OBJECT LINKAGE MECHANISM FOR THREADED INTERPRETIVE LANGUAGES

Yong M. Lee
Trenton State College

Donald J. Alameda, Jr.
Integrated Automata, Inc.

Keywords: threaded interpretive languages, object oriented programming, Forth, inheritance

Abstract

Several programming languages, most notably Forth, have been implemented as threaded interpretive languages. These languages usually offer static threading as the primary linkage mechanism. The demands of object oriented programming often require more flexible binding options. This paper discusses a linkage for TILs that supports multiple inheritance and various dynamic properties while preserving the other characteristics of these languages.

Introduction

The essence of a *threaded interpretive language* (TIL) is an inner interpreter that sequentially executes atomic machine language primitives in a manner analogous to the hardware instruction fetch cycle. The inner interpreter employs an *interpretive pointer* (IP), usually assigned to a dedicated register, that serves as a software program counter. The IP traverses a sequence of addressed cells called the *interpretive stream* (IS).

In this discussion, the atomic machine language primitives are called *code field routines* (CFRs). Once a TIL run time environment is launched, the hardware PC always remains inside a CFR. As a program runs, the current CFR executes, then passes control to the CFR for the next item in the IS. These short machine language routines are the software analogs of hardware instructions.

In Forth, the most well known of the languages typically implemented as TILs, the lexical unit is called a *word*. Each item in a Forth IS represents a word after it as been compiled. Forth words fall into two categories depending on the nature of their CFRs: *machine language primitive* (MLP) and *high level* (HL) words. The CFR for each MLP is unique and dedicated to that word.

The CFR for a HL word characterizes a unique word *type* and is shared among all words of that type. Basic HL word types include constants, variables and colon definitions. That means all variables share one CFR and all constants share another. The portion of a HL word that makes it unique is its *parameter field* (PF).

In the case of constant or variable words, the PF contains a numeric value. The PF of a MLP word is its dedicated CFR. Colon definitions are the most interesting of the basic word types. The PF of a colon definition is a discrete segment of IS code. In a typical Forth implementation, most of the words built into the system are colon words. A colon definition threads together other words, both MLP and HL, to create a new word representing the sequence. This threading gives the class of languages its name.

Indirect threading is the most popular threading technique found in TILs [1]. Its IS comprises a sequence of pointers to word bodies which in turn point to the execution addresses for machine code. Modification of the linkage in an indirect thread system requires patching the IS segment in the PF of a colon definition.

Vector threading and table threading sacrifice a little speed to provide more dynamic linkage. In these implementations, the IS is a sequence of cells containing the addresses of pointers to the word bodies or indices into a table of execution addresses [2]. In vector and table threading, link modification need not disturb defined colon words. In this way, the behavior of compiled code can be modified or extended at run time by other code written and compiled at a later date.

This paper describes a multiple table threading scheme, which the authors have dubbed *object threading*, that supports flexible link options, permitting several approaches to *object oriented programming* (OOP) language extension for TILs. The code examples presented to facilitate the discussion are written for the Intel 80x86 family of processors (specifically MS-DOS and the Microsoft Macro Assembler Version 5.00 or later) but are as easily or even more easily implemented under other architectures.

The implementation described here also takes advantage of 80x86 memory segmentation and the segment registers to provide natural boundaries between software objects. The 8086 memory addressing scheme uses four segment registers to divide the physical memory address space into four simultaneous flat sixteen-bit (64 Kbyte) address spaces called segments. Any or all of these segments may overlap to any degree or they may all be independent.

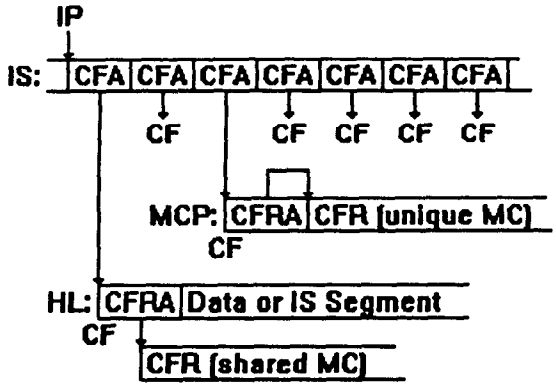
In this implementation, the 8086 code segment (CS) and stack segment (SS) registers are always equal and never change. All hardware stack operations and instruction executions reference this one sixty-four Kbyte segment. Therefore, the parameter

stack (P-stack), all the CFRs in the system and the cold start code must all reside here.

