BLISS/11: A Lesson in Object Code Optimization

Gary L. Luckenbaugh

Department of Electrical Engineering
University of Delaware, Newark, Delaware 19711

ABSTRACT

Optimizing compilers are examined as an alternative to systems programming in assembly language. The Bliss/11 compiler is studied as an example of a good optimizing compiler with many optimization strategies. These strategies are then tested to find out which ones are likely to have the highest payoff.

INTRODUCTION

At our installation in the University of Delaware, Electrical Engineering Department, we write a large number of systems programs. As an alternative to assembly language, most of our programs are written in a high level language called C [1]. Virtually all of these programs are for mini or micro computers such as the PDP-11, IBM Series/1, and M6800. Because of this, it is important that the object code produced by the compilers be as efficient as possible in terms of memory utilization and execution time. As a means of determining if it would be worth the effort to improve our current compilers, we studied the Bliss/11 compiler as an example of a good optimizing compiler.

OVERVIEW OF BLISS/11

The language Bliss was originally implemented by Wulf et. al. of Carnegie-Mellon University on the PDP-10 computer [2]. Bliss was designed to be used as an alternative to programming in PDP-10 assembly language. As such, it was important that the language have many of the characteristics of a powerful high level language without losing some of the intimate characteristics of assembly language. It is equally important that the language produce good object code.

Bliss/11 is a dialect of Bliss for the PDP-11 computer. Bliss/11 is an Algol like language with some unique characteristics [4]. For example there are no data types in Bliss. Declarative statements in Bliss only reserve blocks of memory that can be used for any purpose. The types of operations performed on these data items determine what they are. An identifier used in Bliss evaluates to the address of the data item. To obtain the contents of a location, the contents-of operator, signified by a dot, must be used. As a result of this, the assignment operator is a binary operator which takes the value of the expression on it's right and stores it in the location indicated by the value of the expression on the left [5]. Thus, a statement such as X=X+1 written in Fortran would be written X=.X+1 in Bliss. The language also allows the programmer to declare his own addressing formulas for addressing arrays or other structures and has the capability of operating on bit fields within words.

Bliss is an expression language. All procedures are composed of a single expression. Unlike APL or LISP, expressions are built from the familiar control structures of Algol, such as: the IF-THEN-ELSE, and WHILE-DO constructs. The value returned by the WHILE-DO construct would be the value returned by the expression following the DO the last time the loop was executed. The GOTO statement is not implemented in Bliss because it is meaningless to jump into the middle of an expression. The language does, however, allow loops to be exited prematurely with the LEAVE statement. This is a restricted form of a forward branch.

Most of the characteristics of the language were chosen so that the compiler could produce highly optimal code [3]. The compiler is primarily concerned with improving memory utilization; however, if the code is shorter, in most cases, it will also execute faster. The reason they chose to make Bliss a typeless language was so that the compiler could view all data as being the same. This makes it easier to optimize simple expressions since no special cases can exist. Similarily, the reason for making Bliss an expression language is so that the compiler could easily analyze the flow of control in a program. The GOTO statement was omitted from the language to aid flow analysis.

THE BLISS/11 COMPILER

The Bliss/11 compiler can be viewed as seven major phases [3]. The input to each phase can be thought of as a chunk of code written in a given language. Each phase of the compiler translates the code from one language to another and then passes it's output to the next phase in the sequence. The name of the seven phases are LEXSYNFLO, DELAY, TLA, RANK, PACK, CODE and FINAL. Each phase is concerned with a particular type of code optimization. The LEXSYNFLO phase performs lexical analysis, syntactic analysis, and global flow analysis. The DELAY phase determines the order of evaluation of expressions and the general shape of the object code. The TLA, RANK, and PACK phases are primarily concerned with register allocation. The CODE phase produces the object code and also looks for some minor optimizations that can be made. The last phase of the compiler called the FINAL phase performs various ad hoc peephole optimizations. Peephole optimization cleans up the assembly language output produced by the CODE phase.

