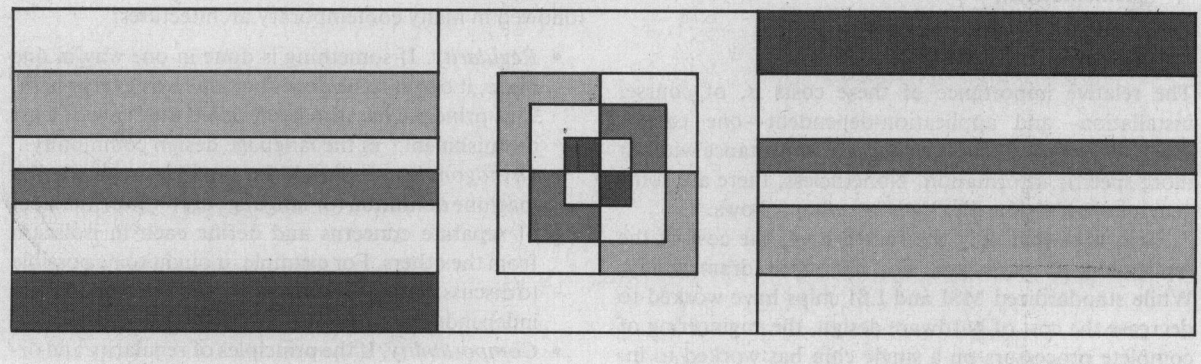
An examination of the relation between architecture and compiler design leads to several principles which can simplify compilers and improve the object code they produce.

Compilers and Computer Architecture

William A. Wulf
Carnegie-Mellon University



The interactions between the design of a computer's instruction set and the design of compilers that generate code for that computer have serious implications for overall computational cost and efficiency. This article, which investigates those interactions, should ideally be based on comprehensive data; unfortunately, there is a paucity of such information. And while there is data on the use of instruction sets, the relation of this data to compiler design is lacking. This is, therefore, a frankly personal statement, but one which is based on extensive experience.

My colleagues and I are in the midst of a research effort aimed at automating the construction of production-quality compilers. (To limit the scope of what is already an ambitious project, we have considered only algebraic languages and conventional computers.) In brief, unlike many compiler-compiler efforts of the past, ours involves automatically generating all of the phases of a compiler—including the optimization and code generation phases found in optimizing compilers. The only input to this generation process is a formal definition of the source language and target computer. The formulation of compilation algorithms that, with suitable parameters, are effective across a broad class of computer architectures has been fundamental to this research. In turn, finding these algorithms has led us to critically examine many architectures and the problems they pose. Much of the opinion that follows is based on our experiences in trying to do this, with notes on the difficulties we encountered.

Articles of this sort commonly begin by observing that the cost of hardware is falling rapidly while the cost of software is rising. The inevitable conclusion is that we ought to find ways for the hardware to simplify the software task. One area in which this might be done is in the design of instruction sets that better reflect the needs of high-level languages. Better instruction sets would both simplify compilers and improve the size and speed of the programs they generate. This observation and conclusion are absolutely correct. Treated too simplistically, however, they lead to mistaken inferences. For instance, many people have concluded that efficiency of the object code from a compiler is no longer important—or, at least, that it will not be in the near future. This is patently false. To suggest why I believe it is false and to lay the groundwork for much of what follows, let me illustrate with an example. Today I use a timesharing system that is five times faster than the one I used a decade ago; it also has eight times the primary memory and vastly more secondary storage. Yet, it supports about the same number of people, and response is worse. The reason for this anomaly is simply that our aspirations have grown much faster than technology has been able to satisfy them. It now takes more cycles, more memory, more disk—more everything—to do what a typical user wants and expects. I believe this is a good thing; despite its degraded performance, the system is more responsive to my overall needs—it's more humane. Nonetheless, there is a premium on the efficiency of the programs it executes.

The past is always our safest predictor of the future, at least if we interpret it properly. Although hardware costs will continue to fall dramatically and machine speeds will increase equally dramatically, we must assume that our aspirations will rise even more. Because of this we are not about to face either a cycle or memory surplus. For the near-term future, the dominant effect will not be machine cost or speed alone, but rather a continuing attempt to increase the return from a finite resource—that is, a particular computer at our disposal. This isn't a simple matter of "yankee thrift." Given any system, people will want to improve it and make it more responsive to human needs; inefficiencies that thwart those aspirations will not be tolerated.

