UDC 681.3-181.4:681.3.06

STANDARDIZATION OF MICROCOMPUTER SOFTWARE USING VIRTUAL-MACHINE DESIGN[1]

Yu. V. Matiyasevich, A. N. Terekhov, B. A. Fedotov

Translated from "Avtomatika i Telemekhanika", No.5, pp.168-175, May 1990

The paper describes the architecture and the instruction set of a virtual machine designed for icrocomputer implementation of algorithmic high-level languages with static type checking (Ada, Algol, Modula 2, Pascal, and others).

1. INTRODUCTION

Hundreds of different microcomputers with various architectures are currently in use. Some of these microcomputers are quite fast but have a small directly addressable random-access memory. The development of compilers or cross-compilers from algorithmic high-level languages, file systems, text editors, and other software tools for all microcomputers is a very difficult problem. The following technique may be applied for this purpose: create a single standard architecture and instruction set for some virtual machine (VM) and design all the software tools for this VM only; then implement VM interpreters on each particular microcomputer [1]. This approach, in addition to software standardization, usually achieves a 3-5 times saving of RAM at the cost of a 2-4 times slowdown in computations.

The idea of using a VM is not new. Recall the simple stack machines created by Wirth for the implementation of Pascal [2]. Subsequently Wirth created a more modern VM design for Modula 2. This VM was hardware-implemented as the computer LILITH [3]. Similar work is being carried out at the Computational Center of the Siberian Division of the Academy of Sciences of the USSR [4].

The first VMs had a fairly primitive design in order to simplify the interpreter implementation for new computers. For example, the well known P-code [2] included loading into a stack and unloading from the stack into RAM, four arithmetic operations, conditional jumps, and a number of auxiliary instructions (fewer than 10). Although the LILITH instruction set [3] comprised nearly 250 instructions, most of the codes were for short instructions packed into one byte with the addresses (e.g., load into stack a local variable with address 0,1, ..., 15, dittofor a global variable, etc.). A single action in a high-level language is executed by several small instructions, while on the other hand an excessively strong influence of a single language is felt (Modula 2 in this particular case), and therefore many of the architectural features are uncomfortable even in related languages.

This paper is a continuation and further development of [5], which originally proposed a VM architecture and instruction set designed for a class of algorithmic high-level languages with static type checking (Pascal, Modula 2, Algol 68, Ada, and similar languages). For this class of languages, it was possible to standardize the representation of the main data types, memory allocation schemes, procedure call organization, and array manipulation. The clear separation of duties and responsibilities between the compiler from the high-level language to VM codes and the VM interpreter for a specific computer made it possible to simplify the VM structure (e.g., items checked in compile time are never checked again in execution time). The compiling process was

---

[1] Leningrad Branch, V. A. Steklov Mathematical Institute, Academy of Sciences of the USSR. Scientific Research Institute of Mathematical Methods of the Leningrad State University. "Krasnaya Zarya" Scientific Programming Office, Leningrad. Translated from Avto-matika i Telemekhanika, No. 5, pp. 168-175, May, 1990. Original article submitted November 5, 1988.

substantially simplified by the use of inverse Polish notation in object code, the use of a stack for computations, the provision of special instructions for the most frequent high-level language constructs, and the orthogonality of the VM instruction set.

Orthogonality of the instruction set is understood in the sense of uniformity of instruction types and addressing types and absence of exclusions and nonstandard actions. For instance, if a transfer instruction may have nine types of sources and three types of destinations, then there should be a total of 27 transfer instructions. If there are three types of addition instructions (A + B, A+ := B, A[I]+ := B), then the same types of instructions should be provided for multiplication, for logical operations, etc. On ES computers say, formula generation is a major part of any compiler, because integer addition is performed by one technique, decimal addition by another technique, and logical addition by yet another technique; the instructions AH, SH, MH are provided, while DH (half-word division) is missing. With an orthogonal design, the instruction set may obviously include rarely used commands, but this is necessary in the interest of simpler compiling.

1.1. Modifications. The following aspects of the VM were modified compared with [5].

a) The register stack was eliminated. The arithmetic expressions are evaluated in a stack which overlaps the procedure static space (i.e., the memory allocated to the procedure identifiers and work area). As a result, we avoid quantitative restrictions on stack depth, simplify the call scheme, reduce the variety of instructions, and no longer restrict the stack top to single-word values.

b) The representation of Boolean and real variables, the coding of characters and the specification of base registers is not fixed in the VM description. The compiler should be tuned to a specific VM interpreter by introducing appropriate interchangeable procedures.

c) The representation of the array ticket is hidden from the compiler. The array construct is supported by array generation instructions (allowing multidimensional arrays), array assignments, copy instructions, creation of arrays from separate elements, evaluation of array bounds and array element addresses. The compiler always works with a reference to the array ticket and never overrides standard instructions to manipulate the ticket attributes. Other VMs provide only a rudimentary support of array manipulation.

