

Εικονικές Μηχανές, Διερμηνείες και Δυναμική Διαχείριση Μνήμης



Franz Marc, *Rehe im Walde (II)*, 1913-14

Κωστής Σαγώνας <kostis@cs.ntua.gr>

Περιεχόμενα

- Εικονικές μηχανές (Virtual Machines)
- Διερμηνείς (Interpreters)
- Δυναμική διαχείριση μνήμης
 - Στοίβες (Stacks)
 - Σωροί (Heaps)
- Συλλογή σκουπιδιών (Garbage Collection)
 - Μαρκάρισμα και σκούπισμα (mark and sweep)
 - Αντιγραφή (copying)
 - Μέτρηση αναφορών (reference counting)

Εικονικές Μηχανές (Virtual Machines)

- Οι **εικονικές μηχανές** (VMs) αποτελούν ένα ενδιάμεσο στάδιο στη μεταγλώττιση των γλωσσών προγραμματισμού
- Οι VMs είναι **μηχανές** διότι επιτρέπουν τη βήμα προς βήμα εκτέλεση των προγραμμάτων
 - Οι VMs είναι **εικονικές (ή αφηρημένες)** διότι συνήθως
 - δεν υλοποιούνται στο υλικό κάποιου υπολογιστή
 - παραλείπουν πολλές από τις λεπτομέρειες των πραγματικών υπολογιστικών μηχανών (αυτών που υλοποιούνται σε hardware)
 - Οι VMs συνήθως έχουν συστατικά στοιχεία που υλοποιούν λειτουργίες που είναι αναγκαίες για την υλοποίηση συγκεκριμένων (κλάσεων) γλωσσών προγραμματισμού

Εικονικές Μηχανές: Πλεονεκτήματα

- Γεφυρώνουν το χάσμα του υψηλού επιπέδου των γλωσσών προγραμματισμού και του χαμηλού επιπέδου των πραγματικών υπολογιστών μηχανών
- Απαιτούν λιγότερη προσπάθεια για την υλοποίησή τους και για τη μεταγλώττιση των προγραμμάτων
- Ο πειραματισμός και η μετατροπή τους είναι ευκολότερη από ότι η μετατροπή ενός μεταγλωττιστή για τη γλώσσα
 - Σημαντικές ιδιότητες, ειδικά για πρωτοεμφανιζόμενες γλώσσες

Εικονικές Μηχανές: Πλεονεκτήματα

- Προσφέρουν αυξημένη φορητότητα (portability)
 - Οι διερμηνείς VM συνήθως υλοποιούνται σε C
 - Ο VM κώδικας μπορεί να μεταφερθεί μέσω διαδικτύου και να τρέξει στις περισσότερες υπολογιστικές πλατφόρμες
 - Συνήθως, ο VM κώδικας για ένα πρόγραμμα είναι (σημαντικά) μικρότερος σε μέγεθος από τον αντίστοιχο κώδικα μηχανής
- Αρκετές ιδιότητες ασφάλειας του VM κώδικα μπορούν να επαληθευθούν (verified) μηχανικά
- Είναι ευκολότερο να αποδείξουμε τυπικά την ορθότητα του σχεδιασμού και της υλοποίησής των VMs
- Η μέτρηση της επίδοσης (profiling) και η αποσφαλμάτωση (debugging) προγραμμάτων είναι ευκολότερες

Εικονικές Μηχανές: Μειοκτήματα

- Χειρότερη επίδοση (σε χρόνο) των διερμηνέων εικονικών μηχανών σε σχέση με την εκτέλεση εντολών σε γλώσσα μηχανής λόγω του
 - Επιπλέον κόστους διερμηνείας (overhead of interpretation)
 - Ότι είναι σημαντικά πιο δύσκολο να επωφεληθούμε από κάποια χαρακτηριστικά του μοντέρνου hardware των υπολογιστών (π.χ. του hardware-based branch prediction)