The PFs for HL words are defined with respect to the data segment (DS). Therefore, the IS as well as all data always exist within an 8086 data segment. In a simple sixteen bit Forth implementation, the DS would be set equal to the CS and SS registers and all operations would share the same address space. None of the implementations discussed here require any special usage for the fourth segment register, the extra segment (ES) register.

In the object threading implementation described here, the DS register changes to map various different high level code contexts. A *threaded segment* is identified by a unique DS value and can occupy up to sixty-four Kbytes of physical memory. Each object is a threaded segment as is the set of high level words built into the system (which is not an object).

The ANSI standard for the Forth programming language does not specify or require any type of threading [3], but many Forth systems are still implemented as TILs. Therefore, one design goal for the object threading linkage mechanism was that it provide a suitable substrate for Forth implementation.



IS	Interpretive Stream	CF	Code Field
MC	Machine Code	CFA	CF Address
MCP	MC Primitive Word	CFR	CF Routine
HL	High Level Word	CFRA	CFR Address

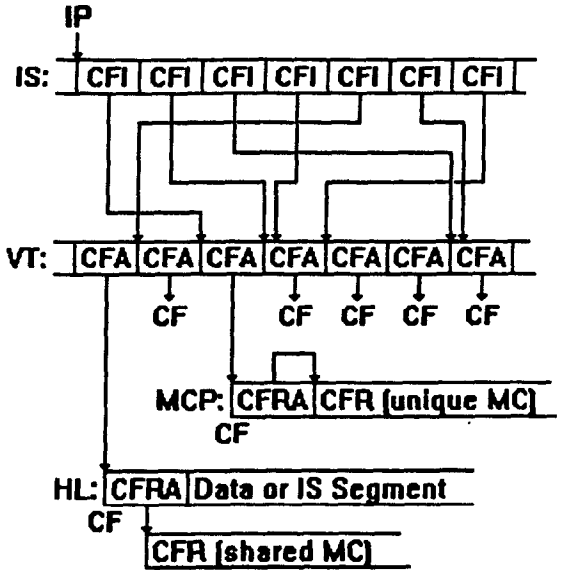
Figure 1: Indirect Threading

The Inner Interpreter

Just as the hardware PC must always point to valid executable machine code for correct program performance, the TIL IP must always point to a cell within an IS thread in a colon word PF. The inner interpreter is usually called NEXT because it uses the IP to fetch the next interpretive token just as the hardware instruction fetch logic uses the program counter (PC) to fetch the next machine instruction. NEXT advances the IP and redirects the hardware PC accordingly by jumping to the machine code CFR related to the current interpretive cell.

In some implementations, NEXT is a macro that is expanded when required. In other systems, including the one described here, NEXT is a shared piece of code and the system employs a jump to invoke NEXT.

In a word body, a pointer to the CFR for that word is called a *code field* (CF). The address of a code field is called a *code field address* (CFA). For indirect threaded code, the IS is a sequence of CFAs [1]. Figure 1 illustrates the relationships between the elements of indirect threaded code. Figure 2 illustrates those relationships for table threaded code.



IS	Interpretive Stream	CF	Code Field
VT	Vector Table	CFA	CF Address
MC	Machine Code	CFI	CF Index
MCP	MC Primitive Word	CFR	CF Routine
HL	High Level Word	CFRA	CFR Address

Figure 2: Table Threading

CFRs, both primitive and characteristic, usually end in an invocation of NEXT. Figure 3 illustrates an indirect threaded NEXT in a sixteen bit implementation of Forth for an Intel 80x86 processor.

```
NEXT:
        LODSW
        MOV     BX,AX
        JMP     word ptr [BX]
```

Figure 3: Indirect Threaded Inner Interpreter

In all of the implementations described here, the SI register serves as the IP. In the indirect threaded NEXT, the address loaded into AX is the CFA for the next word in the IS. The CFR pointer is transferred to the base register, BX, to permit an indirect jump to the CFR.