Cross-compilers are currently available for the VH assembler language, Pascal, and Algol 68; these compilers will be transported to the VM using generation methods. All the points with explicit ties to a particular computer architecture and OS have been identified (their number is quite small). When the software tools are transported to a new computer, these tie points will be reprogrammed.

The VM is often useful not only from standardization considerations. Many process control problems and other real-time applications impose memory constraints, while not more than 10% of programs are time-critical. It is therefore better to use two cross-compilers in each system: time-critical programs should be compiled directly into a special-purpose computer code, while all other programs should be compiled into virtual computer code with subsequent execution by interpretation. This technique will save hundreds of thousands of RAM bytes.

2. THE MAIN VIRTUAL-MACHINE ELEMENTS

We start with an informal description of VM architecture. The formal model (which is not given here) has been implemented in Algol 68 and may be used as the basis for VM implementation on various object computers.

The VM architecture includes a set of internal registers and several types of memory, specifically: program memory, work memory, and a call stack.

2.1. Internal Registers of the VM Processor. Several registers are provided (pointers to program and work memory, counters), which are used by the VM processor for interpreting the program code and are not accessible directly from the VM program. These registers include G — a

pointer to the beginning of the static space of the program proper; L — a pointer to the beginning of the static space of the currently executing procedure; X — a pointer to the end of the current static space (the top of the value stack); PC — a pointer to the current byte of the code being interpreted; IR — executing instruction register; PN — current procedure number register; SP — pointer to the call stack top; DT — pointer to the array field top; HT — pointer to the heap top; LINE — a register that provides linkage to the source-language program.

Some registers may be missing in specific VM implementations (e.g., IR or LINE), while additional registers may be included.

2.2. Program Memory. This memory stores the executing VM programs and is a read-only memory. The program memory is divided into two functional parts: interpreted code and procedure table.

The interpreted code is a homogeneous array of 8-bit bytes. The interpreted code is byte-addressable. No equalization of instructions or their operands is required. The access to the interpreted code is through the pointer PC, which always points to the current code byte. When this byte is selected, the pointer is advanced to the next byte.

Specific values of instruction codes on the level of VM description are not fixed. If for some reason the standard instruction coding is unsatisfactory for the VM-interpreter implementation, an alternative scheme may be used.

The procedure table has one input, which may receive a number from 0 to 255. The table output is a three-field structure: EP is the address of the procedure entry point (the initial value of PC for the procedure); PL is the packet length of the procedure parameters; XLIM is the required memory in the procedure static space.

The procedure table is used by the call instruction, whose argument is the procedure number. The procedure is entered through the table.

This technical solution makes it possible to shift the interpreted code or its component procedures in memory without any changes, by simply adjusting the procedure table. Moreover, it allows dynamic procedure loading in interpretation time.

The field lengths in the procedure table and the table organization (whether as a single table or as several compatible tables) are not fixed on the level of VM architecture description.

2.3. Work Memory. Unlike program memory, work memory may be used for both reading and writing.

The least addressable element in work memory is a 16-bit word. The entire memory is a linear arrow of such words.

When the VM is implemented on a computer with byte memory organization, a VM word may be identified with two adjacent bytes 2N and 2N + 1 in real memory, using 2N as the address.

The address range is limited by the 16-bit representation. Only one distinguished address is used: specifically, the number 0, which causes a halt when accessed. This address is used to represent the value NIL.

The bits in a word are numbered 1 to 16. The order of numbering of the bits is fixed in the VM-interpreter implementation and should be consistent with integer representation.

The contents of a data word may be interpreted in four different ways:

1) as a variable of type bits, i.e., as 16 independent bits; 2) as a variable of type bool: all the information is stored in a single bit, and the values of the other 15 bits are irrelevant; 0 in the significant bit is false, 1 is true; the number of the significant bit is fixed in the VM implementation;

3) as a signed integer represented in two's complement form; bit 1 is the sign bit, bit 16 is the weight 1 bit (units), bit 15 is the weight 2 bit (tens), and so on;

4) as a word address, i.e., an unsigned number.

