

Chapter 5

Stack Optimisation

In the previous chapter we saw how representing the FORTH stack as a sequence of untyped elements can be useful when formally defining the operations of a FORTH system, compiler or application.

This concept has lead us on to viewing the stack as a simple sequence of (untyped) elements. In turn, this has lead to the development of a compiler optimisation technique that uses these ideas. In this chapter, we review the traditional methods of optimisation used in native code FORTH implementations. We then go on to describe our new technique and how it can be applied to a micro-processor with a large register file, such as the Motorola 68000 or a RISC processor.

5.1 Introduction

Over the years, FORTH compilers have been implemented in a number of different ways. These include:

Threaded Code: In this type of system the compiler will simply generate a list of addresses for a definition. An “Inner Interpreter” is used to pass through the words definition. The first cell of the definition is the address of some assembler code that is able to interpret the rest of the definition. Every FORTH word in the dictionary is defined in this manner including assembler level words.

This is probably the most commonly used method of implementation. It is the implementation method promoted by the early standards (Forth Interest Group 1980; Forth Interest Group 1983). It leads to a system that tend to be small in size (as most of the definitions are lists of addresses) and have a very fast compile time. The method does lead to relatively slow execution timings. It is fairly simple to develop interactive debuggers and structured de-compilers (Buege 1984; Sjolander 1987; Bradley 1985) for such systems.

This method is known as “*Indirect threaded code*”. A slight variation on this system (known as “*Direct threaded code*”) is also widely used. In this variant, each word is assumed to be an assembler definition, thus when a word is invoked, the system makes a machine level call to the word. Such a system will compile a call to the inner interpreter as the first operation of a high-level definition. This method is well suited for systems that are capable of easily supporting two (or more) stacks.

The “*Direct*” system has a level of indirection removed, thus it tends to be slightly faster than “*Indirect*” systems. Applications consisting mainly of low level (assembler) definitions tend to show an execution speed increase due to the inner interpreter being invoked only when required. The size of the final code is dependent on the system and the complexity of the application. On a Z80

⁰This is a chapter taken from the Ph.D. thesis “Practical and Theoretical Aspects of Forth Software Development”. A full copy of the thesis is available from the British library. Both the this chapter and the thesis are Copyright © Peter Knaggs.

based system the code will be slightly larger than that produced with a *Indirect* system, whilst on a RISC based system there will be no size difference. If the application consists mainly of low level definitions the code may be smaller.

In general, it is considered that the slow execution speed disadvantage is overshadowed by the size and de-compiler advantages. The portability of high level definitions is also considered a major advantage of this method.

Subroutines: In some systems, a high level definition consists of a sequence of subroutine calls to the relevant words. Every word is assumed to be a machine code definition. This kind of system is referred to as using the “*Threaded Subroutine*” method as it is close to the *Direct Threaded Code* method using subroutines rather than an inner interpreter.

The main advantage in using this method is that it does away with the need for an inner interpreter. A result of this is a speed increase over *Direct Threaded* systems, although compilation speed may be slightly reduced (Dowling 1981).

The size of the code is dependent on the system and complexity of the application. On a 68000 system although we gain a speed advantage, the code produced is normally larger than *Direct Threaded Code*. On a RISC system there is not only a speed advantage but also a small saving in code size as we don't need to invoke an inner interpreter to interpret high level definitions.

This kind of coding is only useful on systems that can support two or more stacks. On systems that only support one (processor) stack, it is necessary to synthesize one of FORTH's stacks. It is normal to synthesize the “return” stack leaving the built in (faster) stack for the “parameter” passing.

It is a fairly simple step to make some of the low level kernel words macros, such that, rather than compiling a subroutine call, it compiles the instructions needed to perform the function directly. This is referred to as “*inline code*”.

Typically, the code compiled into the definition consists of simple stack manipulations and some flow control operations. The code that is compiled “inline” in this way is normally fixed by the compiler designer. A number of people have looked at automating this process (Pawley 1984; Rose 1986; Almy 1987). In this chapter, we look into these methods and show how treating the stack formally (as a sequence of items) can lead to an advanced optimisation technique.

