CONVERSION OF A TOKEN THREADED LANGUAGE TO AN ADDRESS THREADED LANGUAGE

Bob Buege
RTL Programming Aids
10844 Deerwood SE
Lowell, MI 49331

ABSTRACT

After becoming spoiled by almost four years of experience with a token threaded language, I find it difficult to imagine ever going back to an address threaded language. Token threading gives me the flexibility of a metacompiler without the complexity. All code is relocatable. I can modify or delete words at the beginning of the dictionary without recompiling. I can eliminate the words not needed for an application before burning the application in ROM, and since garbage collection becomes a trivial problem, I can even eliminate the need for screens by decompiling object code and doing all my editing from RAM.

Many of the tools which I have come to depend on would be difficult or impossible to adapt to an address threaded language. However, address threaded languages do have a speed advantage over token threaded languages for most applications.

A compromise has been achieved which allows me to do all my development work in a token threaded language and then use self-modifying code to convert the language into an address threaded language for increased speed when the project is completed.

INTRODUCTION

There are many ways in which a token threaded language may be implemented. The language RTL (Relocatable Threaded Language) is based on 16 bit tokens in which each token is divisible by two (4 in 32 bit versions). The token is used as an index to one of two address tables to find the name or code assigned to each token. Each entry in the names table contains an address which points to a name in the names block. The corresponding entry in the code table contains an address which points to the code for that word in the code block of the appropriate vocabulary.

Since there is a one-to-one correspondence between tokens and code addresses and the tokens take the same amount of space as addresses, it is possible to go through the entire language and replace each token with the code address associated with that token.

This results in two advantages. There is an increase in speed because the inner interpreter gets the code address directly from the instruction stream instead of looking it up from a table and the memory required is sometimes less because the code table is not needed for turnkey systems which don't require the outer interpreter.

THEORY

It's very easy to see how the token threaded language works and how the address threaded language works. The difficulty arises during the conversion process. While the conversion process is taking place, part of the language is implemented as a token threaded language and part of the language is implemented as an address threaded language. In fact, the division usually occurs within a word so that part of a word is token threaded and part is address threaded. The inner interpreter must be able to distinguish tokens from addresses and make the appropriate response.

I simplify the task of the inner interpreter by ensuring that the words are converted to address threading in order of their location in memory. A marker is continuously updated so that it always points to the last address where a token was converted to an address. The inner interpreter can thus determine whether the instruction pointer is pointing to a token or an address just by comparing the instruction pointer to the last address which was converted to address threading. The steps for my conversion algorithm are as follows:

1. Declare space for a temporary table.

2. Make a list of all tokens which correspond to threaded words.

3. Sort the list so that the code addresses are in ascending order.

4. Set the marker to zero, indicating that all threaded words are token threaded.

5. Substitute a smarter interpreter which uses the marker to distinguish between token threaded code and address threaded code.

6. Go through each word in the list, replacing each token with its code address and adjusting the marker.

7. Replace the smart interpreter with an interpreter which expects all code to be address threaded code.

Step 6 of the algorithm requires the ability to distinguish between code and in line parameters. This problem has been treated in prior papers which I have given at the 1984 Rochester Conference and the 1984 FORML Conference and will not be treated in this paper.

CONCLUSION

There is a considerable loss in flexibility by the conversion to address threading which makes the language barely operable after the conversion is completed, the main problem being the fact that the compiler still generates token threaded code. This technique is therefore only useful in applications where the flexibility of a full language isn't needed in the finished application. A simple example of such an application is a utility which I wrote for a multiprocessing CPM system which accepts arguments from the command line to initialize various serial ports for differing baud rates, bits per character, stop bits, and parity.

The following procedure shows how such an application may be turned into an executable program.

1. The word COLD is redefined to perform the desired application and then return to the operating system. COLD is the first word executed when the language is booted.

2. The dictionary is separated into two vocabularies with all words initially placed in the second vocabulary.

3. The command KEEP COLD then moves the word COLD to vocabulary 1 along with all the words which COLD calls either directly or indirectly.

4. The various modules of the language are moved to appropriate areas of memory so that the variables block and the code for vocabulary 1 are adjacent and may be saved without saving unneccessary modules such as the names table, code table, names block, or code for vocabulary 2.

5. The language is converted to an address threaded language.

6. The portion of the language containing the variables block and vocabulary 1 is saved as an executable file.

REFERENCES

1. R. Buege, "Status Threaded Code", Proceedings of the 1984 Rochester Forth Conference.

2. R. Buege, "A Decompiler Design", Proceedings of the 1984 FORML Conference