0018-9162/81/0700-0041$00.75 © 1981 IEEE

The cost equation

Before discussing the mutual impact of compilers and target-machine architectures, we need to clarify the objectives of the exercise. A number of costs, and conversely benefits, are involved. They include

- designing (writing) compilers,
- designing the hardware architecture,
- designing the hardware implementation of that architecture,
- manufacturing the hardware (i.e., replicating the implementation),
- executing the compiler, and
- executing the compiled programs.

The relative importance of these costs is, of course, installation- and application-dependent—one cannot make statements about their relative importance without more specific information. Nonetheless, there are some general observations that bear on what follows.

First, note that only the fourth item, the cost of the replication of hardware, has decreased dramatically. While standardized MSI and LSI chips have worked to decrease the cost of hardware design, the engineering of complete processors on a single chip has worked to increase it again. Designing irregular structures at the chip level is *very* expensive.

Designing irregular structures at the chip level is *very* expensive.

Second, all of the design activities (compilers, architectures, and implementations) are one-time costs. From the customer's perspective, they are amortized over the number of units sold. Thus, it makes sense to design an architecture more responsive to compilation problems only if the reduction in the cost of writing the compiler, executing it, or executing the code generated by it, offsets the increase in the cost of design. Often this is easily done, as will be discussed later. However, we must remember that it is still more expensive to design hardware than to design software that does the same job, unless the job is very simple. This explains why it doesn't make sense, for example, to move the entire compiler into hardware.

Third, both software and architectures have a life substantially longer than that of the hardware technology for which they are initially implemented. Despite the predictable decline in hardware costs, too often architectures have been designed to cater to the anomalies of the technology prevalent at the time they were designed. In effect, there has been a confusion between the second and third costs listed above.

Finally, the last two items—the costs of compiling and executing programs—are not strictly comparable to those preceding them. Dollar costs can be assigned to these, of course. Often, however, the correct measure is not in terms of dollars but in terms of things that cannot be done as a consequence of compilation or execution inefficiencies. A typical installation can only occasionally acquire a new computer (and suffer the trauma of doing so). Between acquisitions the available computer is a fixed and finite resource. Hence inefficiencies in compilation and execution manifest themselves as decreased productivity, unavailable functionality, and similar costs. Although difficult to measure, these costs are the first-order effects noticed by users once a machine has been acquired.

Some general principles

Principles that would improve the impedence match between compilers and a target computer have been followed in many contemporary architectures:

- *Regularity.* If something is done in one way in one place, it ought to be done the same way everywhere. This principle has also been called the "law of least astonishment" in the language design community.
- *Orthogonality.* It should be possible to divide the machine definition (or language definition) into a set of separate concerns and define each in isolation from the others. For example, it ought to be possible to discuss data types, addressing, and instruction sets independently.
- *Composability.* If the principles of regularity and orthogonality have been followed, then it should also be possible to compose the orthogonal, regular notions in arbitrary ways. It ought to be possible, for example, to use every addressing mode with every operator and every data type.

However, these principles have not been completely followed, and the deviations from them present the compiler writer with some of his worst problems.

I will have a few more principles to suggest below. Before doing so, however, let me comment on the preceding ones.

Although many compiler optimizations are cast in terms of esoteric-sounding operations such as flow analysis and algebraic simplification—in fact these techniques are simply aimed at performing an enormous case analysis. The objective is to determine the best object code for a given source program. Because the case analysis is so enormous, the various optimization algorithms attempt to glean information and/or shape the intermediate form(s) of the program in a manner that allows the final case analysis and code selection to be done rapidly and thoroughly.