In such a system, we would expect to find a dramatic speed increase although a slight compilation speed decrease is also possible. We would also expect to see less use of the return stack. The compiler designer will have to make the choice of possible smaller slower code or slightly larger and faster code. Although the final code may be faster, it may also be larger, dependent on the compiler.

Hardware: A number of attempts have been made to produce micro-processors that are capable of executing a FORTH-like machine language. Chips such as the Novix-4016 (Golden, Moore, and Brodie 1985; Jennings 1985; Miller 1987), Harris RTX-2000 (Danile and Malinowski 1987; Jones, Malinowski, and Zepp 1987; Harris Semiconductor 1988c) and the MuP20 (Moore 1990a) have been designed along these lines. The binary instruction set of these processors is a basic implementation of the low level kernel required to execute a FORTH program. A compiler will generate native code to execute on one of these chips. As this code is close to the FORTH system, such compilers produce fast and small code.

A number of more advanced designs are currently under development including designs by Chuck Moore (Computer Cowboys), Marty Fraeman (Johns Hopkins University/Applied Physics Laboratory), Pedro Luis Prospero Sanchez (Cidade University in Sao Paulo) and Sergei Baranoff (St. Petersburg Institute for Informatics and Automation).

The majority of FORTH system use “*threaded code*” techniques, thus the compilers are quick and simple. The alternative “*native code*” techniques tend to use more space but offer more opportunity

for optimisation. The rest of this chapter looks at some of these optimisation techniques and introduces a new one based on a formal view of the parameter stack.

It is worth noting that the nature of the language makes many of the traditional optimisation techniques (such as common subexpression, copy propagation, loop optimisation and code motion) irrelevant. There are, however, a couple of the traditional techniques (such as dead-code elimination and flow graphs) that can be used (Aho, Sethi, and Ullman 1986; Bruno and Lasseigne 1975).

5.2 Code Generation

In a FORTH system that generates native code instead of threaded code, the operation of a word could simply be described by a sequence of subroutine calls to the relevant operations. This has been termed “*subroutine threaded code*”. For example, a definition of the standard word `+` could be:

```
: +!      ( n1 addr -- ; Add n to contents of addr )
  DUP    ( n1 addr addr )
  @      ( n1 addr n2 )
  ROT    ( addr n2 n1 )
  +      ( addr n1+n2 )
  SWAP   ( n2+n1 addr )
  !      ( )
;
```

This would produce a operation body of:

```
JSR  DUP      Dup
JSR  _Fetch   @
JSR  ROT      Rot
JSR  _Add     +
JSR  SWAP    Swap
JSR  _Store   !
RTS                          ; (Return to caller)
```

5.3 Inline Compilation

The code generated by this definition is just as simple to produce as in the threaded code technique. However, we now incur an overhead in using the subroutine (`JSR`) instruction. This places an inherent slowness into the system, a subroutine call has to place its return address on the stack, execute the target code and return to its calling location. On the 68000, the threaded code method is more complex and slightly slower (Almy 1987) whilst a RISC is additionally required to clear out its instruction pipeline. Anthony Rose (1986) has proposed a system of “inline compilation” that reduces this overhead.

The basic principal behind Rose’s idea is that each word is to be compiled so that it can be used as a subroutine call (as with subroutine threaded systems). The length of the code (excluding the subroutine return (`RTS`) instruction) will be stored in the word’s header. When a reference to the word is to be compiled, the compiler will compare the length of the word’s body to a compilation variable (`CSIZE`). If the word is smaller in length to the given size then the code will be copied directly into the new definition. If, however, the code is over the limit set by `CSIZE`, a subroutine call to the code will be compiled into the new definition. By having a reasonable value for the “inline” limit (Rose suggest 13 for a 68000 based system), the overhead of making a subroutine call can be removed as the code is compiled directly into the definition. Note, this does not eliminate the subroutine calls to pre-compiled code, it simply reduces the frequency of them.

Rose also noted a requirement for two additional compilation flags, *subroutine* (**SUBR**) and *inline* (**INLINE**) to be used in the same manner as **IMMEDIATE**. If the **INLINE** flag is set, the word is compiled directly into the code ignoring the value of **CSIZE**. Conversely, if the **SUBR** flag is set, a subroutine call to the code will *always* be compiled.

