

---

# Threaded Code

James R. Bell  
Digital Equipment Corporation

**The concept of "threaded code" is presented as an alternative to machine language code. Hardware and software realizations of it are given. In software it is realized as interpretive code not needing an interpreter. Extensions and optimizations are mentioned.**

**Key Words and Phrases:** interpreter, machine code, time tradeoff, space tradeoff, compiled code, subroutine calls, threaded code

**CR Categories:** 4.12, 4.13, 6.33

## 1. Introduction

One of the most fundamental tradeoffs in software engineering is that of space versus time. It is normally possible to write a faster and larger (or a smaller but slower) version of a given program. In this paper we describe a technique called threaded code of implementing programs. Under suitable circumstances, it is shown to achieve a desirable balance between speed and small size.

The most common alternative techniques of programming might be denoted "hard code" and "interpretive code." Hard code (or machine code) is the most used method of programming. Each instruction of the program is chosen from the set wired into the host computer by its designers. Each such instruction executes rapidly since it is wired into the physical circuitry of the computer. On the other hand, the given instruction set is suboptimal for almost any specific problem because the user is forced to accept unneeded generality in some places and to circumvent a lack of

---

Copyright © 1973, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's address: Digital Equipment Corporation, 146 Main Street, Maynard, MA 01754.

desired generality in others. In summary, when writing hard code the user needs relatively many instructions, each of which executes relatively rapidly.

An interpreter, by contrast, is a vehicle by which the user can choose his own instruction set to correspond to his specific problem. Obviously such freedom allows a much shorter program for that problem to be written. The penalty is that the instruction set of the interpreter is not in fact implemented in the computer's hardware. Instead the interpreter must itself be a computer program which simulates the action of the interpretive instruction set in terms of the actual instruction set. This can be a time-consuming proposition. Thus interpretive code tends to be shorter but slower than hard code.

It is instructive to look at the relation between the host hardware and the alternatives discussed. In the case of hard code an instruction directs the flow of processing by its actual execution from the IR, or instruction register, of the machine. In the case of an interpreter, an "instruction" is in fact merely a datum from the interpreting program. Thus it directs the flow of processing from an accumulator or the equivalent. We may now describe a "threaded code computer"—a machine in which an "instruction" controls the PC, or position counter.

## 2. Threaded Code: Hardware and Software Realizations

Let us imagine a computer which works in the following way:

- Step 1. S, the value of the PCth word of memory, is fetched.
- Step 2 (a). The routine starting at location S of memory is executed.
- Step 2 (b). The value of PC is incremented by one.
- Step 3. Go to Step 1.

We shall call this machine a threaded code computer.

It is quite feasible to build a physical device corresponding to the above description. However, this is unnecessary. We shall show that it is possible to economically transform a suitable general purpose computer into a threaded code computer via programming.

Let us describe the implementation on a PDP-11 computer. Let the PC of the threaded code computer correspond to a general register *R* of the PDP-11. Then to use the PDP-11 as a threaded code computer we need only end each of the set of routines described in Step 2(a) with the instruction

JMP @ (R) +

which links the end of one routine to the beginning of the next. This is PDP-11 notation for:

- Step A. Transfer control to the routine beginning at the location whose address is the value of the *R*th word of memory.
- Step B. Increment *R* by one word.

Fig. 1. Flow of control: hard code.

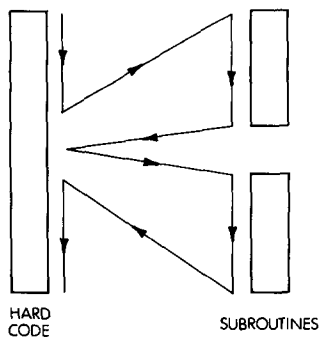


Fig. 2. Flow of control: interpretive code.

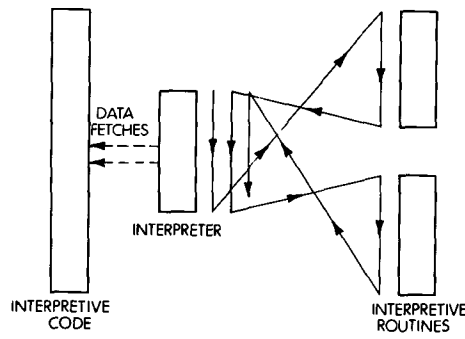
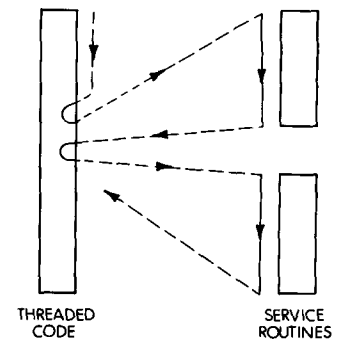


Fig. 3. Flow of control: threaded code.



