Compiling Forth for Performance

Thomas Almy
17830 SW Shasta Trail
Tualatin, Oregon 97062

Abstract

Conventional Forth environments use threaded code interpretation techniques. While this makes compilation fast, and keeps the compiler size extremely small, the execution performance tends to be five to ten times slower than that obtainable with compiled languages. Subroutine threaded code, combined with more intelligent compiling words can increase performance markedly on some systems, and several companies sell code optimizers which turn colon definitions into code words either by copying primitives to make inline code or by more sophisticated techniques.

By compiling the complete application, many code optimizations are possible that are not available to mixed systems. For instance, no memory space is allocated for constants, and colon definitions become machine language subroutines. Compiled programs run faster than equivalent programs compiled in C or Pascal, yet have the same code compactness of conventional Forth. Additionally, the compiler compiles faster than most C or Pascal compilers. Performance on an IBM/PC was measured to be roughly 0.25 MOPS (using the sieve benchmark), or about 19 clock cycles per Forth primitive.

The paper will discuss the user interface and implementation concepts of a Forth compiler, and will give examples of some possible optimizations that can be performed.

Overview

Forth has mainly been viewed as an interactive, interpreted, programming environment. This article discusses some design features and decisions of CFORTH, a non-interactive, native-code Forth compiler. The CFORTH compiler is invoked as a DOS command, and produces an executable `.COM` file.

Selective compilation of colon definitions into native machine code was first reported by Tom Dowling of Miller Microcomputer Services [DOW81]. The first commercially available compilers appeared with the author's Z-80 and Cascade Forth systems in 1982, and became widely available as the "Native Code Compiler" from Laboratory Microsystems. Another compiler, "AUTO-OPT" is available from Harvard Softworks. More recent published efforts include a 68000 compiler by Anthony Rose [ROS86], and the compilers for the NOVIX chip [GOL85] [MOO85].

While the compiler exists in both Z-80 CP/M and 8086/8 MS-DOS versions, most of the examples shown here are for the 8086/8 when the distinction is necessary (namely when discussing the generated assembly language code).

The User Interface

Source programs for CFORTH can either be screen files (matching traditional Forth usage) or text files (matching traditional compiler usage). For screen files, screen 1 is automatically loaded, and is expected to be a load screen for the application. Text files may contain tabs and form feeds. Either type of source file can have `INCLUDE` statements to load libraries or other source files.

The user invokes the compiler with the command `CFORTH filename options`, where `filename` is the name of the source file and `options` is a string to interpret. Typical compiler and linker options, such as libraries to use, printing or deletion of load map, and the memory model, are placed either in the source file or on the command line.

CFORTH is a single pass compiler which creates a `.COM` or `.EXE` file without any separate assembly or link passes as is found in most other compilers. Except as mentioned later in this article, all definitions in the source are compiled into the `.COM` file, referred to as the "target". Execution occurring in interpret state happens in the compiler, referred to as the "host".

A typical load screen looks like the following:

```
1000 SEPSSEG       \ ''linker'' directive specifying a separate
                   \ stack segment 1000 bytes in length.
100 MSDOS          \ ''linker'' directive specifying a .COM file with
                   \ 100 byte return stack. This command also
                   \ generates the code preamble (about 30 bytes)
                   \ that initializes the program. Other options are
                   \ for .EXE file (separate code and data segments)
                   \ or romable file (not for MS-DOS execution).
INCLUDE VARS       \ load file containing definitions of all Forth 83
                   \ variables.
2 50 THRU          \ load application screens.
INCLUDE MYLIB      \ library of words created by the programmer,
                   \ selectively compiled to resolve any unresolved
                   \ forward references.
INCLUDE FORTHLIB   \ Library of 83 Standard words selectively
                   \ compiled to resolve any unresolved forward
                   \ references.
END                \ Writes file and produces load map.
```

Notice that the libraries and code preambles are all in source code form. The compiler runs sufficiently fast that it was decided that a true linker and object files were unnecessary. The programmer can also modify the libraries at his or her own risk. For instance, CP/M 86 `.CMD` files could be generated without modifying the compiler.