5.4 Peep-Hole Optimisation

Using inline compilation, the word **+** would now have a operation body of (in standard 68000 assembler notation):

```

MOVE.L  (A6),-(A6)      Dup
MOVEA.L (A6),AO        @
MOVE.L  (AO),(A6)
JSR     ROT            Rot
MOVE.L  (A6)+,DO       +
ADD.L   DO,(A6)
MOVE.L  (A6)+,DO       Swap
MOVE.L  (A6),D1
MOVE.L  DO,(A6)
MOVE.L  D1,-(A6)
MOVEA.L (A6)+,AO      !
MOVE.L  (A6)+,(AO)
RTS
;
```

Notice the additional time incurred by the passing of arguments on the stack which is then used by the next word in the definition. The **DUP** is used so that the **@** will not destroy the address on the stack. This is an additional overhead caused by using this inline compilation. However, if the word **@** were to be **IMMEDIATE**, it can scan back through the code just compiled. This means that it can recognise that a **DUP** was compiled just prior to it. Having done this it can now overwrite the code for **DUP** with some code that would perform the fetch without removing the address from the stack (or compile an implied **DUP@**). Given that all the basic¹ words perform this kind of scanning back to the previous word compiled, the body for **+** would now be:

```

*                               Dup
MOVEA.L (A6),AO              @
MOVE.L  (AO),-(A6)
JSR     ROT                  Rot
MOVE.L  (A6)+,DO            +
ADD.L   DO,(A6)
*                               Swap
MOVEA.L 4(A6),AO             !
MOVE.L  (A6)+,(AO)
ADDQ    #8,A6
RTS
;
```

¹There is an ongoing argument in the FORTH community as to what precisely constitutes this basic set of words.

The `!` word has replaced the code generated by `SWAP`. Notice how `!` has used the instruction `ADDQ #8,A6` to move the parameter stack back to its expected location.

Here a saving is made by each word scanning back to the previous word compiled. It is common for a word to be able to scan back up to four previously compiled words. If the word can scan back further (to the beginning of the definition), a greater saving can be achieved. The body for this could then be:

```
MOVEA.L (A6)+,A0    Dup @
MOVE.L  (A6)+,D0    Rot
ADD.L   D0,(A0)     + Swap !
RTS                                           ;
```

This form of backwards scanning of the compiled code (known as “*peep-hole optimisation*” (Tanenbaum, van Staveren, and Stevenson 1982)) can give rise to some rather more optimal coding that would not otherwise be available. Any function that relies on information from the stack can be optimised in this way.

5.4.1 Conditionals

Another place for excessive optimisation is the conditional words (such as `IF`). Including all of the words that leave a boolean flag on the stack (such as `=`). Currently, the word `IF` simply interrogates the boolean flag on the top of the stack to make a conditional jump. If it were to scan backwards then the boolean flag would not be required as it could be integrated into the jump instruction at the condition test. For example, let us take the code “`0 > IF`”, this would normally be compiled as:

```
MOVE.L  #0,-(A6)    0

MOVE.L  (A6)+,D0    >
MOVE.L  (A6),D1
CLR.L   D2          0 (false)
CMP     D1,D0
BLE     f0
SUBQ   #1,D2       -1 (true)
0: MOVE.L D2,(A6)

MOVE.L  (A6)+,D0    IF
BEQ     <n>
```

Where `<n>` is the conditional offset. It is initially set to `0` while its correct value is calculated by the word ‘`THEN`’.

With the `>` and `IF` words producing optimal code, this could now be compiled as:

```
CLR.L   D0          0
MOVE.L  (A6)+,D1    >
CMP     D1,D0
BLE     <n>
```

Here the `>` word replaced the code to place a literal `0` on the stack as it is not required. The `IF` word replaces the processing (and production) of the boolean flag by placing its offset in the code produced by `>` (the `BLE` instruction), thus removing the need to pass the boolean flag from the one word to the next.

If multiple conditions are being tested (ie a logical `OR` or `AND` is being used), it would be possible to use lazy evaluation (Minker and Minker 1980; Hanson 1980) to reduce the time taken to evaluate the logical expression prior to the `IF`.

