropriate also for branch addresses. Transfer between separately placed instruction sequences requires some other mechanism, such as a CALL and RETURN instruction. But within an instruction sequence, relative addressing has the advantage for coding, as well as another advantage: relocatability without base registers.

A major problem confronts the user of a machine with more than one length of branch address: During instruction assembly, how much space should be reserved for a forward branch? Richards[15] has given a solution for two sizes that is nonlinear in the number of branches, and for more sizes the problem appears to require a combinatoric solution. The cause of this complexity, however, is simply the notorious *go to.* If we restrict ourselves to disciplined control structures such as *if . . . then . . . else . . ., while . . . do . . .,* and *repeat . . . until . . .*, the solution becomes linear in the number of branches. If code is generated "from inside to outside," then the restriction to structured programming ensures that branch addresses at each stage can be calculated knowing only the size of inner structures. This is compatible with the order that reductions are performed during a "bottom-up" parse, and the order of code generation required for certain optimizations.

Here is where we pay for the gains described in the preceding sections: if instruction lengths are completely variable, then we must express the length of a jump in bits, rather than bytes or words, and this tends to lengthen branch addresses. In our sample, the average branch address required 14 bits. Fortunately, as the next section shows, the loss is much less than the gain.

Evaluation of instruction encodings

Putting together the above results, we reduced the space required for a machine-language representation of our sample to one quarter of the original IBM 360 machine language version. This gain could be attributed about equally to the more appropriate instruction set, and to the more appropriate encodings of the fields within instructions.

Variabilty of data lengths

If we are given the distribution f of the lengths of storage spaces allotted to variables, then we can find the appropriate addressable unit for data in a manner similar to that for instructions. As was the case there, assuming that all variables are to be addressable, a coarse addressable unit saves space in the address field, and costs space in the addressed item.

Let the addressable unit of data, and therefore the effective degree of variability in data lengths, be b bits. Then the average number of addressable units per variable is

$$a = \sum_{s=1}^{\infty} \lceil s/b \rceil f(s),$$

where the rounding-up operation ensures addressability. Using the representation of the appendix for addresses, and assuming that the probability of referencing a variable is independent of the variable, the average space per address is approximately

$$d = (1/n) \sum_{i=1}^{n} 2 \log_2 (ia)$$

$$\approx 2 \log_2 (na) \text{ for large } n$$

where n is the total number of variables.

Suppose there are, on the average, x data addresses per data item. For branch addresses we have $x \ll 1$ since most instructions are not the object of any branch. But all useful variables are referred to at least once, so for data addresses $x \geq 1$. In one sample,[8] $x = 10.4$. The total space for addresses and variables, per variable, is

$$S = xd + ab,$$

with units

$$\begin{aligned}(\text{space/vbl}) = {} & (\text{addresses/vbl}) * (\text{space/address}) \\ & + (\text{addressable units/vbl}) \\ & * (\text{space/addressable unit}).\end{aligned}$$

Given f and x, we choose b to minimize S. It can be seen that the value of b that minimizes S is independent of n.

If the form of variable declaration specifies the space to be allocated to each variable, as in PL/I, then the distribution f, as well as the addressing frequency x, are easily tabulated from a sample of programs. Unfortunately, most PL/I programmers are well aware of the architecture of their machines, and their declarations tend to reflect this knowledge, rather than the problem requirements. So an analysis will simply confirm the appropriateness of their machine architecture for their programs. Pascal declarations can provide the same information, but in a form that is independent of representation—by limiting the values of variables to a specified finite range.

In many languages, a declaration specifies only the type, not the range, of a variable. In the absence of more information, all variables of a given type are allocated equal space. Then S is minimized when b is that space, independent of x. In our sample, some declarations gave some space information, but it was heavily influenced by the machine architecture; other declarations gave only type information. We therefore allocated equal space to all variables of a type. As we see in the next section, the space appropriate for integer and character data types turned out to be equal.

