Language Coprocessor Boosting the Execution Speed of Threaded Code Programs

Eddy H. Debaere
Electronics Laboratory
State University of Ghent
St.-Pietersnieuwstraat 41, B-9000 Ghent, Belgium

Abstract

Threaded code is an interpretive language implementation technique which favors execution speed and program representation size. However, it causes considerable overhead when executed on standard microprocessors. In this letter we propose a coprocessor accelerating the execution of programs represented in threaded code, while maintaining hardware and software compatibility with the existing microprocessor environment.

Introduction

Several high level language implementations have used an intermediate language to lower their representation sizes (e.g. Modula-2 and M-code, Pascal and P-code). The threaded code technique ([1], [2]) fulfills this objective by using compactly coded intermediate instructions, and by sharing 'subroutines' where possible.

In both cases a software interpreter is needed when the program is executed by a standard microprocessor. In contrast to the wired fetch/decode mechanism of native code execution, the fetching and decoding of an intermediate instruction is performed by machine instructions. Hence, the sematic gap between the virtual and the native architecture causes considerable overhead.

In previous work [3] we have developed the concept of a *language coprocessor* (LCP) for traditional intermediate languages such as M-code and P-code. These intermediate languages are the machine code for a virtual stack machine. Programs are linear lists of instructions, and sequencing is controlled by means of jump instructions. The language coprocessor fetches and decodes the intermediate instructions and generates the corresponding semantic instruction stream for the microprocessor, which executes it in parallel. The fetch/decode mechanism for such intermediate languages is fixed and simple.

Threaded code (TC), on the other hand, is hierarchical and can be represented by a tree. The *leaves* of the tree correspond to primitives, routines coded in machine language implementing primitive operations. The *nodes* of the tree are routines (secondaries) which call either primitives or other nodes. The program execution consists of traversing the tree and executing the encountered primitives.

Compared to traditional intermediate languages, the fetch/decode algorithm of threaded code is more complex and corresponds to a tree-traversal algorithm. The interpretation overhead per semantic routine is higher, and is highly program dependent. Nevertheless, threaded code implementations are becoming increasingly important. Forth [4], a Reverse Polish Language, is the most widely known language using this technique. Moreover, as Forth primitives may be used to implement other high level languages [5] they play a role as a universal assembly language and deserve special attention.

Among several varieties of TC [6], indirect threaded code (ITC) and token threaded code (TTC) are processor independent. The ITC is widely used. The secondaries (nodes) of the ITC tree contain addresses which *indirectly* point to primitive or other secondary routines. The execution of such program operations requires a large number of memory indirect memory references and memory indirect jumps.

Dedicated microprogrammed microprocessors have been developed to execute threaded code representations [7]. Recently, *special purpose* architectures have been designed ([8], [9], [10]) to support threaded code execution. These architectures contain a separate return stack (containing the tree traversal information), a data stack (of the stack evaluation machine), and a program representation space.

It is not our intention to develop an architecture superior to these specialized architectures. Rather, we want to accelerate the execution of threaded code in a *standard* environment and at a low additional cost.

Language Coprocessor For Threaded Code

In our LCP-approach, the code (intermediate program representation), data stack and return stack all reside in the processor memory. The program representation for the CPU-LCP combination is identical to the representation used by a software interpreter.

The CPU accesses the LCP as a piece of memory mapped into its address space. The LCP generates an instruction stream triggered by the code requests from the CPU. Intermixed with the CPU code and data bus cycles, the LCP performs DMA cycles to traverse the tree and to acquire the identification (code address) of the primitive to be executed next. As the tree depth is highly program dependent, the number of DMA cycles required per primitive is not fixed.

In a TC implementation, the actions traversing the tree (finding child and parent) may be considered as true primitives. In Forth the primitives changing the level in the tree correspond to the '**:**' and '**;**' symbols. The '**:**' selects a deeper level of the program tree; the '**;**' leads back to a higher level. Each primitive ends with a `NEXT` action which gets the next Forth word. The information necessary to traverse the tree and to generate the semantic routines is kept in the LCP RAM.