The Lexsynflo Phase

The LEXSYNFLO phase is the first phase of the compiler [3]. Besides performing the lexical analysis and syntactic analysis of the input program, it is also responsible for global flow analysis. In general, this involves looking at the overall flow of control in the program and making decisions on how the code can be re-arranged in order to decrease the amount of object code produced. One of the things that this phase looks for are common sub-expressions that can be moved. A common sub-expression is an expression or piece of an expression that appears more than once in the program. As long as the contents of the variables don't change between two appearances of the expression, then the two expressions must always evaluate to the same result. Thus, it is only necessary to evaluate the expression once. A common sub-expression may be as simple as a single variable or as complex as a large arithmetic formula. A similar situation occurs with loop invariants. Loop invariants are expressions whose value does not change during repeated executions of a loop. Such expressions can be evaluated before starting the loop and their values stored in temporary registers for use within the loop.

Another form of optimization that the LEXSYNFLO phase performs is called code motion optimization. This type of optimization can be performed when a common sub-expression appears in all the branches of an n-way branching structure. If the value of the expression is the same regardless of which path is taken, then the expression can be evaluated before the branch is taken. This will decrease the amount of code produced since it only needs to appear once instead of n times. Similarly, if a common sub-expression appears in all n branches it may be evaluated after the point where all the n branches rejoin. In order to do this, the contents of the variables used in the expression must be the same at the point where the n branches rejoin as they were where the expression was originally located. This also implies that the value of the expression was not further referenced until the point where the n branches rejoin.

The Delay Phase

The DELAY phase of the compiler is responsible for a variety of optimizations [3]. The primary purpose of DELAY is to determine which of the optimizations detected during global flow analysis are really desireable. For example, a common sub-expression such as .A is very simple to evaluate and it may be more advantageous not to treat it as a common sub expression.

Another responsibility of the DELAY phase is to change the order of evaluation of an expression in order to shorten the number of instructions needed to find the value. DELAY will also attempt to algebraically simplify the expression where possible. Care must be taken in doing this, of course, since computers have fixed precision. Some of the identities used for such simplification are as follows: $x-y = x+(-y)$, $x-y = -(y-x)$, $x*y = (-x)*(-y)$, and $a \wedge b = \sim(\sim a \vee \sim b)$. The DELAY phase will also perform simple optimizations, such as: turning multiplications into shifts where possible, and distributing multiplications if this would be advantageous.

The Tla, Rank, Pack and Code Phases

The TLA, RANK, and PACK phases together are responsible for register allocation [3]. They determine which variables in a subroutine are used enough to merit being stored in a register. They also determine which registers should be used as temporaries.

The CODE phase of the compiler is primarily responsible for producing the assembly language output from the parse trees and results of the previous phases. The code generator does have some intelligence, however, so that the object code produced for a given operator may vary depending on the context in which it is used.

The Final Phase

The FINAL phase of the compiler performs peephole optimizations on the assembly language code produced by the CODE phase [3]. The purpose of the peephole optimizer is to clean up the resulting code. Many of the things that can be done during peephole optimization are not done by the Bliss compiler because it finds them in previous phases. The rather ad hoc activities of the peephole optimizer will be described in terms of the Unix C compiler since it is a richer optimizer.

One of the things that a peephole optimizer can look for are jumps to jumps. If a jump instruction that jumps to another jump instruction is found, it can be modified to jump directly to the destination. Similarly, if a jump instruction is found that jumps to the next location, it can be removed altogether. The C compiler looks for impossible to reach code and removes it. This is not a problem with Bliss/11 since the nature of the language, precludes the generation of such structures.

On the PDP-11 computer several optimizations of the jump and branch instructions can be made during peephole optimization. For example, short jumps can be changed to branch instructions. On the PDP-11 this will save one word of memory. Conditional branch instructions which skip over unconditional branch instructions can be changed to a single conditional branch of the opposite sign.

If two jump or branch instructions have the same destination address and they are preceded by identical blocks of code, then one of the blocks of code can be removed and a jump to the second block put in it's place. A similar type of optimization can be made when a block of code preceding a jump instruction is identical to a block of code preceding the destination address. This is called a cross jump and can be optimized by deleting the block of code preceding the jump instruction and altering the destination to point to the identical block before the old destination.