Space allocation for data

Variables present us with a problem that constants do not—namely, their values vary. If we know only the types of variables, then whatever representation of their values we choose, whatever space we allocate for them, we may find that one of them is being assigned a value that won't fit in the space provided (or for which there is no representation). When this happens, the usual "solution" is to shout "overflow" and give up, or worse, to overwrite the following data item. To minimize these undesirable occurrences, we usually allocate much more space than is required on the average, and consequently use space inefficiently. If we know the range of values of each variable, we can allocate the maximum space required for any value. This solves the overflow problem, but most of the time it is a great waste of space.

For large variables, such as character strings and arrays with variable dimensions, the storage problem has sometimes been solved by the use of fixed-length descriptors, which give the length and location of the value in a "free area," with periodic or continual space reclamation (garbage collection). In one respect this is an improvement, in that overflow occurs only when all variables simultaneously have values such that their total length exceeds the free area. With a proper encoding of values, i.e., one that gives long codes only to uncommon values, the probability of overflow is greatly reduced.

The descriptor scheme has a drawback that prevents its general adoption for all data. The space overhead for an extra address, and the time overhead for an extra memory access and for storage management, may be small relative to large data items, but they are enormous relative to small ones. A complete trace of all values of all integer variables and array elements during an execution of our sample program revealed that the value of an integer variable required, on the average, 6.5 bits according to our encodings. They seem, therefore, to be too small for the indirect "descriptor" mechanism. Surprisingly, the average length of character string variables turned out to be two characters, so they also seem too small for descriptors, most of the time. (Character string constants were longer—14.6 characters on the average.)

The solution is to allow the number of levels of indirection to vary, being zero for values which fit into a few bits, and one or more for longer values. That way we can keep the space allotted to each variable to a minimum, and pay for indirection only when it is needed to avoid overflow. One bit per variable can be reserved to indicate whether the value is currently short and present, or long and in the free area.

Evaluation of data encodings

How much space should each variable be allocated? How large should the free area be, and what should the addressable unit within the free area be? And how does the cost of this scheme compare to the cost of standard "fixed-length" allocations of space?

For our evaluation, we chose the space/time product as our cost function. Space includes the initial space per variable, and the free area. Time is measured in memory references required to load and store the values of variables, and memory references required to perform garbage collection. To evaluate the last factor, garbage collection, we made the conservative assumption that values consume space until the free area is exhausted, with no attempt to fit values into available spaces that are interspersed with occupied spaces; a smarter algorithm may give better results. Some recent work[16,17] has shown that the garbage collection process can take place concurrently with the main (garbage producing) process; the running cost of the two processes together should therefore be less than the sum of their separate costs. Our results do not incorporate these improvements.

For integer data, we found that an initial space per variable of 16 bits, free space equal to 2.8 bits per variable, addressable unit within the free area of 8 bits, and data path width of at least 16 bits, gave the same cost as the standard fixed-length scheme that allocates 22 bits per variable, with a data path width of at least 22 bits.

Contemporary machines often give integer data more than 22 bits. Their running costs are higher, according to our cost function, than the variable-length scheme described above. If a machine gives us a variety of fixed-length number sizes, we must choose among them, and overflow occurs if the value of any one variable requires more space than it was allocated. But for less cost, the variable-length scheme frees us from choosing a number size, and overflow occurs only when all variables simultaneously have values such that the total length of all values in the free area exceeds the free area.

For character data, we found that an initial space per variable of 2 characters, free area equal to 9.4 characters per variable addressable to the character, and data path width of 8 characters, gave the same cost as the fixed-length scheme that allocates 33 characters per variable.

If each character is 8 bits, then our initial space per variable and the addressable unit within the free area are, by coincidence, the same for character strings and integers. In each case, the initial space is the minimum required to address the free area.

Conclusion

It is foolish to provide what is easy instead of what is wanted. For example, concatenation of variable-length character strings is a common operation in some programming languages and computing environments, yet it is exceedingly rare in the operation set of computers. Of course, provision of this operation is difficult, involving memory management. But the problem does not go away by ignoring it. In fact, it grows: it must be solved by each compiler writer, or worse, by each programmer who wants concatenation. And their solutions are bound to execute more slowly and require much more memory than a hardware or firmware operation.