Interpreting in the Compiler

Unlike other language compilers, but like existing Forth interpreter environments, Forth code can be executed at compile time, typically for creating data structures. Words can be written which will execute on the host, by starting the definition with `H:` rather than `:`. These user-defined words plus most of the 83 Standard Forth wordset [FOR83] are available. Memory referencing words such as `@ ! CMOVE` and `FILL` always refer to the target program's address space, rather than the compiler's. The dictionary lookup functions `' [']` and `FIND` always return the address of target words. Target variables, constants, arrays, and table words are all executable during compilation, so that they can easily be initialized, or used as temporary data storage during compilation.

An interpretable, nestable `#IF..#ELSE..#THEN` statement allows for conditional compilation so that many versions of the same program can share the same source.

Usage of Registers

An important consideration in the design of any compiler is how to make best use of the registers and other architecture features of the processor. Ideally one wants to minimize the number of memory references by keeping the most used values in registers. The Forth language guides the register optimization; CFORTH buffers the top of stack values in registers. This assignment can be done at compile-time very cleanly since only well-formed control structures are allowed (languages which allow GOTO type statements have trouble with register optimization). In the current implementations, up to the two top of stack values may be in registers at any time. Registers AX and BX are used for this purpose. Register CX contains the innermost loop index in a `DO LOOP` structure. CS, DS, and SS point to the (possibly different) segments for code, data, and stacks, while SP and BP are the parameter and return stack pointers. Register SI sometimes contains the return address and registers DX, DI, and ES are used as temporaries—all the registers are utilized. (Z-80 users make following substitutions: AX->HL BX->DE CX->BC SI->IY BP->IX SP->SP. DX,CS,DS,SS,ES->no equivalent, A is temporary.)

Classes of Words

In an interpreted Forth environment, all words are treated identically by the text interpreter/compiler, except that `IMMEDIATE` words are always executed rather than being compiled. If CFORTH used the same technique, then the generated code would be what is commonly called "subroutine threaded", which gives increased performance at a space penalty, but is becoming popular with 68000 based systems. In order to greatly improve performance and code density over subroutine threaded code, CFORTH divides words into many different classes, each of which is handled differently. These classes are subroutines, data, literals, arrays and tables, intrinsics, and `DOES>` words.

Subroutines – Colon and code definitions

Colon definitions are compiled as machine language subroutines. The individual words within the colon definition are compiled in accordance with their class. Colon definitions may be "ticked" and `EXECUTE`d at target execution time.

Code words are essentially the same as those in a Forth interpretive environment, except they end with a subroutine return instead of a jump to the threaded code interpreter. One advantage of CFORTH is that colon definitions can be called from code words without having to switch back to threaded code.

The current implementations use the machine stack as the parameter stack. This means potential awkwardness in accessing a value on the stack since the return address would be in the way. Normally the return address is saved on the return stack during the colon definition's execution, but two optimization techniques are provided for the programmer. `PRIMITIVE` causes a dedicated register to be used to hold the return address; subroutines may not be called from within a primitive word. The other optimization, `IN/OUT` lets the programmer specify how many parameter and result values are passed to/from the word. If the values are both two or less (in the current implementations) then the values are passed in registers, and the return address is left on the stack.

To illustrate most of the above points, this is a code word which prints two times the value of its argument. We will assume that "`.`" has been defined, and has been specified as `1 Ø IN/OUT`.

```
1 Ø IN/OUT    \ Parameter passed in AX
CODE 2*.      AX AX ADD              \ 2*
              ' . CALL               \ Call .
              RET END-CODE           \ Return nothing
```

A wise assembly language programmer could improve this by replacing the `CALL` and `RET` instructions with `JMP`, and in fact CFORTH would do this if our definition had been `: 2*. 2* . ;` .

Data Address – `CREATE` and `VARIABLE`

One important distinction of CFORTH is that all variables are treated as literals. This means that there is no explicit code for variables—only the data area is allocated in the target, there is no "code field". Variables may be executed while in interpret state since they are defined as constants in the host. But, because they are literals, they may not be `EXECUTE`d.