As the interpretation mechanism of TC programs is more complex compared to traditional intermediate languages, the LCP needs more resources. By providing three registers (Instruction Pointer, Word Address and Return Stack Pointer), and by microprogramming the traversal algorithm, a new threaded code interpreter results.

Such a CPU-LCP configuration for the execution of threaded code has several advantages. It speeds up the traversal of the program tree at a low hardware cost. Literals may be directly embedded into the generated instruction stream. Coprocessing with other coprocessors is still possible (e.g. floating point operations may be performed, using standard numeric coprocessors). The instruction stream generated to the CPU is linear and the frequent jump instructions, inherent to software TC interpretation, are eliminated. The standard environment is unhampered and the coprocessor can be used for other languages by changing the LCP RAM contents.

A Language Coprocessor Prototype For Threaded Code

The prototype described in previous work [3] has been redesigned. The first prototype only boosts the execution of traditional, linear intermediate languages, such as M-code, in a standard

5MHz Intel 8086 microprocessor environment. The new prototype supports both the interpretation of Modula-2 and Forth programs. For the remainder of this note, however, we shall only mention results for Forth programs.

The hardware of the prototype is built out of 12 PALs (Altera EPLD900) and two standard RAM chips (8Kbyte each). The memory mainly contains microcode. These microinstructions (13 bit wide) are chosen such that they provide a complete and minimal set of commands to perform the coprocessor operations. Any application (or a part of it) which may be expressed in terms of these microinstructions may benefit from the same hardware and achieve a higher performance. During its design only those operations of the interpretive process which are very frequent and subject to considerable increase in performance at a low additional hardware cost were allocated to the coprocessor.

Fig. 1 illustrates the internal architecture of the LCP prototype. We may destinguish between two main LCP parts: the Bus Interface Unit (BIU) and the Execution Unit (EU).

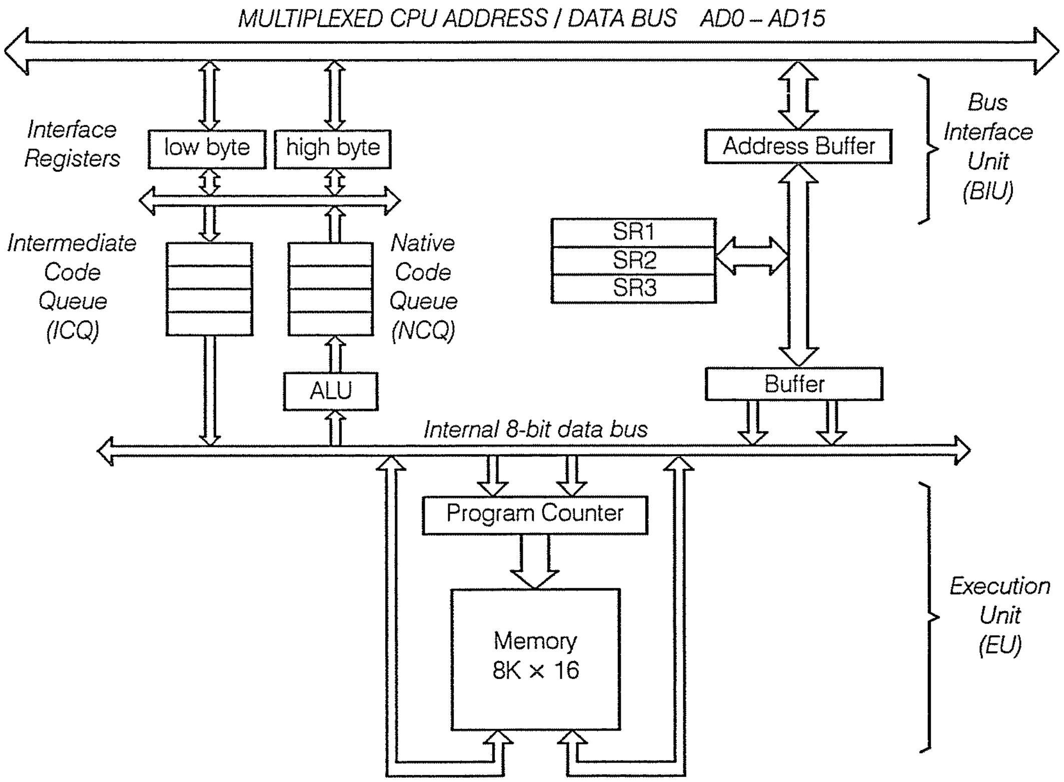


Fig. 1. The internal architecture of the proposed language coprocessor.