Viewing a compiler's task as a large case analysis helps to explain why regularity, orthogonality, and composability are so important to simplifying the task. Every deviation from these principles manifests itself as an ad hoc case to be considered. And, alas, the consideration of a special case is not limited to the code selection process; because every preparatory phase requires certain information intermediate representations, the manifestations of the anomaly creep back through nearly all of them. I will try to illustrate this with more examples in the next section. For the moment, however, consider the genre of general-register machines. The name suggests that the registers are all "general"—i.e., that they can be used for any of the purposes to which registers are put on the machine. In fact, however, almost no machine actually treats all the

registers uniformly—multiplicands must be in "even" registers, or double precision operands must be in even-odd pairs, or a zero in an indexing field of an instruction denotes no indexing (and hence the zeroth register cannot be used for indexing), or some operations can be performed only between registers while others can be performed only between a register and memory, or any of many other variations. Each of these distinctions is a violation of one or more of the principles above and results in additional complexity.

Some additional, more specific principles are

- *One vs. all.* There should be precisely one way to do something, or all ways should be possible.
- *Provide primitives, not solutions.* It is far better to provide good primitives from which solutions to code generation problems can be synthesized than to provide the solutions themselves.

Both of these also relate to the simplicity (or complexity) of the compiler's case analysis. Consider the "one-vs.-all" principle. Either of these extreme positions implies that the compiler need not do any case analysis. If, for example, the only conditional branching instructions are ones that test for EQUALITY and LESS THAN, there is only one way to generate code for each of the six relations. Alternatively, if there is a direct implementation of all six relations, there is an obvious coding for each. Difficulties arise, however, if only three or four of the six are provided. For example, the compiler must decide whether, by commuting operands, there is a cheaper implementation of some of the remaining relations. Unfortunately, this is not a simple decision—it may imply determining whether side-effect semantics are violated, whether there is an interaction with register allocation, and so on.

Now consider the "provide primitives, not solutions" principle. In what I believe to be an honest attempt to help compiler writers, some modern architectures have provided direct implementations of high-level concepts such as FOR and CASE statements and PROCEDURE calls. In many, if not most cases, these turn out to be more trouble than they are worth. Invariably they either support only one language well, or are so general that they are inefficient for special cases—thus forcing the compiler to perform even more analysis to discover the common cases where a more efficient implementation is possible. The problem arises from a "semantic clash" between the language and high-level instructions; by giving too much semantic content to the instruction, the machine designer has made it possible to use the instruction only in limited contexts. Again, I will try to illustrate the problem in the following section.

The last three principles are even more blatantly my opinion than those listed earlier:

- *Addressing.* Address computations are paths! Addressing is not limited to simple array and record accesses! The addressing modes of a machine should be designed to recognize these facts.
- *Environment support.* All modern architectures support arithmetic and logical computations reasonably well. They do not do nearly as well in supporting the run-time environments for programs—stack frames, displays or static/dynamic links, exceptions, processes, and so on. The writer should provide such run-time support.
- *Deviations.* The writer should deviate from these principles only in ways that are implementation-independent.

The first two of these principles, addressing and environment support, are among the most difficult to deal with—and among the most likely to run afoul of the earlier "primitives, not solutions" principle. The conscientious designer must remember that different languages impose different constraints on the notions of procedures, tasks, and exceptions. Even things as mundane as case and iteration statements and array representations are dictated by the semantics of the language and may differ from one language to another.

I will have more to say about these points later, but let me illustrate some of the issues with the problem of addressing. In my experience, effective use of implicit addressing is the most important aspect of generating good code. It is often the most difficult to achieve. In general, accessing a datum from a data structure involves following a path whose length is arbitrary (but is known at compile time). Each step along the path is an addition (indexing into arrays or records) or an indirection (through pointers of various sorts). Typical computers supply a fixed and finite menu—a collection of special cases—of such path steps. The compiler's problem is how to choose effectively from among this menu; I know of no technique for doing this except exhaustive special-case analysis. Even when the target computer's addressing modes are well-suited to the most common cases, its compiler remains complex since it must be ready to handle the general case.

Some architectures have provided direct implementations of high-level concepts. In many cases these turn out to be more trouble than they are worth.

The last principle is, I suppose, more in the nature of a plea. At any time there are technological anomalies that can be exploited to make a machine faster or cheaper. It is tempting to allow these factors to influence the architecture—many examples abound. It takes the greatest restraint to look beyond the current state of technology, to realize that the architecture will outlive that state, and to design for the future. I think that most of the violations of notions like orthogonality and regularity can be traced to a shortsighted view of costs. Adhering to the principles presented here has a measurable hardware cost to be sure, but one which decreases exponentially as technology changes.

