Performance Analysis in Threaded Code Systems

Michael Perry

Abstract

The importance of performance analysis to the iterative design process is discussed. Several techniques for performance analysis in Forth systems are described. Some related debugging techniques are mentioned.

Iterative Design

Forth encourages the use of iterative design methodology, which involves the rapid repetition of a short design/implement/test cycle. Most Forth systems provide reasonably good tools for finding bugs in code and flaws in algorithms. Few systems provide adequate tools for diagnosing performance problems. A good rule of thumb is that a program spends ninety percent of its time executing ten percent of the code. When some performance goal must be met it is neccessary to find those routines where most time is spent and make them run faster. Forth encourages modular programming, and it is easy to replace a slow routine once it has been found.

Memory Allocation

One requirement which many of these tools share is the allocation of memory for storing the results, usually a counter for each word in the dictionary. A very useful technique is available on systems which have a view field in the header of the dictionary entry for each word. The view field points to the source code block on disk. Once having used a system with the ability to automatically locate the source for any word, it is difficult to go back to a system which lacks it.

The view field can be borrowed for other uses, such as the counter needed for these tools. When the view field is not available, an array with one entry per word in the dictionary is required. For compatability, the word >VIEW should convert the address of a code field into the address of the corresponding cell This kind of code is very dependent on the particular system and CPU being used. Examples later in the paper are for the MC68000 running F83, a public-domain, 83-standard Forth system.

Monte Carlo Analysis

One simple tool which gives a statistical indication of the frequency of execution of a word uses a real time clock interrupt routine which periodically samples the interpreter pointer (IP) or program counter (PC). This can be used in a couple of different ways, either by incrementing a counter for the word being executed, or by building a histogram of IP or PC activity.

There are several ways to decide which counter to increment. Assuming a post-incrementing, indirect-threaded system, IP points to a pointer to the code field of the next word to execute. It would be convenient, but wrong, to increment that word's counter. It may or may not be a slow routine; after all, it is not even being executed at the moment. To increment the counter for the word into which the IP points is better (and slower), but still incorrect, because that word merely called the word being executed.

It is best to increment the counter for the word currently executing, which is pointed to by the cell before the one pointed to by the IP.

```
LABEL BUMP   ( increment counter of word being executed. )
   -2 IP D) A0 MOVE   A0 ) A0 MOVE   TO-VIEW #) JSR   1 A1 ) ADDQ   RTE
( TO-VIEW converts a code field address in A0 to a view field address in A1 )
```

To build a histogram, an array is allocated whose size depends on the desired resolution. Two variables contain the upper and lower limits of the range of addresses of interest. If the IP (or PC, as the case may be) is inside the window of interest, the cell corresponding to its relative position is incremented. The results are displayed as a histogram, and can give an idea of where most time is spent. After each pass the window can be made narrower to get more precise results, but more time will be required to get the same number of counts inside the window.

Static Frequency Analysis

Another method is static frequency analysis. A counter is associated with each word in the dictionary, and they are initially set to zero. Each time the word is encountered by the compiler its count is incremented. After compiling, a table of counts is printed. While this does not show the relative execution times or frequency of execution, it at least indicates which words may deserve some attention.

Implementation requires writing a new compiler loop which finds each word and increments its count. Immediate words such as IF present a problem. The best solution is probably to redefine them to increment the counter for any words which they compile. Comments and words which terminate compilation must be executed as usual while compiling.

```
: COMPILER   ( -- )   setup
   BEGIN  BL WORD FIND DUP 0>
     IF  DROP EXECUTE
     ELSE  IF  1 OVER >VIEW +!  ,  ELSE  DROP  THEN
     THEN  finished?
   UNTIL  ;

: IF   ( -- adr flag )
   COMPILE ?BRANCH  >MARK   1 ['] ?BRANCH >VIEW +!  ; IMMEDIATE
```

Dynamic Frequency Analysis

In some systems the address interpreter (NEXT) is a single routine shared by all code words. In other systems it is distributed, with each word having its own copy. A distributed NEXT is usually faster, but prevents the use of some interesting techniques which involve replacing NEXT with a routine which does additional work.

In dynamic frequency analysis, NEXT is replaced by a routine which increments the counter for the word which is about to be executed, in addition to performing the usual NEXT function.

```
LABEL BUMP-NEXT
   IP )+ A0 MOVE   TO-VIEW #) JSR   1 A1 ) ADDQ   A0 ) A0 MOVE   A0 ) JMP
```
(Note the similarity to the BUMP routine given earlier.)