Ιστορία των Εικονικών Μηχανών

- Οι πρώτες VMs εμφανίστηκαν στα τέλη της δεκαετίας του 1950
- Οι πρώτες υλοποιήσεις της Lisp (1958) χρησιμοποίησαν VMs με αυτόματη ανακύκλωση μνήμης (garbage collection), sandboxing, reflection, και διαδραστικό φλοιό (interactive shell)
- Στις αρχές της δεκαετίας του 1970, η γλώσσα Forth χρησιμοποίησε μια μικρή και σχετικά εύκολα υλοποιήσιμη VM
- Η γλώσσα Smalltalk (τέλη του 1970) ήταν από τις πρώτες γλώσσες της οποίας η υλοποίηση επέτρεπε την αλλαγή του κώδικα κατά τη διάρκεια εκτέλεσης του προγράμματος (πρώτο πραγματικά διαδραστική υλοποίηση αντικειμενοστρεφούς γλώσσας προγραμματισμού)
- Η υλοποίηση της γλώσσας Self μέσω VM (τέλη του 1980), μιας γλώσσας με πολλά κοινά στοιχεία με τη Smalltalk, κατάφερε να είναι τέτοια ώστε η επίδοσή της να μη διαφέρει πολύ από αυτήν που αντιστοιχεί μεταγλωττιστές της γλώσσας επιτύγχαναν
- Η Java (μέσα του 1990) έκανε τις VMs ευρέως χρησιμοποιούμενες και γνωστές

Επιλογές σχεδιασμού των Εικονικών Μηχανών

- Κάποιες επιλογές σχεδιασμού των VMs είναι παρόμοιες με επιλογές που κάνουμε κατά το σχεδιασμό της μορφής του ενδιάμεσου κώδικα ενός μεταγλωττιστή. Για παράδειγμα
 - Θέλουμε η μηχανή να χρησιμοποιείται σε πολλές διαφορετικές αρχιτεκτονικές και λειτουργικά συστήματα; (όπως π.χ. η JVM)
 - Θέλουμε η μηχανή να χρησιμοποιείται σε πολλές διαφορετικές γλώσσες πηγαίου κώδικα; (όπως π.χ. το CLI/CLR (.NET))
- Κάποιες άλλες είναι παρόμοιες με αυτές του backend ενός compiler:
 - Είναι καλύτερη επίδοση πιο σημαντική από μεγαλύτερη φορητότητα;
 - Είναι η αξιοπιστία πιο σημαντική από την επίδοση;
 - Είναι το (μικρότερο) μέγεθος του κώδικα πιο σημαντικός παράγοντας από την επίδοση σε ταχύτητα της μηχανής;
- Κάποιες άλλες παρόμοιες με αυτές των λειτουργικών συστημάτων:
 - Πως υλοποιείται η διαχείριση μνήμης, ο ταυτοχρονισμός, οι εξαιρέσεις, ...
 - Είναι η μικρή κατανάλωση μνήμης, το scalability, ή η ασφάλεια πιο σημαντικά χαρακτηριστικά από την επίδοση;

Συστατικά των Εικονικών Μηχανών

- Εξαρτώνται από πολλούς παράγοντες:
 - Είναι η γλώσσα (το περιβάλλον της γλώσσας) διαδραστική;
 - Η γλώσσα υποστηρίζει ενδοσκόπηση (reflection) και/ή δυναμικό φόρτωμα κώδικα (dynamic loading);
 - Είναι η επίδοση το παν;
 - Χρειάζεται η γλώσσα να υποστηρίζει ταυτοχρονισμό;
 - Είναι απαραίτητο το sandboxing;

Υλοποίηση των Εικονικών Μηχανών

- Οι εικονικές μηχανές συνήθως γράφονται σε “φορητές” γλώσσες προγραμματισμού όπως η C ή η C++
- Για μέρη που είναι σημαντικά για την επίδοση της εικονικής μηχανής, συνήθως χρησιμοποιούμε assembly
- Εικονικές μηχανές για κάποιες γλώσσες (π.χ. Lisp, Forth, Smalltalk) γράφονται με χρήση της ίδιας τη γλώσσας
- Πολλοί διερμηνείς για VMs γράφονται σε GNU C, για λόγους που θα γίνουν προφανείς στις επόμενες διαφάνειες