Figure 4 illustrates a table threaded NEXT in a sixteen bit implementation of Forth for an Intel 80x86 processor. This time the sixteen bit cell loaded into AX is an index into a table

(CFA_TABLE) that contains the CFAs for every word in the system. Consequently, each word is represented by a unique index. Because each CFA is two bytes long, BX is added to itself to convert the index into a table offset. The NEXT routine then loads the CFA from CFA_TABLE into BX allowing an indirect jump to the appropriate CFR.

```
NEXT:
        LODSW
        MOV     BX,AX
        ADD     BX,BX
        MOV     BX,CFA_TABLE [BX]
        JMP     word ptr [BX]
```

Figure 4: Table Threaded Inner Interpreter

Word Vector Tables

Object threading is similar to table threading in that the interpretive cells in its IS represent table indices rather than hardware addresses. However, the cells in the vector table used in object threading are more complex than the simple CFAs of table threading. Figure 5 illustrates the structure of a *word vector* (WV) in a *word vector table* (WVT).

CFR	Code Field Routine
PF	Parameter Field

Figure 5: Object Threading Word Vector

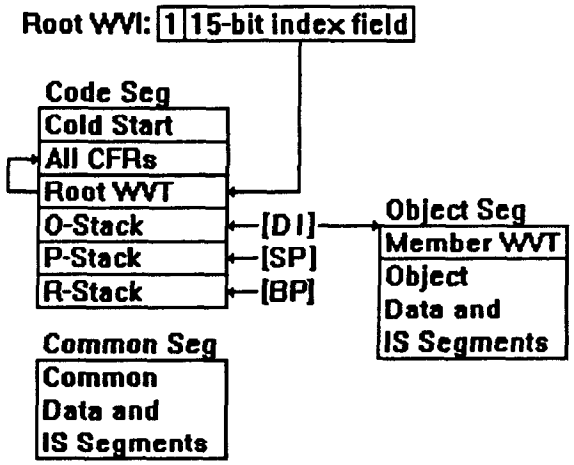
A WV is a two dimensional vector. It has three distinct sixteen bit fields: two address offset values and one address segment value. The first part is the familiar CFR execution address with a range of sixty-four Kbytes. The next two fields together locate the associated PF within a specified threaded segment. The PF dimension of the vector has the entire 8086 segmented address space as its range.

The threaded segment that contains the parameter fields for the high level words built into the system is called the *common threaded segment*. All other threaded segments are associated with objects and are called *object threaded segments*.

The *root words* are the set of machine code primitive words together with the set of common threaded words plus all extensions to root word lists. They are accessible to all other words through the normal outer interpreter source stream at compile time and the IS at run time. All words in standard Forth word lists (CORE, CORE EXT, DOUBLE, FILE, etc.) would be implemented as root words.

Object threaded code segments each define an object. One feature of an object threaded segment is that it contains a WVT while the common threaded segment does not. In fact, the WVT in an object threaded segment is at the base (offset zero) of the segment address space. Forth words whose PFs are located via object segment WVTs are called *member words*.

Because an interpretive token in an object threaded IS serves as an index into a WVT, it is called a word vector index (WVI). The high order bit of a WVI is called the *root bit* and is used as a flag to the NEXT routine.



WVI	Word Vector Index
WVT	Word Vector Table
CFR	Code Field Routine
IS	Interpretive Stream

Figure 6: Root WVI CFR Address Translation

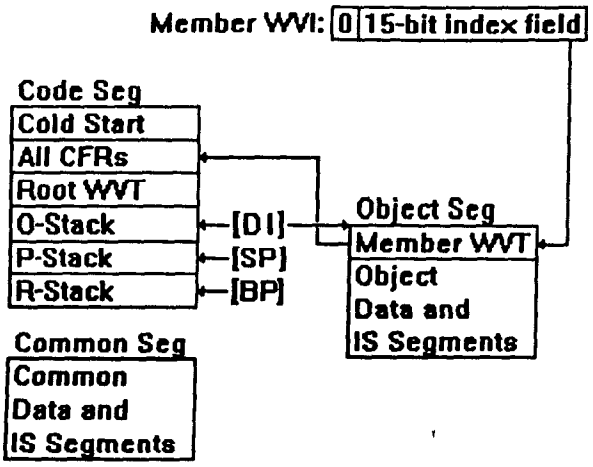
The root bit distinguishes two classes of WVI. A *root WVI* (one with its root bit set) is the index for a root word and is unique within the system. It always refers to the same word. The root WVT resides in the code segment along with all of the CFRs for the system (even the CFRs for member words). A *member WVI* (one with its root bit clear) is the index for a member word and is interpreted in the context of the current object. Figure 6 illustrates CFR address translation for root WVIs and figure 7 does the same for member WVIs.

