SPE
480

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 18(00), 000–000 (1988)

Compiler Support for Floating-point Computation

CHARLES FARNUM
Computer Science Department, University of California, Berkeley, Berkeley, California 94720, U.S.A.

SUMMARY

Predictability is a basic requirement for compilers of floating-point code — it must be possible to determine the exact floating-point operations that will be executed for a particular source-level construction. Experience shows that many compilers fail to provide predictability, either because of an inadequate understanding of its importance or from an attempt to produce locally better code. Predictability can be attained through careful attention to code generation and a knowledge of the common pitfalls. Most language standards do not completely define the precision of floating-point operations, and so a good compiler must also make a good choice in assigning precisions of subexpression computation. Choosing the widest precision that will be used in the expression usually gives the best trade-off between efficiency and accuracy. Finally, certain optimizations are particularly useful for floating-point and should be included in a compiler aimed at scientific computation. But predictability is more important than efficiency; obtaining incorrect answers fast helps no one.

KEY WORDS Compilers Floating-point arithmetic Optimization

INTRODUCTION

Floating-point programs must be carefully compiled in order to produce accurate results. Data accessing primitives, control structures and integer arithmetic are clearly defined in most language standards, since most hardware is more or less equivalent in the support it offers for these tasks. But floating-point systems vary widely, so language standards cannot specify perfectly the semantics of source-level floating-point operations.[1] The implementor is left with the difficult task of deciding what machine-level operations will result from source-level code.

Unfortunately, most compiler writers are ill-equipped to handle this task. Compiler texts and classes rarely address the peculiar problems of floating-point computation, and research literature on the topic is generally confined to journals read by numerical analysts, not compiler writers. Many production-quality compilers that are excellent in other respects make basic mistakes in their compilation of floating-point, resulting in programs that produce patently absurd results or, worse, reasonable but inaccurate results.

An implementation of a large floating-point library is a job for specialists. But given such a library, little information is actually needed to produce good floating-point code

0038–0644/88/000000–00$05.00

Received 24 March 1987
Revised 21 December 1987

for current machines. This paper provides the necessary information. Much of this information is specific, and a short paper on the topic cannot possibly address all of the issues; therefore, this paper is a compendium of floating-point concerns. A more detailed view of the issues involved in floating-point computation can be obtained by learning about the IEEE floating-point standard,[2, 3] a popular system that is fast becoming the norm on new floating-point hardware.

Notational conventions

The following conventions are used in this article:

1. In general, computer-oriented entities are set in sanserif and mathematical entities are in *italics*.
2. When discussing different precisions, the precision of programming variables is indicated by the first letter of their name: s for single and d for double.
3. A mathematical symbol with the same name as a programming variable denotes the mathematical value represented by the current bit-pattern of the programming value.
4. The functions sngl and dble round their operands to single or double precision, respectively.
5. The letter *s* or *d* over an arithmetic operator indicates that the result of that operator is rounded to the given precision.

PRODUCE PREDICTABLE CODE

A programmer must be able to predict the floating-point operations that will result from a particular source-level construct in order to write good code. Without such predictability, guaranteeing error bounds on the output is as difficult as proving the correctness of a loop without knowing whether it is tested at the top or bottom. Yet many compilers, in practice, violate this basic need for predictability either by accident or in a quest for 'better' code. This section lists common errors in past compilers.

Differing precision in registers and memory

Faulty implementations of common subexpression elimination can lead to trouble if subexpressions are computed to a higher precision than variables, a common tendency in new floating-point chips. Consider the following sequence of statements:

```
s := dx/dy;
d := s - dx/dy;
```

The compiler may notice, when computing d, that the value of s is still lying around in a register and can be used without loading. This optimization is valuable if the value in s and the value in the register are, in fact, the same; but if the value in s is different, owing to rounding from subexpression precision to storage precision, the substitution would be incorrect.