5.5 Registers

Some implementors have extended this thinking so that they place the top of the stack in a specific (*static*) register (Bradley and Saari 1988). Others have tried placing the top three items in registers. The second one being in an *Address* register rather than an *Data* register (`comp.lang.forth` 1992).

This form of optimisation can be used with all of the implementation methods. However, it is felt that the use of *static* registers to hold elements of the stack is cumbersome. The speed advantage is outweighed by the complexity of the system. Indeed this is often quoted as being the most frequent source of error in such systems (`comp.lang.forth` 1992).

5.6 Optimisation using a Stack image

In the rest of this chapter we present a new technique which allows the top items of the stack to be stored in internal registers. The registers used to store the items (S_0 , S_1 and S_2) are to be allocated *dynamically*. In order to do this, the compiler will have to maintain a record of which register holds which stack element. This record is in the form of an image. To show this “*stack image*” we use a notation that relates a register to its stack element, thus the sequence:

$$\langle D_0, D_1, D_2 \rangle$$

indicates that the top item of the stack (S_0) is stored in the register D_0 with the second stack element (S_1) in D_1 and the third (S_2) in D_2 . Any stack item not given in the sequence is assumed to be on the physical stack of the micro-processor. Hence, the notation $\langle D_2, D_0 \rangle$ signifies that S_0 is in register D_2 , S_1 is in D_0 and that S_2 is not being held in an internal register but is on the micro-processor’s ‘physical’ stack.

Underflow of the parameter stack may still occur, however it is possible to include some form of checking into the compiler to check for this (Hoffmann 1991). By having a formal list of arguments into and out-of a word, the compiler will be able to find out if underflow will result from the expected usage.

To show how our system works, let us look at the standard word **SWAP**. If the top two elements of the stack are stored in registers, **SWAP** need only change the compilation stack image to achieve its run-time effect. However, if one or both of the elements are on the parameter stack then they will have to be brought into internal registers. We now have to update the stack image. In doing so it can also swap the items around.

Table 5.1 shows the possible compilation actions that can be taken by the word **SWAP** dependent on the current state of the stack image².

Code Generated:	Stack Image
none	Before: $\langle D_0, D_1 \rangle$ After: $\langle D_1, D_0 \rangle$
MOVE.L (A6)+,D1	Before: $\langle D_0 \rangle$ After: $\langle D_1, D_0 \rangle$
MOVE.L (A6)+,D1 MOVE.L (A6)+,D0	Before: $\langle \rangle$ After: $\langle D_0, D_1 \rangle$

Table 5.1: Example **SWAP** actions

²By this we mean the current values in the stack image.

5.6.1 Argument Passing

Sometimes it will be necessary to invoke these (compiling) words as a subroutine (ie when invoked from the keyboard interpreter). In such cases we do not know the state of the stack. To counter this, we say that the stack image must always be

< D0, D1, D2 >

on entry to and exit from the subroutine. Thus it is now the responsibility of the subroutine to make sure that the stack image is in the same form on exit of the routine.

Let us take the **SWAP** word. This now has two different actions to take depending on whether it was invoked as a subroutine or from the compiler. If the word is invoked from the compiler, it must compile the relevant code into the dictionary (as shown in table 5.1). Otherwise it will have to perform as a subroutine, the following code defines the action of the word when invoked as a subroutine:

```
_SWAP:  EXG      D1,D0
        RTS
```

This subroutine knows what the stack image is supposed to be on entry and exit of the code. By simply exchanging the contents of the top two registers, it has performed its function without having to alter the stack image.