Manifest Constants and Literals — `CONSTANT 2CONSTANT` and Numbers

Declared constants in CFORTH generate no code and take no data space in the target. Instead, they behave just as numeric literals. This means that constants may not be ticked, nor may values be stored into them. A common trick in many Forth applications, changing the value of a constant at execution time won't work if the application is placed in ROM, and is a direct violation of the 83 Standard as well.

While compiling a target colon definition, literals are held on a "literal stack" until needed. This means that most literal arithmetic is performed at compile time rather than execution time. For example:

```
VARIABLE X
20 CONSTANT Y
: FOO X DUP Y + SWAP DO ... LOOP ;
```

generates the same code as

```
VARIABLE X
X 20 + CONSTANT Z
: FOO Z X DO ... LOOP ;
```

Arrays — `TABLE` and `ARRAY`

Unlike the data types `VARIABLE` and `CREATE`, `ARRAY`s and `TABLE`s generate in-line code, which is used to handle indexing. They may also be accessed while in interpret state. The equivalent Forth definitions of these defining words are:

```
: ARRAY CREATE 2 * ALLOT DOES> SWAP 2 * + ;
: CARRAY CREATE ALLOT DOES> + ;
: TABLE CREATE DOES> SWAP 2 * + @ ;
: CTABLE CREATE DOES> + C@ ;
```

The compiler does as much of the operation at compile time as possible. For instance, with the declarations:

```
TABLE FOO 10 , 20 , 90 , -200 , 60 30 * ,
10 ARRAY BAR
```

The compiled sequence `3 FOO` becomes the literal `-200`, and the sequence `10 4 BAR !` becomes just a single machine instruction `10 # ' BAR >BODY 8 + [] MOV`.

Intrinsics — Many, Many Words

Most of the 83 Standard word set, and many other words besides, are built into the compiler and generate in-line machine code rather than subroutine calls. These words, roughly 120 of them, are called "intrinsic" functions. Such a concept is not new to Forth, indeed compiling words do just this.

All standard words for which efficient in-line code sequences can be generated are intrinsics. Any word which needs more than a few instructions is placed in the library, `FORTHLIB`, and is accessed via a subroutine call. For instance, in the Z-80 version, all the multiply and divide operations are in the library, but in the 8086/8 version, only the multiplication and division words needing signed divide (`/ MOD /MOD */` and `*/MOD`) are in the library. Because of the much more powerful instruction set in the 8086/8, more words are intrinsic.

Certain additional non-standard words were made CFORTH intrinsics. Because it is impossible for users to write their own compiling words (existing Forth interpreter words would never work, anyway), some additional popular words were added, such as a case statement, `?DO`, `?LEAVE`, and string literal words. A full set of comparison functions such as `<=` and some extensions made popular by Laxen and Perry's F83 such as `NIP TUCK -ROT` were also added.

Since the target program has no headers (Forth symbol table), there are no target definitions of ' or FIND (note that these can be used in the host at compile time). The mass storage words BLOCK UPDATE SAVE-BUFFERS etc. are also missing, but could be easily added. The words COMPILE and [COMPILE] make no sense in target words, so are absent. LITERAL, [, and] are also missing because they are typically no longer needed.

To aid in the development of application programs, libraries are provided for software floating point, MS/DOS (or CP/M) file system interface, IBM PC direct video output, and redirection of KEY and EMIT.

User Defined – DOES>

It was felt to be necessary for CFORTH to allow user created defining words. The resulting capability is the same as interpreted Forth with the exception that recursive and nested defining words are not allowed, i.e., you cannot define a defining word which defines a defining word...

The host compile state is what makes this possible, along with a very diabolical implementation of DOES>. Notice that defining words which do not use DOES>, and thus rely on the normal run-time execution of CREATE or other defining words, are possible. For instance:

```
H: FOOVAR CREATE 2 ALLOT ;
```

generates a defining word that behaves identically to VARIABLE, and

```
H: BLANKARRAY HERE OVER CARRAY SWAP BL FILL ;
```

generates a defining word which creates a character array with all elements initialized to blanks.

In the general case,

```
H: newdefword olddefword xxx DOES> yyy ;
```

the execution of newdefword zzz will do the following:

1. olddefword zzz will be executed, putting zzz in the dictionary as a data structure, constant, array or table. Data structure allocation in xxx occurs as necessary.
2. DOES> changes the type of zzz to DOES> word, and associates the target-compiled code yyy with it.