Many factors go into the choice of representation of machine languages; this paper is concerned with only one of them, and is by no means a complete analysis. We have shown that matching the instructions and their representations to the distributions of usage can save 75% of the space taken by contemporary machine representations. At the same time, we make more efficient use of data paths, and perhaps reduce execution time. We can also eliminate all forms of overflow from machine-language, and greatly reduce the probability of overflow in data. ■

Appendix

This appendix contains a variable-length binary sign-and-magnitude number scheme. The number of leading 0's (after the minus sign if present) tells how many bits follow constituting the value.

0: 11	
1: 01	-1: 101
2: 0010	-2: 10010
3: 0011	-3: 10011
4: 000100	-4: 1000100
5: 000101	-5: 1000101
6: 000110	-6: 1000110
7: 000111	-7: 1000111
8: 00001000	-8: 100001000

This scheme is positively biased; it matches a distribution in which a positive integer is twice as common as the corresponding negative integer. When only non-negative integers are required, zero may be represented by a single bit. When only positive integers are required, the leading 0 is unnecessary.

Acknowledgment

This work has benefitted from discussions with several people, particularly D. B. Wortman, J. J. Horning, W. M. McKeeman, and R. N. S. Horspool.

References

1. P. S. Abrams, "An APL Machine," Ph.D. thesis, Computer Science Department, Stanford University, Palo Alto, California, June 1970.
2. D. B. Wortman, "A Study of Language Directed Machine Design," Ph.D. thesis, Computer Science Department, Stanford University, Palo Alto, California, 1972.
3. T. R. Bashkow, A. Sasson, and A. Kronfeld, "System Design of a FORTRAN Machine," *IEEE Transactions on Electronic Computers*, Vol. EC-16 (4), August 1967, pp. 485-499.
4. W. M. McKeeman, "Language Directed Computer Design," *Proc. AFIPS 1967 FJCC*, Vol. 31, AFIPS Press, Montvale, New Jersey, pp. 413-417.
5. C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, Urbana, 1949.
6. E. C. R. Hehner, "Information Content of Programs and Operation Encoding," *JACM* (to appear).
7. D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proc. I.R.E.*, Vol. 40 (9), pp. 1098-1101, September, 1952.
8. W. G. Alexander and D. B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer*, Vol. 8, No. 11, November 1975, pp. 41-46.
9. W. M. McKeeman, J. J. Horning, and D. B. Wortman, *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, New Jersey, 1970.
10. E. C. R. Hehner, "Matching Program and Data Representations to a Computing Environment," Tech. Report CSRG-44, Computer Systems Research Group, University of Toronto, Ontario, November 1974.
11. C. C. Foster and R. H. Gonter, "Conditional Interpretation of Operation Codes," *IEEE Transactions on Computers*, Vol. C-20 (1), January 1971, pp. 108-111.
12. W. T. Wilner, "Design of the B1700," *Proc. AFIPS 1972 FJCC*, Vol. 41, pp. 489-497.
13. E. C. R. Hehner and R. N. S. Horspool, "Variable-length Radix Complement Number Representations," to appear.
14. H. J. Saal and L. J. Shustek, "Microprogrammed Implementation of Computer Measurement Techniques," *Preprints, Fifth Annual Workshop on Microprogramming*, University of Illinois, September 1972, pp. 42-48.
15. D. L. Richards, "How to Keep the Addresses Short," *CACM*, Vol. 14 (5), p. 346, May 1971.
16. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly Garbage Collection: *An Exercise in Cooperation*," private communication, Oct. 1975.
17. G. L. Steele, Jr., "Multiprocessing Compactifying Garbage Collection," *CACM*, Vol. 18, No. 9, September 1975, pp. 495-508.

Eric C. R. Hehner is an assistant professor in the Department of Computer Science at the University of Toronto, and is also a member of the Computer Systems Research Group. He received his B.Sc. in mathematics and physics from Carleton University in 1969, his M.Sc. and Ph.D. degrees in computer science from the University of Toronto in 1970 and 1974 respectively. His current research interests are programming language design, compiler design, and machine design.

Dr. Hehner is a member of the ACM and the IEEE Computer Society.