But note that Step A does exactly what was specified in Steps 1 and 2(a) of our description of the threaded code machine. Similarly, Step B does exactly what was specified in Step 2(b). Thus we have in fact transformed a PDP-11 into a threaded code computer.

If a computer contains instructions which can increment a register and can load the PC through two levels of indirect addressing or the equivalent, then such a procedure is possible. If these things can all be done with a single instruction, then a very economical implementation using that instruction is available.

What we have created is in effect interpretive code which needs no interpreter. Figures 1, 2, and 3 contrast the flow control patterns of threaded, hard and interpretive code.

### 3. The Economics of Threaded Code

Let us now show the practical value of threaded code. As an example let us use the evaluation of arithmetic expressions. Ignoring unary operators we may visualize such an expression as an infix sequence  $X_1 \text{ op}_2 X_3 \text{ op}_4 \dots X_n$ , where each  $X$  is a variable name and each  $\text{op}$  is an arithmetic operation.

This expression can be rearranged by well known methods into a Polish suffix sequence  $P_1 P_2 P_3 \dots P_n$ , where each  $P_i$  represents either variable name or an operator. A wide variety of code might be generated to evaluate this sequence. For example, one straightforward possibility is to generate a sequence  $C$  of code  $C_1 C_2 C_3 \dots C_n$

Here a  $P_i$  which is a variable name generates code  $C_i$  to stack the value of that variable, and a  $P_i$  which is an operator causes code  $C_i$  to be generated which applies that operation to the stack top. Given suitable hardware, the sequence  $C$  should execute rapidly. If the size of  $C$  is sufficiently small, then this "hard code" solution serves well.

However, there are reasons that  $C$  may become unduly bulky: a multiple word instruction or a multiple instruction sequence may be needed to implement some of the  $C_i$ 's. Some data types (e.g. long floating, double integer, or complex numbers) may be awkward to move. Or some operations may have to be executed by a software routine (e.g. floating operations or integer multiply/divide on many small machines; complex arithmetic on almost all machines).

Under such conditions threaded code has the advantage of an interpreter (i.e. shorter code than the hard code) while still being quite competitive with regard to speed. The speed of the threaded code is due primarily to the fact that the single instruction for threading can replace both a call and a return from a subroutine. For example, on the PDP-11 Model 45, threading takes  $1.2 \mu\text{s}$  whereas a subroutine call plus a return takes  $2.6 \mu\text{s}$ . Of course, an inline code sequence needs no linkage and thus is  $1.2 \mu\text{s}$  faster than threaded code. Therefore, threaded code is relatively the most attractive where subroutines are frequent.

Time and space comparisons are heavily influenced by the examples chosen and the assumptions made about the code generated (e.g. whether registers or a stack is used). Using a large sample of actual FORTRAN programs compiled into threaded code on the PDP-11 computer, we have found threaded code roughly equal to hard code in speed (typically 2 or 3 percent slower) while somewhat shorter (typically 10-20 percent). Since this particular compiler generates threaded code even for those operations such as integer arithmetic, where such a course is not advantageous, it is reasonable to assume that threaded code can be even more beneficial in other applications.

Threaded code tends to become more attractive as program size increases, since the cost of each threaded service routine can be amortized across more calls. It is also worth noting that threaded code, unlike typical interpreters, contains only those service routines actually called upon by a given program.

#### 4. Variations on Threaded Code

The previous example assumed a stack was used as the basic discipline for data. Actually this assumption is unnecessary. The threaded code service routines can pass or receive data according to any convention; they may even be passed parameters if desired. The parameters of a routine can immediately follow the threaded link to the routine. As each is used by the service routine, the link pointer can be incremented to step through the parameters. For example, on the PDP-11 a two-parameter routine to copy a word *A* to a word *B* could look like this:

```
CALL: COPY      }
              A  } threaded code
              B  }
              ⋮  }
COPY: MOV @ (R) +, @ (R) + } service routine
      JMP @ (R) + }
```

We have presented the concept of threaded code in its most basic form. There are numerous time and space optimizations which could be made. For example, it can easily be determined whether a given service routine *R* is always followed by the same other service routine *S*. If so, then *R* can end with a jump directly to *S*, leaving one less link to thread. Moreover in many cases the routine for *R* can be placed immediately before the routine for *S*, thereby eliminating the need for any jump at all. This clearly saves both space and time.

In a practical application it may be expedient to write some sections in threaded code and some in hard code, provided that shifting between modes is rapid.

#### 5. Conclusions

We have shown that under certain circumstances threaded code provides an attractive alternative to hard code, saving space at little cost in time.

*Acknowledgments.* The FORTRAN IV compiler for DEC's PDP-11 has been written to generate threaded code. In the course of that project many improvements have been suggested by those associated with it. Of particular value to the author have been the ideas of Ronald Brender, David Knight, Louis Cohen, Nick Pappas, and Hank Spencer.

Received June 1971; revised December 1972