Kudos and gripes

From the compiler writer's perspective, various machines have both good and bad features which illustrate—or violate—the principles discussed above. This is not

meant to be an exhaustive list of such features, nor is it intended to criticize particular computers or manufacturers.

On regularity. As a compiler writer, I must applaud the trend in many recent machines to allow each instruction operand to be specified by any of the addressing modes. The ability of the compiler to treat registers and memory as well as source and destination symmetrically is an excellent example of the benefits of regularity. The compiler is simpler and the object code is better.

Not all aspects of modern machines have been designed regularly, however. Most machines support several data types (including fixed and floating-point forms), several types of words (essentially boolean vectors), and several types of addresses—often with several sizes of each and sometimes with variations such as signed and unsigned and normalized or unnormalized. It is rare for the operators on these types to be defined regularly, even when it would make sense for them to be. A compiler, for example, would like to represent small integers as addresses or bytes. Yet, one machine provides a complete set of byte operations that are symmetric with word (integer) operations except that the ADD and SUB bytes are missing, and another defines the setting of condition codes slightly differently for byte operations than for full-word operations. Such differences prevent simple compilers from using the obvious byte representations. In more ambitious compilers, substantial analysis must be done to determine whether the differences matter in the particular program being compiled.

The familiar arithmetic shift instructions are another example of irregularity.

The familiar arithmetic shift instructions are another example of irregularity. I trust everyone realizes that arithmetic-right-shift is *not* a division by a power of two on most machines.

A particularly annoying violation of regularity arises from the instructions of machines that make special provision for "immediate mode" arithmetic. We know from analysis of source programs that certain constants, notably 0, ± 1, and the number of bytes per word, appear frequently. It makes good sense to provide special handling for them in an instruction set. Yet it seems that many machine designers feel that such instructions are useful only in forming addresses—or at least that they need not have effects identical to their more expensive equivalents. Manifestations include

- condition codes that are not set in the same way,
- carries that do not propagate beyond the size of an "address," and
- operations that are restricted to operate on a selected set of "index registers."

In practice, of course, $i = i + 1$ (and its implicit counterpart in iteration statements) is one of the most common source program statements. Irregularities such as those above preclude simple compilers from using immediate mode arithmetic for these common cases and add substantial complexity to more ambitious compilers.

Similar remarks apply to the floating-point instructions of many machines. In addition to providing an even more restricted set of operations, these machines often fail to support the abstraction of "real" numbers intended by many higher-level languages. (Someone once observed that the best characterization of the floating-point hardware of most machines is as "unreal" numbers!) Most language designers and programmers want to think of real numbers as an abstraction of real arithmetic (except for their finite precision)—they would like, for example, for them to be commutative and associative. Floating-point representation is often neither, and hence the compiler writer is constrained from exploiting some obvious optimizations. The cost is both increased compiler complexity and slower programs—and the sad thing is that the cases where the optimizations are illegal are rare in practice.

On orthogonality. By orthogonality I mean the ability to define separate concerns—such as addressing, operations, and data types—separately. This property is closely allied with regularity and composability. The failure of general-register machines to treat all their registers alike could be characterized as a failure of any of these properties. Several machines contain both long and short forms of branch instructions, for example, in which the short form is taken as a constant displacement relative to the program counter and is an addressing mode not available in any other kind of instruction. Some machines include instructions whose effect depends on the addressing mode used. For example, on some machines sign extension is (or is not) done depending on the destination location. Some other machines create long or short forms of the result of a multiplication, depending on the even-oddness of the destination register. Some of the most popular machines contain different instruction sets for register-to-register, memory-to-register, and memory-to-memory operations (and worse, these instruction sets partially—but not completely—overlap).

It should be clear that the compiler should perform a separate analysis for each of these cases. And unlike some of my previous examples, this requirement should apply to simple as well as ambitious compilers.

On composability. From the compiler writer's perspective, the ideal machine would, among other things, make available the full cross-product of operations and addressing modes on similar data types. The ADD instruction should have identical effects whether it adds literals, bytes, or words, for example. Moreover, any addressing mode available in one variant should be available in the others.