The definition of the compiler word **SWAP** would have to be intelligent enough to discover whether it was invoked from the compiler or interpreter taking the required action. The following is a possible definition of the **SWAP** word³:

```
HEX
: SWAP ( n1 n2 -- n2 n1 ; Swap over the top two stack elements )
  ?COMP IF
    ( In compilation mode - Take relevant action )
    SLEN @ CASE

    0 OF ( All argument on physical stack )
      IN-LINE A6 )+ D0 .L MOVE
          A6 )+ D1 .L MOVE  END-CODE
      , , ( Place Op-Code into Definition )
      2 SLEN ! ( Update Stack Len )
      0 >S0 1 >S1 ( Set up Stack Image )
    ENDOF

    1 OF ( Second argument on physical stack )
      SO> 1+ 3 MOD DUP ( Find next register )
      IN-LINE A6 )+ D0 .L MOVE  END-CODE
      200 * OR ( Replace D0 with correct reg.)
      , ( Compile into the definition )
      2 SLEN ! ( Update Stack Length )
      SO> >S1 >S0 ( Update stack image )
    ENDOF

    ( Both in internal registers )
      SO> S1> ( Read old stack image )
      >SO >S1 ( Set new stack image )
    ENDCASE
  ELSE

    ( Called form Keyboard Interpreter )
    _SWAP
```

³The method of implementing the interrogation of the current interpreting/compiling state was chosen to show the concept, other methods may be better for full implementations.

```

THEN
; IMMEDIATE

```

Here we can see the definition of the kernel words, such as **SWAP**, are far more complex than the simple **CODE** definitions that we are used to in the more traditional compilers. However, only the basic (kernel) operations will need such a complex definition.

In this definition, we have used many non-standard words. These have been used to make the definitions more readable. The function **?COMP** is used to detect whether the **FORTH** system is in interpret or compilation mode. The version of the word used here is different from its conventional usage in that it returns a flag rather than producing an error message. The words **S0>**, **S1>** and **S2>** read the value of the register holding the stack elements S_0 , S_1 and S_2 respectively. In contrast, the words **>S0**, **>S1** and **>S2** place a register number in the stack image for the relevant element. The variable **SLEN** is used to hold the current size of the stack image. The action taken by the word at compile time is dependent of the size of the stack image.

Finally, we have introduced the new concept in the word **IN-LINE**. It is used to enter the assembler such that the machine instructions are compiled into the definition as literal values and not as executable code. This is the same as working out the op-codes by hand and coding them into the definition as literal values. However, **IN-LINE** uses the built in assembler to compute the op-codes for us, it also makes the source code a lot more readable.

Now let us apply this idea to our definition of **+!**. The body of the word (after compilation with such words) could now be:

```

*
MOVE.L  D2,-(A6)  Dup      < D0,D1,D2 >
MOVE.L  D0,D2    < D2,D0,D1 >
MOVEA.L D2,A0    @
MOVE.L  (A0),D2  < D2,D0,D1 >
*
ADD.L   D1,D2    Rot      < D1,D2,D0 >
+
*
MOVEA.L D0,A0    !
MOVE.L  D2,(A0)  <>
MOVE.L  (A6)+,D0 ;
MOVE.L  (A6)+,D1
MOVE.L  (A6)+,D2
RTS      < D0,D1,D2 >

```

Notice how the **;** word has compiled three **MOVE.L (A6)+,Dn** instructions. This is in order to regenerate the correct stack image for exit from the subroutine. This is the “worst case” situation as all three registers have to be *popped* off the physical stack into registers.

5.6.2 Conditional execution

When a branch is made in the code (an **IF** instruction), the state of the stack image could differ between compile and run time. If the compiler did not take account of this it would compile code to work with one stack image when in fact another stack image is being used. To overcome this, we save a copy of the stack image at the start of the conditional (after the **IF**). At the end of the conditional (the word **THEN**), we will have to force the stack image to be the same as when the condition started (the saved image). In this way, we can say that the stack image will be the same if we skip over the conditional or if we execute it. Notice that the extra code to realign the stack is included with the condition. An example of this is shown in figure 5.1.

```

IF    ← Save stack image
  true part
THEN ← Realign stack image

```

Figure 5.1: Handling a conditional

For an **IF**...**ELSE**...**THEN** construct, we can extend this idea such that the **ELSE** word will swap the current stack image with the saved one. This means that both parts of the conditional code start with the same stack image and that the *false* part is required to realign its stack to be the same as that at the end of the *true* part. Figure 5.2 shows an example of this method.

```

IF    ← Save stack image
  true part
ELSE ← Swap current/saved stack images
  false part
THEN ← Realign stack image