The BIU is concerned with LCP bus operations (tree traversal) and forms the CPU-LCP interface (instruction generation). It traverses the tree as commanded by the EU control vectors. When executing ITC, it saves the encoutered code addresses in the Intermediate Code Queue (ICQ). When the CPU requests code from the LCP space, the contents of the Native Code Queue (NCQ) are transferred to the CPU bus.

During the tree traversal, pointers are kept in internal registers (SR1, SR2, SR3). The CPU may consult, or modify these registers using standard CPU memory operations. These register contents may also be inserted in the instruction stream as well as intermediate code literals from the ICQ. A simple ALU may perform primitive preprocessing operations such as shifting and padding.

The EU decodes the ICQ bytes and generates corresponding control vectors which activate either CPU code generation into the NCQ, or tree traversal actions. The EU is implemented as a finite state sequencer, combined with a writeable control store.

As the traversal/fetch/decode algorithms are fully microprogrammed in carefully tailored micro-instructions, the LCP architecture is flexible and can support the execution of traditional intermediate languages (M-code, P-code) as well as of ITC and TTC programs.

Performance

Fig. 2 illustrates the performance of the language coprocessor prototype. The tree structure of two different Forth programs which compute the function FUNC$(x,y) = 2x^2 + y^2$ of two integers x and y residing on the stack, is presented. These programs were executed by the LCP-i8086 ITC interpretive machine and by a commercial Forth software interpreter **[11]**, on the same harware. The executed semantic primitives are identical in both executions.

The first program (fig. 2a) contains three secondaries (root included) and five primitives (we assume '*' implemented as a primitive). The LCP-CPU configuration is 1.8 times faster than the software interpreter. (The execution times are measured from the prototype).

The second program (fig. 2b) consists of the same five primitives and one root secondary. Hence the overhead is lower and we may expect a lower speed up. A value of 1.5 was observed.

Another benchmark program lacks semantic operations, but has $n+1$ levels in its tree. It represents a nested empty procedure calling. The performance gain is given in Fig. 2c for varying n and approximates 3.1 for large n. In this respect, it should be noticed that structured programming styles emphasize the use of subroutines. Adherence to this style leads to deep program trees.

Conclusion

Previous research has resulted in a language coprocessor for the interpretive execution of M-code programs. The current work extends the scope to the execution of programs represented in processor-independent threaded code. The performance gain is highly program dependent, and varies between 1.5 and 3.1 measured on an, as yet not optimized, language coprocessor prototype for the considered threaded code Forth programs. Although we only considered 8086-environments, the LCP-concept may be used in other microsystems as well (such as in a 80386-system) and will also contribute to an increased performance.

References

[1] Bell, J.R., "Threaded Code." *ACM Communications*, Vol. 16, No. 6, 1973, pp. 370-371.

[2] Dewar, R.B.K., "Indirect Threaded Code." *ACM Communications*, Vol. 18, No. 6, 1975, pp. 330-331.

[3] Debaere, E. H., "Language Coprocessor for Interpretive Execution of Modula-2 Programs." *IEE Electronics Letters*, Vol. 22, No. 24, 1986, pp. 1302-1304.

[4] Ouverson, M., *Dr. Dobb's Toolbook of Forth*. M&T Books, USA, 1986.

[5] NOVIX Inc., "NS4100: Novix Small C". Data Sheet, 1987.

[6] Ritter, T. and Walker, G., "Varieties of Threaded Code for Language Implementation." *Byte*, Vol. 5, No. 9, 1980, pp. 206-226.