More germane to the present point is the notion of conversion. Many machines fail badly in this respect. For example, most only provide relational operators (i.e., conditional branch instructions) that affect control flow, while source languages generally allow relational expressions to appear in contexts where a boolean value must be

made manifest (e.g., allow them to be stored into a user's boolean variable). In addition, many machines do not provide for conversion between data types such as integer and floating point, nor do they provide a conversion that differs from that specified in source languages.

The root of the problem lies in the fact that programming languages view type as a property of data (or variables), while machines view type as a property of operators. Because the number of machine data types is moderately large, the number of operation codes needed to implement a full cross-product is unreasonable. For example, it is usually impossible to add a byte to a word without first converting the byte to full-word form. The need for such explicit conversions makes it difficult to determine when overall cost is reduced by choosing a particular representation. Admittedly, where the conversion is a significant one (as in converting integer to floating-point representation) this doesn't feel so bad—but it adds complexity to the compiler as well as slowing both compilation and execution in trivial cases.

I will end this discussion with an example of the complexity imposed on a compiler by the lack of regularity, orthogonality, and composability. Consider a simple statement such as A := B*C and suppose we are compiling for a machine on which the operand of a multiply must be in an odd register. A simple compiler generally allocates registers "on the fly" as it generates code. In this example, such a strategy appears to work well enough; the allocator requires only minor complexity to know about even/odd registers, and the code generator must specify its needs on each request. But in a trivially more complex expression such as A := (B + D)*C, the strategy breaks down. Addition can be done in either an even or odd register; a simple "on the fly" allocation is as likely to get an even register as an odd one for B + D—and, of course, the choice of an even one will necessitate an extra data move to implement the multiplication by C. More ambitious compilers must therefore analyze a complete expression tree before making any allocations. In trees involving conflicting requirements, an assignment must be found that minimizes data movements.

Ambitious compilers can even have a problem with the simple assignent A :+ B*C. Such compilers often employ a technique called "load/store motion" in which they attempt to move variables accessed frequently in a program region into registers over that region; this tends to eliminate loads and stores. The simple assignment above suggests that it would be good to have A allocated to an odd register, since the entire right-hand side could then be evaluated in A and another data move eliminated. Whether this is desirable, however, depends on all the other uses of A in the program region under the register in which A resides. This involves more complex analysis than that for single expressions and, again, may require trade-offs in the number of data moves needed.

Note that such complexities arise in other contexts besides even/odd register pairs. At least one machine has distinct accumulators and index registers plus a few elements that can be used as either. Precisely the same sort of compiler difficulties arise in deciding where the result of an arithmetic expression or user variable should go; the compiler must examine all uses to determine whether the result is used in an indexing context, an arithmetic context, or both.

On one vs. all. I am sure most readers who are familiar with a number of machines can supply examples of violations of this principle. My favorite can be found in one of the newest machines—one with a generally excellent instruction set. This set includes reasonably complete boolean operations, but does not provide AND, instead providing only AND NOT. AND is commutative and associative, but AND NOT is not, so it requires a truly bewildering analysis to determine which operand to complement and when to apply DeMorgan's law in order to generate an optimal code sequence.

On primitives vs. solutions. Most people would agree that Pascal is more powerful than Fortran. The precise meaning of "more powerful" may be a bit unclear, but certainly any *algorithm* than can be coded in Fortran can also be coded in Pascal—but not conversely. However, this does not mean that it is easy, or even possible, to translate Fortran *programs* into Pascal. Features such as COMMON and EQUIVALENCE are not present in Pascal—and some uses of them are even prohibited by that language. There is a "semantic clash" between the languages.

A machine that attempts to support all implementation requirements will probably fail to support any of them efficiently.

The same phenomenon can be observed in machine designs. Among the common higher-level languages one finds many different views of essentially similar concepts. The detailed semantics associated with parameter passing, FOR statements, type conversions, and so on are often quite different. These differences can lead to significantly different implementation requirements. A machine that builds-in instructions satisfying one set of these requirements cannot support other languages. A machine that attempts to support all the requirements will probably fail to support any of them efficiently—and hence will provoke additional special-case analysis in the compiler. Examples of the misplaced enthusiasm of machine designers include

- subroutine call instructions that support, for example, only some parameter passing mechanisms,
- looping instructions that support only certain models of initialization, test and increment, and recomputation,
- addressing modes that presume certain stack frame layouts—or even presume particular representations of arrays,
- case instructions that only do or don't implicitly test the boundary conditions of the case index and only do or don't assume such bounds are static at compile time.