Some words may contain service information (e.g., array tickets) required by the interpreter. The VM program does not use service information directly (the VM program is a program in VM

code).

Word pairs may be interpreted as floating point numbers.. The representation of such numbers is fixed in the VM-interpreter implementation.

The interpretation of the data words is determined by the instruction that processes these words. Since the author of the VM program in most cases is a compiler with static data type checking, there is no need for additional validity checking of the data.

The work memory is divided into four fields: the static field, the array field, the heap, and the string pool. The relative position of these fields is determined by the VM implementation.

The static field is a connected memory space. Three pointers are associated with this field: G always points to the beginning of the static space of the program proper, L points to the beginning of the static space of the currently executing procedure (if the program proper is executing, L coincides with G), and X points to the end of the current static space. The pointer L is moved by procedure calls and returns from procedures. The pointer X is used for implementing a stack in the current static space. It is modified by instructions that manipulate the stack top and explicitly by the instruction SETX.

The stack is manipulated in the ordinary pushdown mode, i.e., the instructions may read a value from the stack top and write a value into the stack. When a value is reach, the pointer X is decremented by the length of the value; when a value is written, the pointer is incremented; the point X is moreover decremented by one word when indirect addressing of the stack top is used. Stack overflow is the responsibility of the author of the VM program.

The static field is addressed in instructions relative to the pointers L or G. In this case, the one-byte instruction operand contains a local/global addressing attribute in one of the bits (0 corresponds to L, 1 to G), while the other seven bits give the offset in words relative to L or G. Which bit is used as the attribute depends on the VM-interpreter implementation.

Dynamic checking of static field sufficiency is performed by the VM: when a call instruction is executed, the VM receives from the procedure table information about the static memory space requirements of the procedure. If sufficient space is available in the static field execution continues, otherwise the VM aborts.

The array field is intended for dynamically created objects of limited lifetime, e.g., for arrays generated by the instruction GENR. When execution returns from a subprogram or a block that captured memory in the array field, this memory is released. Whether or not the released memory is reused depends on the VM implementation.

Unlike memory in the array field, the memory captured in the heap is not released until the end of interpreting.

The string pool stores string constants during VM program loading. The memory allocated to these strings is not released until the end of interpreting.

The value of any word in the static field, the array field, or the string pool may be accessed by its absolute address.

2.4. Call Stack. This stack is used by procedure call instructions, entry instructions into a block or a structured sentence, and loop instructions. It saves and restores the current static space pointer (L) or the stack top pointer (X); the procedure return address (PC) or the exist address from a structured sentence.

The structure of each location in the call stack is determined by the VM-interpreter implementation and should include fields for saving the pointers L or X, PC.

The call stack is basically manipulated in the traditional style. Two additional instructions are provided: an instruction that clears one position from the call stack and an instruction that erases the entire stack.

3. INSTRUCTION SET

The VM instruction set comprises all the basic constructs of static high-level languages. It includes the following instruction groups; word transfer; word group transfers; arithmetic and logic operations on words; real arithmetic; execution control; array manipulation.

The VM instructions are written in a traditional format: one byte is allocated to the instruction code, which is followed by one or several operands. Operand types are determined by the instruction code. An operand may consist of several bytes in succession. The following different instruction operands are possible: a one-byte offset relative to the pointer L or G (this defines an address); a one-, two-, or four-byte immediate operand; one- or two-byte offset relative to PC, which defines a label in the program code; a one-byte value-length specifier; a one-byte specifier of the number of operands in the instruction.

3.1. Examples of Instructions. As the first example, consider the instructions LOOP and OD used in loops with counters. The instruction LOOP has four operands: F is an integer specifying the lower bound of the loop; B is an integer specifying the loop step; T is an integer specifying the upper bound of the loop; and END is a label that provides the exit address from the loop.

The last operand is always read from the VM program. The first three operands may be read from the stack, or may assume default values, so that there is a total of eight variants of the instruction LOOP. By default, F = 1, B = 1.

When LOOP is executed, a three-word loop control field (LCF) is created at the top of the stack. The first field contains the current value of the loop variable. The two other words are not addressable directly from the program: they should contain service information that the instruction OD needs in order to repeat or end the loop, e.g., the values of B and T. Instead of T we may save K — the remaining number of loop repetitions. When LOOP is executed,

$$K := (T-F) \,\%\, B+1 \quad (\text{for } B \neq 0)$$

If $K \leq 0$, the control is immediately transferred to the label END. LOOP also pushes into the stack the current values of the pointers X and PC for the use of the instruction OD. The instruction OD uses the values of X from the return stack to access the LCF and thus determines if the loop is to be repeated or not. If yes, the value of the loop variable is modifed (and if necessary, also the values of the other words in the LCF). Then the control is transferred by the value of PC stored in the return stack. Neither the value of PC nor the value of X are popped from the return stack. When the loop ends, the control is transferred to the label END, which is an operand of the instruction LOOP. The address of the instruction LOOP is determined from the value of PC in the return stack, and PC and X are then popped from the return stack.