To maintain object context, a third stack (the O-stack) is introduced in addition to the standard P-stack and R-stack. The 80x86 SP, BP and DI registers are reserved as the P-stack, R-stack and O-stack pointers respectively. All three stacks in the described implementation reside in the code segment.

Current object context is defined by the value at the top of the O-stack which is interpreted as the DS segment register value that maps the current object threaded segment, and consequently the current object WVT. Entire objects (including their threaded segments and WVTs) are identified by *object words* which, when executed, change context to the associated object. Because they must be available to the public at all times, the object words themselves are root words and are compiled into the common threaded segment.

The table threaded NEXT processes a very simple table. Each entry is simply the CFA of the indexed word. The location of the associated PF is calculated from the CFA value. The WVT

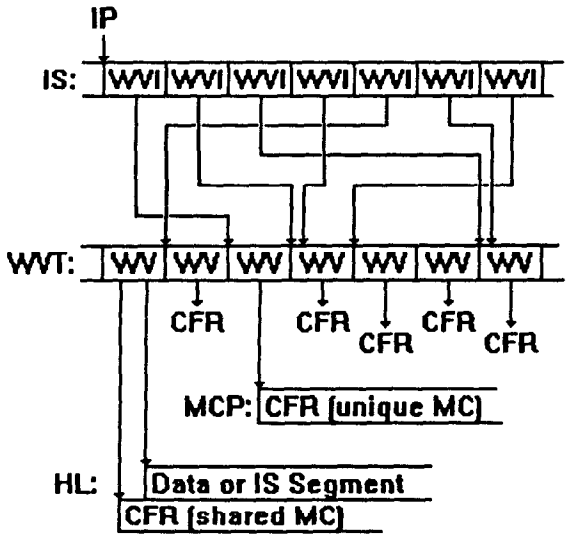
entries employed by object threading in the example 80x86 implementation contain more information. Each entry is six bytes long to support the WV structure described above.



WVI	Word Vector Index
WVT	Word Vector Table
CFR	Code Field Routine
IS	Interpretive Stream

Figure 7: Member WVI CFR Address Translation

Figure 8 illustrates the relationships between the elements of object threaded code. Figure 9 illustrates an object threaded NEXT in a sixteen bit implementation of Forth for an Intel 80x86 processor. The most striking feature of the object threaded NEXT is that it has two distinct branches. The first portion handles root words; the remainder handles object member words.



WV	Word Vector	MC	Machine Code
WVI	WV Index	MCP	MC Primitive Word
WVT	WV Table	CFR	CF Routine
		HL	High Level Word

Figure 8: Object Threading

Like the other 80x86 NEXT routines described above, the object threaded NEXT moves the interpretive token, a WVI in this case, from the IS to the AX register and advances the IP. Then it tests the root bit in the WVI and branches to the member word handler if it is clear. If the WVI root bit is set, the index value is doubled to discard the root bit and placed in the base register allowing an indirect jump to the appropriate CFR via the CFR offset field in the selected WV in the root WVT.

Because the size of each WV is six bytes, the fifteen bit index fields in WVIs must be exact multiples of three. Any other value is illegal and undefined. Assuring that WVIs are generated in this manner is a simple matter for the outer interpreter which performs colon definition compiling.

```
NEXT:
        LODSW
        TEST    AX,ROOT_BIT
        JZ      NEXT1
        MOV     BX,AX
        ADD     BX,BX
        JMP     word ptr CS: ROOT_WVT [BX]
NEXT1:
        MOV     BX,AX
        ADD     BX,BX
        PUSH    DS
        MOV     DS,CS:[DI]
        MOV     DX,[BX]
        POP     DS
        JMP     DX
```

Figure 9: Table Threaded Inner Interpreter

Member word handling is similar. Again the WVI is doubled to discard the root bit and transferred to the base register. Because member WVIs are translated through the current object WVT, the value from the top of the O-stack (at `CS:[DI]`) provides the DS to locate the object segment WVT. The CFR offset field in the selected WV in the object WVT is used for a direct jump (within the CS).

Placing the CFA (CFR offset field) in the WV instead of keeping it with the word body is not a requirement for object threading. This design decision was made for the described system because of the idiosyncrasies of the Intel 80x86 segmented memory architecture. The essence of object threading lies in the selection of a WVT based on object context, not in any strict segmentation of memory or the structure of the WV. In fact, many styles of WV are possible for an 80x86 object threaded implementation.

In flat addressing and virtual addressing schemes, especially with thirty-two bit or larger addresses, the WVT may simply contain long form CFAs as in simple table threading. In this case, the cell that contains the CFR address for a word would still be part of the body of that word and location of object WVTs would require other conventions and address computations.