- instructions that support high-level data structures (such as queues) and make assumptions that differ from some common implementations of these structures, and
- elaborate string manipulation.

In many of these cases, the high-level instructions are synthesized from more primitive operations which, if the compiler writer could access them, could be recomposed to more closely model the features actually needed. An ideal solution to this problem would provide a modest amount of writable microstore for use by the run-time system. The compiler writer could then tailor the instruction set to the needs of a particular language.

On addressing. Modern programming languages permit the definition of data structures that are arbitrary compositions of scalars, arrays, records, and pointers. At least in principle it is possible to define an array of records, a component of which is a pointer to a record containing an array of arrays of yet another kind of record. Accessing a component at the bottom level of this structure involves a lengthy sequence of operations. Further, because of the interactions among block structure, recursive procedures, and "by reference" parameter passing, finding the base address of such a structure can be equally complex—possibly involving indexing through the "display," various sorts of "dope" (descriptor) information, and several levels of indirection through reference parameter values. In practice, of course, data structures are seldom this complex, and most variables accessed are either local to the current procedure or global to the entire program—but the compiler must be prepared to handle the general case. And, surprisingly, even relatively simple constructs such as accessing an element of an array in the current stack frame can give rise to much of this complexity.

The algorithm to access an element of a data structure can be viewed as walking a path from the current invocation record to the element; the initial portion of the path locates the base address of the structure via the display, etc., and the final portion is defined by the structure itself. Each step of the path involves indirection (following a pointer), computing a record element displacement, or indexing (by an array subscript)—all of which may involve multiplication of a subscript by the size (in address units) of the array component. For many languages, constraint checks on array subscripts and nil pointers must also be performed along the path.

It is clearly advantageous to use the target computer's effective address computations to implicitly perform as many of the path steps as possible. Unfortunately, most contemporary machines were designed with a simpler data structure model in mind. Rather than supporting steps along a general path, these machines generally provide an ad hoc collection of indexing and indirection that can be used for only a subset of such steps—there is no notion of their composition. For example,

- one of the most popular machines has no indirection at all, thus forcing an explicit instruction each time a pointer must be followed;
- most machines that provide both indexing and indirection define a fixed order—e.g., first do the indexing, then the indirection, although the opposite order is just as common in practice;
- some modern machines provide implicit multiplication of one operand of an indexing operation, but only by the size of scalars (whereas in general the element of an array may not be a scalar); and
- many machines limit the size of the literal value in an indexing mode, thus forcing contorted code for larger displacements or for cases where the displacement isn't known until link time.

The addressing modes are useful for some, but never all, of these cases. Worse, sometimes more than one mode can be used and the compiler must make a nontrivial choice. Again, the compiler must be made more complex to exploit the hardware. It would be far better (and probably simpler in the hardware as well) to have a more general and consistent model.

On environments. I believe that most people now appreciate hardware support for recursive procedure invocation, dynamic storage allocation, and synchronization and communication processes. I won't comment on this except to note the danger of providing such support at too high a level and hence introducing a semantic clash with some languages. Instead, I will point out some neglected areas of the support environment that induce significant overheads in the implementation of at least some languages. These include:

- *Uninitialized variables.* Many languages define a program to be erroneous if the value of a variable is fetched before being set. The code and data structures to support this are expensive. Yet at least one old machine provided a way to check these "for free" by setting bad parity on uninitialized variables.
- *Constraint checks.* Many languages specify that certain properties of a value be checked before use. Subscript range checking is a particular instance of this, but there are many other cases as well. Generally, machines provide no direct implementation of this, thus forcing explicit tests in the code and often eliminating clever uses of the effective address hardware.
- *Exceptions.* Exceptions (analogous to PL/I's ON conditions) are a part of many languages. Yet few machines support them, and an efficient software implementation of them often violates the hardware assumptions of the support provided for procedures.
- *Debugging support.* If most programming is to be done in high-level languages, then it is axiomatic that debugging should be at the same level. Because most machines do not support this, however, the designer of the debugging system is usually faced with two choices, both unpalatable. The first is to force the user to debug at a low level. The second is to create a special debugging mode in which extra code provides the run-time debugger with the necessary information—this in turn makes it difficult to debug the production version of a system.