When zzz appears in a colon definition, the address of the data structure (or the constant value) normally associated with zzz is pushed on the stack, and a CALL is compiled to yyy.

Except in the case where olddefword is a constant, this definition behaves like most 83 Standard Forth implementations.

Code Optimization

CFORTH attempts to generate the fastest, most compact code sequence by using several techniques:

1. Taking advantage of top of stack registers.
2. Taking advantage of literals.
3. Looking ahead at the next (or later) word.
4. Utilizing the processor state.
5. Having an assembler that knows all the optimized addressing modes.

We will discuss each of these in some detail.

Top of Stack Registers

The majority of Forth implementations do not keep top of stack values in registers, while a very few keep a fixed number (usually one) in registers. CFORTH allows the flexibility of having zero, one, or two values in registers, and either of the two registers can contain top of stack. This drastically reduces the number of stack pushes and pops.

For instance, if both registers are in use (which the compiler considers to be optimum), a `SWAP` emits no machine code, and a `ROT` is three instructions (one on a Z-80). The sequence `VAR1 @ VAR2 @ + VAR3 !` performs no stack manipulation at all. In a typical Forth implementation, it would have to do five push/pop pairs, and even three push/pop pairs in a system with one fixed top of stack register.

In the case of the Z-80, incrementing and decrementing the return stack pointer is so expensive that it is only done when necessary, such as when calling another word. Most return stack accessing is performed using the displacement field in the LD instruction.

Literals

Literals, either numbers, constants, or data addresses (the result of executing variables, for instance) are placed on a compile-time "literal stack", rather than actually having their values pushed on the stack. This provides three advantages:

1. Later arithmetic words that find all their arguments on the literal stack can perform the operation at compile time, pushing the result back on the literal stack. This optimization also occurs for arrays and tables which, of course, perform indexing arithmetic and, for a table, a compile-time memory reference.
2. Arithmetic words which find one argument on the literal stack can sometimes do "strength reduction". For instance, the sequence `1 +` can become an `INC` instruction, and multiplies and divides by powers of two can become shift instructions. On the Z-80, boolean operations (`AND OR XOR`) are optimized by only doing the operation on 8 bits of the stack argument if the literal argument allows. For instance, `16 OR` will not affect the upper 8 bits of the stack argument).
3. Many words can be implemented using immediate operands rather than registers or direct addressing rather than indirect addressing. These former modes tend to save a register (reducing push/pops), and to be faster and more compact. The savings is particularly important on less powerful processors, such as the Z-80, which in order to compile `VAR @` emits a single instruction with the literal stack, but would emit four or five instructions without the literal stack.

Look Ahead

Look ahead is a technique where the next word in the input stream is examined to decide what code sequence to generate. This technique is important in several areas:

1. It is quite often possible to combine a fetch operation (such as `@` or `I`) with a following arithmetic operation (such as `+`) or test (such as `0=`). Thus the sequence `I +` can become the single machine instruction `CX AX ADD` and the sequence `VAR1 @ VAR2 @ VAR3 @ - +` can become (making use of other optimizations as well):
   ```
   VAR1 [] AX MOV VAR2 [] BX MOV VAR3 [] BX SUB BX AX ADD
   ```
2. It is also possible to combine a comparison operation with a following conditional, thus eliminating the need to generate the boolean comparison result. For instance `0< IF` becomes
   ```
   AX AX OR <0 IF,
   ```
   (two machine instructions) instead of
   ```
   AX AX OR 0 # AX MOV <0 IF, AX DEC THEN, AX AX OR =0 ~ IF,
   ```
   (six machine instructions).

The words `0=` and `NOT`, when preceded by a function leaving a boolean result, can appear before `IF WHILE` or `UNTIL` and generate no code—they will just complement the branch condition. Thus `VAR @ IF`, `VAR @ 0= IF`, and `VAR @ 0= 0= IF` all generate the same quantity of code, two instructions.