In addition, common subexpression elimination may notice that dx/dy is used twice, and thus store its value in a temporary location for reuse. If the temporary location has less precision than the register in which the subexpression was evaluated, the wrong

value will be used in the computation of d. This example is primarily of interest on older machines; newer machines generally allow storage of the widest possible format.

Unwanted extra precision

Most languages supporting multiple precisions allow implementations to map more than one source-language precision to the same machine precision. Doing so can make certain codes malfunction and should only be done if the target machine is limited to one precision. In particular, promoting all variables from single to double is *not* doing the programmer a favour, even if the particular target machine 'has plenty of memory'. Saving space is not the only reason for using differing precisions; some programs depend on, for instance, double being twice the precision of single, by making use of the fact that the product of two single values can be represented exactly in a double variable. Other programs may require a difference to be either zero or large compared with round-off error; in these programs, a common practice is to round the difference to single precision, flushing values near the double precision round-off error to zero. Although few language standards insist that double must have greater precision than single, such has been the case on virtually all compilers and machines in the past; the capability should not be removed if the target machine easily supports it.

Algebraic transformations

Different source languages have different restrictions on when algebraic transformations can be used in evaluating expressions. The rules of the language must not be broken unless the transformed code has *exactly* the same execution semantics as the untransformed code on the target machine, including the signals generated. Often the semantics are different; for example, on some machines, such trivial rules as $x*y = y*x$ or $1.0*x = x$ do not hold. Using associativity or distributivity can often change a result by changing the order in which rounding occurs; for example

```
(10.0E-30 + 10.0E30) - 10.0E30
```

and

```
10.0E-30 + (10.0E30 - 10.0E30)
```

will yield quite different results. Although many languages allow the compiler to apply algebraically correct transformations to expressions for optimization purposes, the programmer is usually best served by leaving the expressions alone.

Regardless of what the language allows, the groupings implied by parentheses should never be changed. Carefully written code often requires associating operations in a particular way to ensure, for instance, that overflow does not occur; parentheses are a clear and concise way of showing the programmer's intent.

Decimal to binary conversion

The first requirement of decimal to binary conversion is that identical results should be obtained regardless of when the conversion occurs. Some existing systems yield three different values for the same decimal constant, depending on whether the conversion is

done by the compiler, the assembler or the run-time input libraries; producing these different values is unacceptable.

Secondly, it is important to realize that decimal to binary conversion under the IEEE standard is an important operation in its own right, with the possibility of generating signals and being affected by the current rounding mode. Thus, doing the conversion at compile time is not always a trivial task; for instance, in a system fully supporting dynamic rounding modes, two different versions of each inexact constant must be maintained, each with possibly different signals generated on use.

If run-time signals from conversion are not wanted, they can be avoided by using the particular language's built-in facility for defining constants as static objects, e.g., FORTRAN's DATA statement or Pascal's CONST declarations. Signals generated while converting these constants can be reported appropriately during compilation. The programmer should also be able to specify rounding modes for these static constants.

Finally, the conversion routines themselves should not be written without a great deal of thought, as the obvious algorithms are fraught with danger. Reference 4 provides efficient and accurate algorithms for these conversions.

Type checking

Many new languages require strong type checking across separately compiled modules, supported by interfaces that provide the necessary type information. Compilers for such languages should enforce the strong type checking by including consistency checks to ensure that the same interfaces were used for separately compiled modules; a depressingly large number of current compilers do not do so, usually with the excuse that 'the linker isn't smart enough'. This excuse is invalid; a special linker pre-pass can be written to do the consistency checks, or they can be trivially implemented at run-time by including the test in the initialization code of each module.

Unfortunately, most older languages, including FORTRAN, provide no means for type checking across separate modules. The resulting problems are by no means limited to numeric code, but a particular instance occurs quite often due to a property of certain floating-point formats on the VAX and IBM/370. On these machines, double precision values can be chopped to single precision values by discarding the lower half of the bit pattern. If a variable in memory is treated inconsistently as double precision in some places and single precision in others, the program will usually give plausible results. In FORTRAN, all parameters are passed by reference; thus, this problem occurs any time a double value is passed to a procedure expecting a single value. If such a program is executed on a machine whose formats do not have this property, the program will malfunction; these bugs are notoriously difficult to find.