A note on stacks. A common belief is that all compiler writers prefer a stack machine. I am an exception to that belief—at least insofar as stacks for expression evaluation are concerned. (Stacks to support environments are a different matter.) It is certainly true that there is a trivial mapping from parse trees to postfix, and hence simple compilers can be made even simpler if their target has an expression stack. For more ambitious compilers, however, stack machines pose almost all the same optimization problems as register machines. A common subexpression is still a common subexpression. A loop invariant expression is still invariant. Expression reordering, done on a register machine to minimize the number of registers used, also reduces the depth of the evaluation stack—thus increasing the likelihood that the entire computation can be held in the fast top-of-stack registers. Even register allocation has a counterpart—i.e., allocating variables to the stack frame so that the most frequently accessed ones can utilize short addressing forms on machines that provide such a feature. Moreover, deciding whether an optimization is desirable can be more difficult on a stack machine. On a register machine, for example, it is almost always desirable to save the value of a common subexpression, while on a stack machine it is necessary to determine whether the added cost of storing and retrieving the value is offset by that value's uses. Thus, while expression stacks are nice for simple compilers, they are in no sense a solution.

A note on interpreters and writable microcode. An increasingly popular approach, especially in the microprocessor domain, is to microprogram a special-purpose interpreter for a given source language. This has been done extensively for Pascal, for example. As noted above, in general I prefer "primitives, not solutions" (especially the wrong solutions); tailoring the instruction set through microprogramming is one way to achieve that.

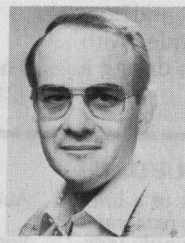
There are problems with this approach, however. It implies that we must find ways to ensure that user-written microcode cannot subvert operating system protection. Similarly, we must provide a means for dynamically associating the right interpreter with a given process, and this may imply substantial context-swap overheads unless one is very careful. (I am making an assumption here. Namely, I believe that multiprogramming, multiprocessing, and protection will all become common in the microprocessor world—essentially in forms evolved from what we see in current large systems. I also believe that single-language systems are not a solution; we must assume a multilanguage environment. The rationale for my beliefs is not appropriate to this article, but the most recent announcements from several chip manufacturers corroborate my views.)

Above all, the approach is not a panacea! In at least one case I have examined, the code produced for such an interpreter is roughly two times slower and larger than it needs to be. In another case, a sophisticated optimizing compiler had to be built just to get reasonable performance.

Both the compiler writer and the machine designer have multiple objectives. The machine designer should certainly attend to the concerns of high-level-language implementation—since most programs will be written in one—but he must also attend to cost, reliability, compatibility, customer acceptance, and so on. The compiler writer must faithfully implement the semantics of the source language, must provide a friendly interface to the user, and must try to reach a reasonable compromise between the speed of compilation and the quality of the resulting object code. Being realistic, both designer and writer know that they can affect only a subset of these objectives on each side. However, they also know that they can still do a great deal to improve the impedence match between languages and machines and so reduce total costs. ■

Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency and monitored by the Air Force Avionics Laboratory. The views and conclusions contained in this article are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US government.



William A. Wulf is a professor of computer science at Carnegie-Mellon University. Prior to joining CMU in 1968, he was an instructor of applied mathematics and computer science at the University of Virginia. His research interests span the fields traditionally called "programming systems" and "computer architecture." He is especially interested in the construction of large systems, notably compilers and operating systems, and in the way the construction of these systems interacts with the architecture of the machine on which they run. Wulf's research activities have included the design and implementation of the Bliss system implementation language, participation in the PDP-11 design, construction of C.mmp—a sixteen-processor multiprocessor computer, the design and implementation of the Hydra operating system for C.mmp, the design of the Alphard programming language, and participation in the development of Ada, the DoD language for embedded computer applications.

Wulf holds the BS in physics and the MSEE from the University of Illinois and the PhD from the University of Virginia.