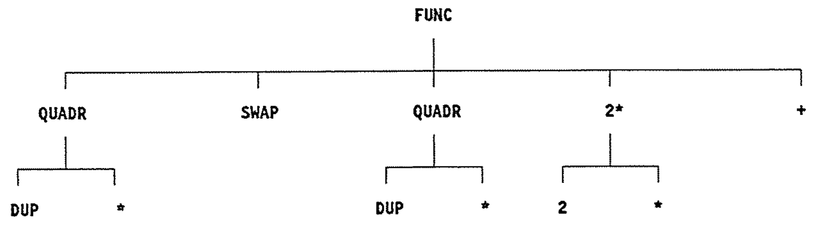
[7] Dumse, R., "The R65F11 Chip." *Forth Dimensions*, Vol. 5, No. 2, July-Aug. 1983, pp. 25.

[8] Bursky, D., "Optimized Processor Handles Forth, and More." *Electronic Design*, Vol. 34, No. 8, Nov. 1986, pp. 47-48.

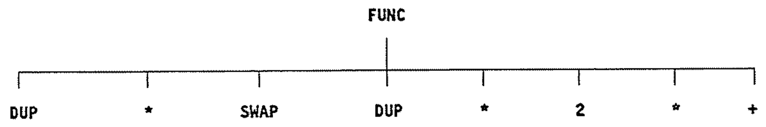
[9] Koopman, P. and Haydon, G., "MVP Microcoded CPU/16 Architecture." *JFAR*, Vol. 4, No. 2, 1986, pp. 277-280.

[10] Moore, C. and Murphy, R. W., "Under the Hood of a Superchip: the Novix Forth Engine." *JFAR*, Vol. 3, No. 2, 1985, pp. 185-188.

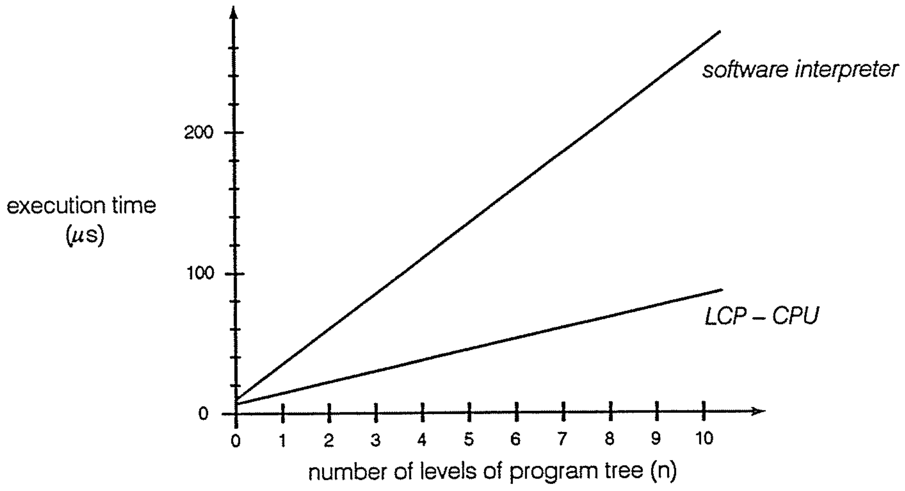
[11] Laxen, H. and Perry, M., *Forth 83 implementation for IBM Personal Computer*, Version 2.1.0.



```
    : QUADR  DUP * ;
    ; 2*     2   * ;
(a) : FUNC  QUADR SWAP QUADR 2*  + ;
```



```
(b) : FUNC  DUP * SWAP DUP * 2 * + ;
```



```
     : levØ ;
     : lev1 levØ ;
     : lev2 lev1 ;
       ...
(c)  ( : levn  lev(n-1) ; )
```

Fig. 2 (a) (b): alternative versions and corresponding tree representations of a Forth program computing the function FUNC $(x,y) = 2x^2 + y^2$;

(c): performance comparison between LCP and software interpretation of nested empty procedure calls.

Eddy Debaere received an M.S. degree in electrical engineering, a B.S. degree in computer science, and a Ph.D. in applied sciences from the State University of Ghent, Belgium, in 1984, 1986, and 1989 respectively. His fields of interest include hardware support for high level languages and interpretation by microprocessors. He is the author of ***Interpretation and Instruction Path Coprocessing****, published by MIT Press, Cambridge, 1989.*