Note that, in contrast to high-level languages the instruction OD is not tied to the instruction LOOP. The linkage between the two is established dynamically. It is thus possible to implement the loop

```
for I from F by B to T
do
       if condition
       then branch1
       else branch2
       fi
od
```

by the following VM program:

```
       <compute F>
```

```
        <compute B>
        <compute T>
        LOOP...END
        <condition>
        BRF. ELSE
        <branch 1>
        OD
ELSE:    <branch 2>
        OD
END
```

Here LOOP... is a variant of the instruction LOOP without default values, BRF. is a conditional jump (by the value false at the stack top).

As another example, consider array manipulation instructions. There are 10 such instructions in total.

GENR and HGENR are array generation instructions (GENR generates arrays in the array field, HGENR in the heap). Both instructions have three one-byte operands in the VM program: LEN is an integer specifying the length of an array element, N is an integer defining the array dimension, K is an integer defining the number of generated arrays.

In addition, 2N numbers are popped from the stack, specifying the pairs of bounds in each dimension. After the instruction is executed, K array ticket addresses are pushed into the stack. The array ticket structure is not fixed on the level of VM description. The array ticket should provide access to information which is needed for the execution of other instructions.

The instruction SL uses the array ticket and the array element index to compute the address of the element. A check is made to ensure that the index values are within the range specified by the bounds. There are 14 variants of the instruction SL, which may read the array address from the stack or the static field and the indexes from the stack or (for one-and two-dimensional arrays) from the static field.

The instruction ASSR performs array assignment. The responsibility for ensuring consistency of dimensions and lengths of array elements is with the author of the VM program, while the VM checks that the bounds coincide.

The instructions LWB and UPB compute the array bounds given the ticket address and the dimension. The instruction COPYR creates a copy of the array.

These instructions allow arrays of arbitrary dimensions. Three special instructions are provided for one-dimensional arrays.

The instruction ROW creates a one-dimensional array with the bounds 1:N and fills it with N values popped from the stack; the value N and the length of the array elements are operands of the VM program.

The instruction CMPR compares one-dimensional arrays consisting of one-word elements. The array bounds do not necessarily coincide. When the instruction is executed, two "representatives" of the compared arrays are pushed into thQ stack. These may be the array lengths, if one array is the head of another, or the first nonmatching array elements. The main purpose of the instruction is to perform lexicographic comparison of strings.

The instruction STRING with a one-byte operand in the VM program creates a one-dimensional character array. The value of the operand is the index of the character string in the special string pool. Each procedure and the program proper has its own independent string indexing.

4. QUANTITATIVE ESTIMATES

As we have noted previously, one of the main purposes of using a VM is to reduce the length

of the instruction code. This characteristic is particularly important for microcomputer software. In order to obtain sufficiently reliable relationships, we compared a large package of rational arithmetic programs (47 Algol 68 procedures with a total volume of 242 source lines). Compiling this package into ES, SM4, and VM code, we obtained object code of 4368, 3074, and 836 bytes, respectively, i.e. the VM code is a factor of 3.7 shorter than the SM4 code and a factor of 5.2 shorter than the ES code. Note that these figures were obtained with index checking disabled for ES and SM4 and enabled for the VM. Enabling index checking on the ES would produce a code of 6696 bytes, i.e., eight times as long as the VM code.

LITERATURE

1. **B. Stramm, D. Hughes** *"A virtual machine design for nest free programming"* // INFOR, 24, No. 1, 45-58 (1986).
2. **K. V. Nori** *"The Pascal <p> Compiler Implementation Notes"* // Tech. Report, Institut für Informatik, ETH, Zurich (July 1976).
3. **N. Wirth** *"Personal Computer LILITH"* // Report 40, Swiss Federal Institute of Technology, Zurich (1981).
4. **D. N. Kuznetsov, E. V. Tarasov, A. E. Nedorya** *"The Processor KRONOS as a Transputer Component in the Prototype System MARS"* // Preprint, VTs Sib. Otd. Akad. Nauk SSSR, Novosibirsk (in Russian, 1987).