```

Figure 5.2: Handling multiple conditions

5.6.3 Looping structures

It is possible to handle loop structures in the same way as conditional structures. We save the stack image at the start of the iteration and realign it at the end. Figure 5.3 shows how we would implement this idea on a **BEGIN**...**WHILE**...**REPEAT** structure.

```

BEGIN ← Save stack image
  condition test
WHILE ← Save stack image
  loop code
REPEAT ← Realign to BEGIN image
          ← Recover WHILE stack image

```

Figure 5.3: Handling loops

This figure shows a peculiar problem with the looping system. When we compile the **BEGIN** word we do not know what kind of loop structure we are compiling. Thus, we save the current stack image in anticipation of it being used. All of the control structures that start with the word **BEGIN** will realign to this value at some point. The **WHILE** loop is a special case as we are required to remember the stack image twice, at the '**BEGIN**' and at the '**WHILE**'. The '**REPEAT**' command will realign the stack to the **BEGIN** stack image. On completion of the loop, the **WHILE** stack image will be in force, thus **REPEAT** will also have to reset the compilation stack image back to the **WHILE** version.

5.7 References

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley.

- Almy, T. (1987). Compiling of FORTH for performance. *Journal of FORTH Application and Research* 4(3), 379–388.
- Bradley, M. (1985). Self-understanding programs. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.
- Bradley, M. and M. Saari (1988). *Sun FORTH User's Guide*. Mountain View, CA: Bradley FORTHware.
- Bruno, J. and T. Lassagne (1975). The generation of optimal code for stack machines. *Journal of the ACM* 22(3), 382–396.
- Buege, B. (1984). A decompiler design. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.
- `comp.lang.forth` (1987–1992). Usenet newsgroup.
- Danile, P. and C. Malinowski (1987, June). FORTH processor core for integrated 16-bit systems. *VLSI Systems Design* 8(7), 98–104.
- Dowling, T. (1981). Automatic code generator for FORTH. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.
- Forth Interest Group (1980). *FORTH-79 Standard*. San Carlos, CA: Forth Interest Group.
- Forth Interest Group (1983). *FORTH-83 Standard*. San Carlos, CA: Forth Interest Group.
- Golden, J., C. Moore, and L. Brodie (1985, March). Fast processor chip takes its instructions directly from FORTH. *Electronic Design*, 127–138.
- Hanson, D. R. (1980). Code improvement via lazy evaluation. *Information Processing Letters* 11(4–5), 163–167.
- Harris Semiconductor (1988). *RTX 2000 Real Time Express Microcontroller Data Sheet*. Melbourne, FL: Harris Corporation.
- Hoffmann, U. (1991). Stack checking — a debugging aid. In *Proc. EuroFORML Conf.*, San Carlos, CA. FORTH Interest Group.
- Jennings, E. (1985, October). The novix NC4000 project. *Computer Language* 2(10), 37–46.
- Jones, T., C. Malinowski, and S. Zepp (1987, May). Standard-cell CPU toolkit crafts potent processors. *Electronic Design* 35(12), 93–101.
- Miller, D. (1987, April). Stack machines and compiler design. *Byte* 12(4), 177–185.
- Minker, J. and R. G. Minker (1980). Oprimization of boolean expressions — historical developments. *A. of the History of Computing* 2(3), 227–238.
- Moore, C. (1990). *MuP20 Microprocessor: Preliminary Specifications*. Woodside, CA: Computer Cowboys.
- Pawley, W. (1984). Using native machine code analogs of interpreted FORTH's elements for high performance. In *Proc. Rochester FORTH Conf. on Real Time Systems*, Rochester, NY, pp. 115–120. Institute of Applied FORTH Research.
- Rose, A. (1986). Design of a fast 68000-based subroutine threaded FORTH with inline code & optimiser. In *Proc. Rochester FORTH Conf. on Artificial Intelligence*, Rochester, NY, pp. 285–288. Institute of Applied FORTH Research.

Sjolander, S. (1987). Novix decoder. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

Tanenbaum, A. S., H. van Staveren, and J. W. Stevenson (1982). Using peephole optimization on intermediate code. *TOPLAS* 4(1), 21–36.