Because of the insidious nature of these types of bugs, and their frequent occurrence in numerical code, as much type checking should be done as possible, regardless of what the language requires or allows.

EVALUATION PRECISION OF SUBEXPRESSIONS

In languages that support multiple floating-point precisions, the arithmetic functions are typically overloaded, e.g., the symbol + might represent the function $+^{s}$ or $+^{d}$

depending on the context.* Deciding which function to use is often partially left to the implementor; the choice should be considered carefully. Below, we discuss some alternatives.

Strict evaluation

Most languages require that the precision used to evaluate subexpressions with an overloaded operator be at least as wide as the widest operand. Strict evaluation uses the narrowest precision allowed by the language. For example, the statement d := d + s*s would store $d \overset{d}{+} (s \overset{s}{\times} s)$ in d. This strategy is easy to implement and the most efficient on machines such as the VAX, whose machine instructions always yield results of the same precision as the operands.

Unfortunately, strict evaluation is almost never what the programmer desires when precisions are mixed within an expression. In the previous example, the extra precision in the variable d is entirely wasted, as it is swamped by the rounding error introduced by rounding $s \times s$ to single precision. Another example is the evaluation of d := 7.0/3.0 * d; if 7.0/3.0 is evaluated to single precision, the extra precision of d will be destroyed.

A common counter to these arguments is that a 'careful programmer' writes

```
d := d + dble(s) * dble(s)
```

to ensure that the necessary precision is carried. These explicit conversions make the code more portable, but at the expense of programmer effort and program legibility. The intent of overloaded operators is to eliminate clutter; the strict evaluation strategy forces such clutter back in.

Furthermore, using strict evaluation does not eliminate the portability problem, since some compilers use other strategies. Thus, the same argument justifies insisting that careful programmers should write explicit coercions for every expression, to ensure that the same strategy is used on all machines. A better solution to this particular portability problem is to leave the job to the compiler; given a compiler that makes well documented decisions on unspecified portions of the language, it is a worthwhile and relatively inexpensive task to write a source-to-source translator that removes most of the implementation specific code. For example, this tool could easily replace d := d + s*s with d := d + dble(s)*dble(s).

Widest available

An easily implemented alternative to strict evaluation uses the widest precision supported by hardware as the result precision of all overloaded operators. In a machine where the widest format supported is also the fastest, e.g. many of the new microprocessor floating-point chips, this strategy yields both the fastest speed and the most accurate results.

---

* Decimal to binary conversion can also be considered as an overloaded operator in that the precision of the constant should ideally depend on the context where it is used. Writing 0.3D0 instead of 0.3 is both tedious and easy to overlook; it has the further disadvantage that a program cannot be upgraded from single to double precision simply by changing the variable declarations.

Although often ideal, using the widest available format is troublesome in the following two cases. On most machines, higher precisions imply execution times. This extra cost comes both from the need to manipulate extra bits during the computation and from the need to convert values from one precision to another. Whether or not the extra precision is worth the higher execution time depends significantly on the context. Three simple examples:

1. `d := d + s*s`

   Here, the higher cost of evaluating $d \overset{d}{+} (s \overset{d}{\times} s)$ instead of $d \overset{d}{+} (s \overset{s}{\times} s)$ is certainly worth while; otherwise, the extra precision of d is wasted.
2. `s := s1 + s2 * (s3 + s4/s5)`

   Although

   $$\mathrm{sngl}(s_1 \overset{d}{+} s_2 \overset{d}{\times} (s_3 \overset{d}{+} s_4 \overset{d}{/} s_5))$$

   may be a marginally better value than

   $$s_1 \overset{s}{+} s_2 \overset{s}{\times} (s_3 \overset{s}{+} s_4 \overset{s}{/} s_5)$$

   it would be hard to justify the added cost.
   If the cost *is* justified, then the programmer should write