Μορφές Διερμηνέων

- Αρκετές υλοποιήσεις γλωσσών προγραμματισμού χρησιμοποιούν διερμηνείς δύο ειδών:
 - **Command-line interpreter**
 - Διαβάζει και αναλύει συντακτικά κομμάτια πηγαίου κώδικα της γλώσσας τα οποία και εκτελεί
 - Χρησιμοποιείται σε συστήματα που αλληλεπιδρούν με το χρήστη
 - **Virtual machine instruction interpreter**
 - Διαβάζει και εκτελεί εντολές μιας ενδιάμεσης μορφής εκτελέσιμου κώδικα όπως bytecode εντολών μιας εικονικής μηχανής

Υλοποίηση των Διερμηνέων

Υπάρχουν πολλοί τρόποι υλοποίησης διερμηνέων:

1. Direct string interpretation

Source level interpreters are very slow because they spend much of their time in doing lexical analysis

2. Compilation into a (typically abstract syntax) tree and interpretation of that tree

Such interpreters avoid lexical analysis costs, but they still have to do much list scanning (e.g. when implementing a 'goto' or 'call')

3. Compilation into a virtual machine and interpretation of the VM code

Διερμηνείς Εντολών Εικονικών Μηχανών

- By compiling the program to the instruction set of a virtual machine and adding a table that maps names and labels to addresses in this program, some of the interpretation overhead can be reduced
- For convenience, most VM instruction sets use integral numbers of bytes to represent everything
 - opcodes, register numbers, stack slot numbers, indices into the function or constant table, etc.



Example: The **GET_CONST2** instruction

Μέρη της Υλοποίησης μιας Εικονικής Μηχανής

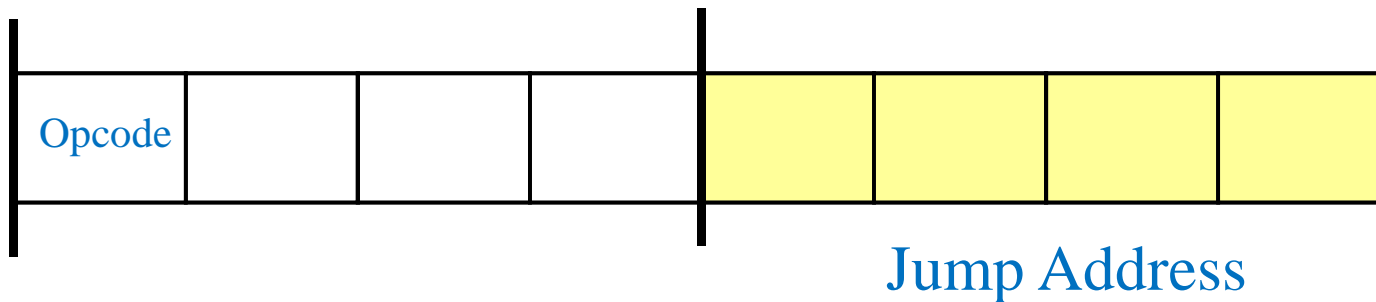
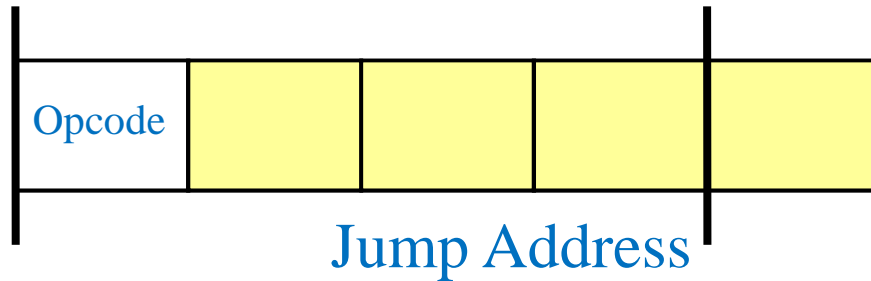
- Το μέρος που αποθηκεύεται το πρόγραμμα (ο κώδικας)
 - Το πρόγραμμα είναι μια ακολουθία από εντολές
 - Φορτωτής (loader)
- Μέρη που καταγράφουν την κατάσταση (της εκτέλεσης)
 - Στοίβα (stack)
 - Σωρός (heap)
 - Καταχωρητές (registers)
 - Ένας ειδικός καταχωρητής, ο **μετρητής προγράμματος (program counter)**, δείχνει πάντα στην επόμενη προς εκτέλεση εντολή
- Το σύστημα χρόνου εκτέλεσης (runtime system)
 - Σύστημα παραχώρησης μνήμης (memory allocator)
 - Συλλέκτης σκουπιδιών (garbage collector)