As before, all counters are zeroed before running the code to be analysed, and a table of counts is printed afterwards. This technique still does not show which routines take the longest to execute, but it does show which are executed most frequently, and that is often a good indication of where time is being spent.

Another limitation is that real time code will almost certainly fail to be fast enough while running the test, and this may change the behavior of the system enough to render the results meaningless. In such time critical cases, only hardware tools will accurately reflect the operation of the system.

Assembler Cycle Count Generation

An assembler in a Forth system is relatively simple. Ordinarily it provides a minimum of error checking, and can be defined in from three to fifteen screens or so. It is possible to add features such as error checking to the assembler. For performance analysis, information about instruction and addressing mode timing can be added to the assembler. Whenever an instruction is assembled, a cycle counter is incremented by the duration of the instruction and its addressing mode or modes. In many cases the actual execution time will differ because of pipelining, variable execution times, or inaccurate data in the manual. However, it is usually possible to get a good idea of the worst case time, and that is normally the best guide to performance.

The execution time for a high level definition can be calculated as the sum of the times of the components, plus the execution time of NEXT times the number of components, plus the entry and exit times. For this calculation to be possible, the timing information for every word must be available. This can be built in to the dictionary entry for each word by the meta-compiler; it is more difficult if the Forth system's source is in assembly language. When target compiling small applications the counts can be kept just in the host system.

One significant limitation of this technique is that most words contain some control structures. In the case of IF ELSE THEN structures, the times of the two branches can be calculated and the larger value used. For DO LOOP and BEGIN WHILE REPEAT or BEGIN UNTIL structures the situation is much more difficult. The structure could be assigned a duration equal to one pass through its body, but this is not likely to be correct.

Fortunately those words on which performance usually depends are those buried deep inside loops of one kind or another, and so it is not so neccessary to analyse higher level words accurately.

Debugging

As mentioned before, several debugging tools can be built on special versions of NEXT. The simplest is a breakpoint. In NEXT the IP is compared to the contents of a variable which contains the address at which a breakpoint has been set. If they are the same, execution is aborted or a breakpoint handler is executed. An array of addresses can be used instead of a single variable if multiple breakpoints are desired.

A debugger can use a NEXT which tests the IP against a pair of addresses, and executes a trace routine if it is between them. Another method is to execute trace whenever the return stack depth has some value, but that approach is a bit trickier to use.

A related tool protects the contents of an address. NEXT compares the present contents of the protected address to a saved value, and enters the debugger if they differ.

```
CREATE SAVED 4 ALLOT   ( address, value )
LABEL PROTECTING-NEXT
   SAVED #) A0 LEA   A0 )+ A1 LMOVE   A0 ) D0 MOVE
   A1 ) D0 CMP  0<> IF   TRACING #) JMP   THEN   ( fall through )
LABEL NORMAL-NEXT
   IP )+ A0 MOVE   A0 ) A0 MOVE   A0 ) JMP
( TRACING sets the debugger window to point to the word which caused the )
( change, and NORMAL-NEXT is a copy of the normal NEXT code. )
LABEL PROTECTING
   PROTECTING-NEXT #) JMP
CODE PATCH-NEXT   ( -- )
   PROTECTING #) >NEXT #) LONG MOVE   NEXT  END-CODE
( installs a jump to the new NEXT on top of the old NEXT )
: PROTECT   ( address -- )
   DUP @ SWAP SAVED 2!   PATCH-NEXT  ;
```

Conclusions

Many powerful tools for debugging and analysis are possible in a threaded-code system. The flexible and open nature of these systems allow interesting dynamic modifications to their structure and behavior. Continued exploration of the possiblilities will doubtless prove rewarding.

Bibliography

Laxen, H.: Debugging Techniques
   Forth Dimensions, v.6 #2 and #3
McClees, H.: A Functional Usage Analyser
   Rochester Forth Conference Proceedings, 1983
Russell, J. & Solntseff, N.: Break Point Utility
   Rochester Forth Conference Proceedings, 1984
Spreier, P.: Performance Monitoring in Forth
   FORML Conference Proceedings, 1981
Whitney, A. & Conrad, M.: Call Forth for Realtime Control Programming
   Computer Design, v.22 #5