In the system from which the object threaded inner interpreter example was derived, CFRs for HL words exist in two variants: one for root words and one for member words. Again, this division was the result of implementation decisions and is not a requirement of object threading. However, object threading *does* introduce one new CFR: the machine code to support the HL object word type. The CFR for object words pushes the segment address for the object onto the O-stack. Until that address is popped, or another object address is pushed on top of it, all member WVI references are interpreted via the WVT belonging to that object by default. In flat addressing implementations, the value on the O-stack would be the address of the base of the object segment, which would also be the address of the WVT for that object.

Inheritance

While a member word executes (the IS is within an object segment), both root and member WVIs are encountered. If the PF vectors within each of the HL WVs in an object segment WVT all refer to only the root common threaded segment and the object itself, the object is independent of all other objects. However, object threading allows an object WVT to refer to PFs in any threaded segment (the set of all object segments plus the common segment). Therefore, for one object segment to inherit behavior from another, it may simply map the parameter field of a colon definition within the other object segment.

So far in this discussion, no distinction has been made between an object and an *instance* of an *object type* or *class*. The reason for deferring that discussion until now is that object threading supports many variants of object implementation. Object threading directly supports encapsulation and other OOP language features, most notably overloaded operators and late binding. But any object design, with or without class support, even with late binding, can be implemented with indirect or table threaded code. A TIL designer would choose object threading for compilation speed, execution efficiency and ease of outer interpreter design and implementation.

In the simple type of inheritance described above, the WVs for each of the WVIs compiled into the inherited word must have the same meaning (same values in the same positions) in each of the two WVTs. Overriding an inherited member word means that the WVI for that member word must have analogous but different meanings (values at the overridden position) in the two WVTs. For class instance support, the colon member WVs (object behavior) would refer to an external object segment while the data member WVs (object state) would refer to the local object segment. Object threading can not enforce such relationships: the inner interpreter only provides the linkage. The proper construction of object segments and WVTs for object type, instance and inheritance support is an outer interpreter issue coupled to the specific semantics of the object relationships to be enforced.

There are many options for outer interpreter implementations of systems employing object threading. The first consideration is that the outer interpreter should not recognize the name of a member word without an object instance context. The most obvious method is to employ a scan forward technique where the token following an object word is plucked from the source stream and lookup is performed in a special word list that belongs to the object word. The outer interpreter must force the compilation of WVIs for the object word, the member word and, unless object context is to be transferred to the referenced object, the MCP root word that performs a return from object.

Separating class and instance makes inheritance much easier to implement. While an object requires both class behavior and instance state to function, inheritance is a property of class behavior only. In a class and instance implementation, class words serve as the defining words used to create instance words and construct their corresponding object segments. Under this design, instance words become the agents that set object context to their own object segments. In the system described, instance object segments contain only base relative data member words [4] and the class object segments contain only colon member words referring only to base relative data in the current instance object segment and the P-stack.

Multiple inheritance presents a more difficult problem. The described object threaded system supports multiple inheritance by further segmenting object segment WVTs with parallel changes in WVI structure. For member words, the fifteen bit index field in the multiple inheritance WV is subdivided into a seven bit member offset and an eight bit parent class offset. Root word WVIs do not change and the root WVT is not segmented. The additional overhead this change introduces into the inner interpreter affects the member word branch only.

Conclusion

Neither simple nor multiple inheritance require a special linkage. In a TIL, these features can be supported by appropriate outer interpreter design. The strengths of object threading result from supporting encapsulation and inheritance at the IS level. Consequently, object threading excels at run time binding and other deferred binding support. Well behaved words that manipulate class or instance object segment WVTs at run time are easily designed.

Address correspondence to:

Yong M. Lee
Computer Science Department
Trenton State College
Trenton, NJ 08650

References

[1] Biggs, T. L. *iaForth Programmer Reference Manual.* Lawrenceville, NJ: Integrated Automata, Inc., 1985.

[2] Alameda, D. J. and T. L. Biggs. *ATIL Design Handbook.* Upper Montclair, NJ: Integrated Automata, Inc., 1991.

[3] Woehr, Jack. *Forth: The New Model.* San Mateo, CA: M&T Books, 1992.

[4] Lee, Y. and D. J. Alameda. C++ Style Class Support Under FIG Forth. In *Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 341-345, 1994.