   `d := s1 + s2 * (s3 + s4/s5)`

   in order to take advantage of the extra precision. The widest needed strategy, suggested below, uses single if the value is to be stored in s and double if it is to be stored in d.
3. `s := s1 + s2`
   On a reasonable machine,

   $$\mathrm{sngl}(s_1 \overset{d}{+} s_2) = s_1 \overset{s}{+} s_2$$

   here the extra expense is simply wasted.

Since most statements in typical code are of the simple form given in the third example, the use of the widest available precision is often wasted; if the widest precision is computationally expensive, then this strategy loses much of its appeal.

Secondly, in a system that supports infinitely many precisions, the 'widest available' format is non-existent. Such systems should use the 'widest needed' strategy outlined below.

Widest needed

Strict evaluation eliminates much of the usefulness of overloaded operators by forcing the programmer to use explicit type conversions whenever a benefit is to be gained from varying precisions. Using the widest available precision wastes the extra computation in frequently arising situations. A natural suggestion is to use the widest precision that is 'needed', i.e. that will be used in the local context.

More precisely, assigning precisions to an expression tree using the widest needed strategy can be described as follows:

1. Assign tentative precisions using the strict evaluation strategy, in a bottom-up traversal of the tree.
2. Using a top-down traversal of the tree, check each overloaded operator. Let the tentative precision of the operator be p_t, and the precision expected by the parent† be p_e. Assign the wider of p_t and p_e as the precision of the operator.

This strategy is superior to strict evaluation; if it is more efficient than the widest available strategy for a particular target machine, then it should be the default. The effects of the other strategies can be obtained by using explicit coercion functions in those uncommon cases where they provide better results. Implementing the widest needed strategy is more difficult than the other strategies, but not terribly so; a modification to a UNIX FORTRAN compiler to implement the widest needed strategy required about 80 man hours[5] in the context of a compiler that was doing optimizations such as in-line expansions. When this strategy is designed into a compiler instead of added after the fact, the time needed should decrease.

IMPROVING EFFICIENCY

The ideal goal of an optimizing compiler is to increase the efficiency of a program without changing the output it produces. Unfortunately, the fact that floating-point arithmetic is normally viewed as an approximation to real or complex arithmetic has led some compiler writers to sacrifice semantics for the sake of speed because 'one approximation is as good as another'. But unknown approximations are not as good as known ones; it is possible to provide tight bounds on the error in many computations by careful coding that takes into account the differences between floating-point and real arithmetic. Therefore, do what the programmer says.

This rule is vitally important; it is broken by several optimizations that are legal under several language standards, but can be painful for the numerical analyst. Reorganizing expressions to use fewer registers is a well understood and common technique. As noted above, doing so can introduce anomalies since floating-point arithmetic fails to honour some common identities. Many languages allow arbitrary application of algebraic identities, regardless of their applicability to the machine arithmetic, so these optimizations do not fall under the heading of 'avoid at all costs'; but parentheses, at the very least, should be respected. Experienced programmers do not introduce parentheses without a good reason; inexperienced programmers do not run their programs often enough for such optimizations to be worth the effort.

Other optimization techniques involve moving computations, e.g. moving code out of loops, storing common subexpressions or evaluating constant arithmetic at compile time. Floating-point arithmetic can often have side effects aside from computing a result, e.g. setting flags or trapping.‡ If code is to be moved, it is important to ensure both that no spurious side-effects are introduced at the new location, and that the correct side-effects occur at the old location.

---

† The result of the operator will be either used in a context expecting a fixed precision (e.g. assignment to a variable or as a parameter) or as an operand of an overloaded operator. In the former case, the expected precision is defined by the language; in the latter, it is the precision assigned to the parent operator.