3. Array words followed by fetch or store can combine the operation into a single indexed memory reference. For instance `VAR1 @ ARRAY1 @` can become:
   ```
   VAR1 [] BX MOV 1 BX SHL ' ARRAY1 >BODY +[BX] AX MOV
   ```

Processor State

The compiler keeps track of whether the condition codes are valid for the value on the top of stack. Thus the sequence `A @ B @ + DUP Ø< IF` never has to test the sign of the sum. It also turns out that when compiling `DUP` the compiler realizes that the duplication is not necessary as well.

Good Assembler

A good assembler can be useful in keeping the amount of code generated to a minimum. The assembler built into CFORTH knows all about generating short data fields and the special accumulator modes, and uses them when possible. A compiler option allows the use of push immediate, multiply immediate and added shift instructions available with 80186, 80188, 80286, 80386, V20, and V30 processors. For the Z-80 version, JR jumps are used instead of JP jumps if possible.

Separate Data Segments

Quite often it is desirable to have separate code and data segments, either because the code is in ROM, or (in the 8086 family) to allow increased program size. Common Forth practice for the former case is to have a dichotomy of management words: `HERE` and `THERE`, `ALLOT` and `RALLOT`. Most interpreted Forth environments do not allow separate 8086 code and data segments because Forth traditionally does not differentiate between code and data. CFORTH does.

CFORTH takes the traditional assembler approach of using directives to specify which segment is desired. `CSEG` causes all further data compilation (such as via `ALLOT` or `CREATE`) to be in the code segment, while `DSEG` causes all further data compilation to be in the data segment. Variables and arrays are always in the data segment, while machine code and tables are always in the code segment. This paradigm is used both for the Z-80 version, where the code and data segments represent ROM and RAM, and the 8086/8 version, where the segments may represent ROM and RAM, and/or different 64k portions of the address space.

In addition, in the 8086/8, a separate stack segment can be specified so that the return and parameter stacks are outside of the 64k data segment.

Within target colon definitions, memory access words (such as `@` and `!`) refer to the data segment. If the memory access word is preceded by `CS:` or `SS:`, then the code or stack segment is used. Standard long address words (such as `@L` and `!L`) are supplied to access any arbitrary segment.

Performance

The following performance was measured on an IBM PC AT with a hard disk and 6Mhz clock speed, using the popular BYTE Magazine "Sieve" benchmark [GIL83]. The Forth source code and disassembled listing are in the appendix. Execution times are for ten iterations (1899 primes). The CFORTH compiler used is version 1.07. Other compilers shown are Borland Turbo Pascal Version 3.01A, IBM Pascal Version 2.0, Computer Innovations C86 C Compiler Version 2.1, Laboratory Microsystems PC/FORTH Version 3.1, and Laxen and Perry F83 Version 2.1.0. Execution times include program loading, except for the Forth interpreters.

Compiler	**Compile Time**	**Algorithm Size**	**.COM Size**	**Execution Time**
	(seconds)	*(bytes)*	*(bytes)*	*(seconds)*
CFORTH	3.7	123	458	3.6
Turbo	~1.0	304	11,692	6.5
IBM Pascal	22	232	26,982	5.3
C	24	194	12,619	5.6
LMI Forth	0.31	166	–	18.9
F83	0.38	163	–	25.6

Conclusion

Much like the traditional interpreted languages, BASIC and Lisp, Forth can be compiled to increase the performance of application programs. Like BASIC and Lisp, some flexibility is lost when going to a compiler, but for the majority of applications the increase in speed is worth the cost.

CFORTH represents an exciting addition to the Forth programmer's tool box. The optimizing compiler keeps code size comparable with that of the interactive Forth environment, while offering performance comparable with the best machine code compilers.

References

[DOW81] Dowling, Tom "Automatic Code Generator for Forth", *1981 FORML Proceedings*. San Jose, CA: Forth Interest Group, 1981.

[FOR83] *Forth-83 Standard*, Forth Standards Team, 1983.

[GIL83] Gilbreath, James, and Gilbreath, J. "Eratosthenes Revisited, One More through the Sieve", *BYTE Magazine*, January 1983, page 283.

[GOL85] Golden, John, Moore, C. and Brodie, L. "Fast processor chip takes its instructions directly from Forth", *Electronic Design*, March 21, 1985.