The peephole optimizer also looks for single instructions which may be replaced with similar instructions which take less space. For instance, on the PDP-11 an instruction such as MOV #0, R0 can be changed to CLR R0. Also, given certain assumptions about the object code, it may also be possible to change an instruction such as ADD #2, SP to the instruction TST (SP)+. Many replacements such as these can be made, but care must be taken if the condition codes are checked after such an instruction.

TEST RESULTS

In order to examine the effects of the various optimization strategies, a program from the Bliss/11 Programmer's Manual was compiled several times with different optimization strategies enabled [4]. The compiler has three switches which control optimization. The F switch is used to control the final peephole optimization phase, the O switch controls common sub-expression elimination and code motion optimizations, and the Q switch controls some special cases of common sub-expression elimination. With the O switch turned off most of the optimization done by the compiler is disabled. The compiler will still check for common sub-expressions within a statement but will not check for common sub-expressions between statements. The Q switch will enable common sub-expression elimination even in the case where one of the variables in the expression may change as the result of being addressed indirectly through some other variable. If the Q switch is not on, it will only eliminate expressions in which it can guarantee the variables will not change.

The following table shows the results of the test. The left column shows which switches were turned on and the right column shows the size of the object code in words. The size of the source program is approximately 200 lines of Bliss/11 code.

TABLE I

CODE SIZE VS. OPTIONS

Switches	Size
None	386
Q	386
F	368
FQ	368
O	363
OF	346
OQ	336
OFQ	326

From the results of the tests, it is not apparently obvious which optimization strategies are really improving the code. One possible explanation for this may arise from the amount of optimization performed by the CODE phase and optimizations on simple expressions, both of which can not be disabled. In other words, the optimization strategies that can be turned off apparently don't really perform the bulk of the work.

Apparent lack of optimization pay off by the peephole optimizer is frequently observed in situations where the original code generation strategy was optimized. An example of this is found in the C compiler for the PDP-11 and the locally developed M6800 C compiler. The PDP-11 C compiler does a fairly good job of optimizing the object code during the code generation phase, and thus, the peephole optimizer does not find too many things can be changed. On the other hand, the M6800 C compiler does very little optimization during the code generation phase, and thus, the peephole optimizer can shorten the object code considerably.

Since the object code size decreased by 15% with all optimizations turned on and the next best result was only about a 9% improvement, the indication is that there isn't really an advantage to using only a subset of the optimization strategies. Probably the most optimistic solution to the problem is for the programmer to take some of these optimization strategies into consideration when he writes a program. In other words, if you know something about the compiler you are using, it may be possible to help the compiler produce better object code by writing better source code in the first place. A simple example of this coding practice would be an IF statement in which the same expression appears in both the THEN clause and the ELSE clause. An expression such as this could be calculated and stored in a temporary variable before the IF statement.

CONCLUSIONS

Bliss/11 is an excellent optimizing compiler; however, in our case it is probably not worth the time and effort to extensively modify the existing C compilers to make use of most of these optimization techniques. The exception and the one component that is relatively easy to add to any compiler is the peephole optimizer. Peephole optimization can be done piecemeal, produces measurable results and time spent improving existing peephole optimizers is likely to yield a return. Some of the optimization strategies used by Bliss/11 in previous phases of the compiler can be incorporated in the peephole optimizer. For very critical sections of code, the source code or even the resulting object code could be optimized by hand, taking into consideration some of the lessons from Bliss/11.

REFERENCES

[1] Kernighan, B. W., & Ritchie, D. M., *The C Programming Language*, Prentice-Hall, 1978.

[2] Wulf, W. A., Russell, D. B., & Habermann, A. N., "BLISS: A Language for Systems Programming", CACM, Vol. 14, No. 12, December 1971.

[3] Wulf, W. A. et al, *The Design of an Optimizing Compiler*, Elsevier Computer Science Library, 1975.

[4] Bliss-11 Programmer's Manual. Digital Equip. Corp., Maynard, Mass. 1974.

[5] Wulf, W. A., "Why the DOT?", Decus No. 10-118 Part II, Digital Equip. Corp., Maynard, Mass. 1971.