‡Recall that the decimal to binary conversion implicit in the appearance of a numeric literal is a full-fledged operation under the IEEE standard, and can generate signals and or be affected by rounding modes.

The optimizations listed above are dangerous, but floating-point code presents several opportunities for optimization. The most important ones are listed below.

A simple local improvement removes unnecessary coercions. When a cautious programmer has written

```
d := d + dble(s)•dble(s)
```

and the target machine supports a double precision product of two single precision numbers, there is no need to penalize the programmer with the extra coercions. The FORTRAN standard considers this example so important that it provides a special function enabling the programmer to perform this optimization at the source level; such a simple peephole optimization should be handled automatically by the compiler. Although it is simple, this optimization is quite important; a multiplication followed by an addition is the most common combination of operations in floating-point code.

A second important optimization deals with vector hardware. Machines with vector instructions cater to numeric processing, and any compiler that fails to vectorize loops will fall into disuse. Unfortunately, many compilers refuse to vectorize any loop with an embedded IF statement. Vectorizing many such loops is possible, albeit difficult, on machines with a select vector operation that chooses one of two possible results based on a boolean condition; the select operation can be used for embedded IF statements such as

```
if (x[i] = 0) then
    y := 1
else
    y := sin(x[i])/x[i]
endif
```

that are used to remove singularities from certain functions.

Branch prediction is another area where great improvement can be gained. Much of the branching in floating-point codes exists solely for the purpose of handling exceptional cases; the jump only needs to be taken a very small percentage of the time. This situation is ideal for branch prediction optimizations. Profiling information appears to be the best way to decide which branch is more likely, but a simpler implementation can provide a notation for the programmer to make the prediction.

Finally, division is almost always much slower than multiplication. Replacing division with multiplication is done by many programmers as a matter of habit; it is certainly within the compiler's jurisdiction to replace division by a constant with multiplication provided the reciprocal is exact and the multiplication has identical effects.

CONCLUSION

Floating-point programs must be faithfully translated if they are to produce meaningful results. The programmer must be able to predict the operations that will be executed, including all explicit and implicit rounding, in order to make useful statements about the accuracy of the result. To make this prediction possible, the compiler writer must often provide details that are missing in the language specification and forego common 'optimizations'.

By providing appropriate support tools, such as a source-to-source translator that inserts explicit precision conversions and an optimizer intended for floating-point code, an implementation can regain some of the portability and speed sacrificed in making a predictable compiler; but even without these tools, a compiler whose output produces correct results slowly is preferable to one that quickly produces misleading numbers.

ACKNOWLEDGEMENTS

This paper is primarily based on the experience of W. Kahan, transmitted via lectures and personal conversations at the University of California, Berkeley. Richard James provided a numerical analyst's wish list of what compiler writers ought to know that was useful in selecting key ideas. David Hough and the anonymous referees of *Software* provided helpful comments on the content and organization of earlier drafts of the paper. The author is supported in part by a U.S. NSF Graduate Fellowship and the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039–84–C–0089.

REFERENCES

1. W. Kahan and J. T. Coonen, 'The near orthogonality of syntax, semantics, and diagnostics in numerical programming environments', in *The Relationship Between Numerical Computation And Programming Languages*, North-Holland Publishing Company, 1982, pp. 103–115.
2. *IEEE Standard 754–1985 for Binary Floating-point Arithmetic*, IEEE, 1985. Reprinted in *SIGPLAN*, **22**, (2), 9–25 (1987) 1985.
3. W. G. Cody *et al.*, 'A proposed radix- and word-length-independent standard for floating-point arithmetic', *IEEE Micro*, **4**, (4), 86–100 (1984).
4. Jerome T. Coonen, 'Contributions to a proposed standard for binary floating-point arithmetic', *PhD Thesis*, University of California, Berkeley, 1984.
5. Robert P. Corbett, 'Enhanced arithmetic for Fortran', *SIGPLAN*, **17**, (12), 41–48 (1982).