[MOO85] Moore, Charles, and Murphy, R. "Under the Hood of a Superchip: The NOVIX FORTH Engine", in *1985 Rochester Forth Conference Proceedings*, J. Forth App. & Res. 3(2):185, 1985.

[ROS86] Rose, Anthony "Design of a Fast 68000-based Subroutine Threaded Forth", *1986 Rochester Forth Conference Proceedings*, June 11-14, 1986, Rochester, NY: Institute for Applied Forth Research, 1986, page 183.

Tom is currently employed as a Principal Research Engineer in the Electronic Systems Laboratory, Tek Labs, Tektronix, Inc., where he has been for the past 13 years. Most recently he has been developing a VLSI layout system, written in Forth, for IBM PC's. Tom's education includes an MSEE degree from Stanford University and a BS in Electrical Engineering from Cornell University. He is married and has three children.

His interest in Forth extends beyond Tektronix to an active role in the Greater Oregon Forth Interest Group, authorship of several Forth compilers being distributed by Laboratory Microsystems, and an industrial Z-80 Forth implementation.

Manuscript received April 1986.

Appendix

BYTE Magazine Benchmark:

```
256 MSDOS
8190  CONSTANT SIZE
10000 CONSTANT FLAGS \ array starts at 10000
                     \ allocating space with ALLOT would increase size
                     \ of .COM file

0 0 IN/OUT
: DO-PRIME
     FLAGS SIZE 1 FILL
     0 SIZE 0 DO
          FLAGS  I + C@ IF
               I 2* 3 +  DUP  I + FLAGS +
               BEGIN
                    DUP SIZE FLAGS + U<
               WHILE
                    DUP 0 C<- OVER +
               REPEAT
               DROP DROP 1+
          THEN
      LOOP
      U. ." PRIMES"  CR   ;

: MAIN
     ." 10 ITERATIONS" CR
     10 0 DO DO-PRIME LOOP ;

INCLUDE FORTHLIB
END
```

Disassembled CFORTH Generated Code:

```
DOPRIME                               ; : DO-PRIME
        PUSH    FLAGS                 ; FLAGS SIZE 1 FILL
        PUSH    SIZE
        PUSH    01
        CALL    FILL
        XOR     CX,CX                 ; 0 SIZE 0 DO
        XOR     AX,AX
L0133
        MOV     BX,FLAGS              ; FLAGS
        ADD     BX,CX                 ; I +
        CMP     BytePtr[BX],0         ; C@ IF
        JZ      L0169
        MOV     BX,CX                 ; I
        SHL     BX,1                  ; 2*
        INC     BX                    ; 3 +
        INC     BX
        INC     BX
        PUSH    AX                    ; DUP
        MOV     AX,BX
        ADD     BX,CX                 ; I +
        ADD     BX,FLAGS              ; FLAGS +
L14D                               ; BEGIN
       PUSH    AX                  ; DUP
       MOV     AX,BX
       CMP     BX,SIZE+FLAGS       ; SIZE FLAGS + U< WHILE
       JNB     L0165
       PUSH    AX                  ; DUP
       MOV     DI,AX               ; 0 C<-
       MOV     BytePtr[DI],0
       POP     AX                  ; OVER
       POP     BX
       PUSH    BX
       ADD     AX,BX               ; +
       XCHG    AX,BX               ; REPEAT
       POP     AX
       JMP     L014D
L0165
       INC     SP                  ; DROP DROP
       INC     SP
       POP     AX                  ; 1+
       INC     AX
L0169                              ; THEN
       INC     CX                  ; LOOP
       CMP     CX,1FFE
       JNZ     L0133
       CALL    UDOT                ; U.
       CALL    DOTQ                ; ." PRIMES"
       DB      6,'PRIMES'
       JMP     CR                  ; CR ;

MAIN                               ; : MAIN
       CALL    DOTQ                ; ." 10 ITERATIONS"
       DB      13,'10 ITERATIONS'
       CALL    CR
       MOV     CX,10               ; 10 0 DO
L0197
       PUSH    CX                  ; DO-PRIME
       CALL    DOPRIME
       POP     CX
       LOOP    L0197               ; LOOP
       RET                         ; ;
```