Η βασική δομή ενός Bytecode Interpreter

```
byte *pc = &byte_program[0];
while(TRUE) {
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            source_reg_num = pc[1];
            const_num_to_match = get_2_bytes(&pc[2]);
            ... // get_const2 code
            pc += 4;
            break;
        ...
        case JUMP:
            jump_addr = get_4_bytes(&pc[1]);
            pc = &byte_program[jump_addr];
            break;
        ...
    }
}
```

To align or to not align VM instructions?

Σε μια 32-bit μηχανή



NOTE: Padding of instructions can be done by the loader.
The size of the bytecode files need not be affected.

Bytecode Interpreter with Aligned Instructions

```
byte *pc = &byte_program[0];
while(TRUE) {
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            source_reg_num = pc[1];
            const_num_to_match = get_2_bytes(&pc[2]);
            ... // get_const2 code
            pc += 4;
            break;
        ...
        case JUMP: // aligned version
            jump_addr = get_4_bytes(&pc[4]);
            pc = &byte_program[jump_addr];
            break;
        ...
    }
}
```

Διερμηνέας με αφηρημένη κωδικοποίηση εντολών

```
byte *pc = &byte_program[0];
while(TRUE) {
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            source_reg_num = pc[GET_CONST2_ARG1];
            const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
            ... // get_const2 code
            pc += GET_CONST2_SIZEOF;
            break;
        ...
        case JUMP: // aligned version
            jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
            pc = &byte_program[jump_addr];
            break;
        ...
    }
}
```

```
#define GET_CONST2_SIZEOF 4
#define JUMP_SIZEOF 8
#define GET_CONST2_ARG1 1
#define GET_CONST2_ARG2 2
#define JUMP_ARG1 4
```

Διερμηνέας με αφηρημένο έλεγχο

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            source_reg_num = pc[GET_CONST2_ARG1];
            const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
            ... // get_const2 code
            pc += GET_CONST2_SIZEOF;
            NEXT_INSTRUCTION;
        ...
        case JUMP: // aligned version
            jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
            pc = &byte_program[jump_addr];
            NEXT_INSTRUCTION;
        ...
    }
}
```

```
#define NEXT_INSTRUCTION \  
goto next_instruction
```

Έμμεσα Νηματικοί Διερμηνείς

- Σε έναν έμμεσα νηματικό διερμηνέα (indirectly threaded interpreter) we do not switch on the opcode encoding; instead we use the bytecodes as indices into a table containing the addresses of the VM instructions
- The term *threaded code* refers to a code representation where every instruction is implicitly a function call to the next instruction
- A threaded interpreter can be very efficiently implemented in assembly
- In GNU CC, we can use the labels as values C language extension and take the address of a label with `&&labelname`
- We can actually write the interpreter in such a way that it uses indirectly threaded code if compiled with GNU CC and a switch for compatibility

Δομή ενός Έμμεσα Νηματικού Διερμηνέα

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            get_const2_label:
                source_reg_num = pc[GET_CONST2_ARG1];
                const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
                ... // get_const2 code
                pc += GET_CONST2_SIZEOF;
                NEXT_INSTRUCTION;
            ...
        case JUMP: // aligned version
            jump_label:
                jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
                pc = &byte_program[jump_addr];
                NEXT_INSTRUCTION;
            ...
    }
}
```

```
static void *label_tab[] = {
    &get_const2_label,
    ...
    &jump_label,
    ...
}
#define NEXT_INSTRUCTION \
    goto *(void *) (label_tab[*pc])
```

Άμεσα Νηματικοί Διερμηνείς

- In a directly threaded interpreter, we do not use the bytecode instruction encoding at all during runtime
- Instead, the loader replaces each bytecode instruction encoding (opcode) with the address of the implementation of the instruction
- This means that we need one word for the opcode, which increases the VM code size

Δομή ενός Άμεσα Νηματικού Διερμηνέα

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            get_const2_label:
                source_reg_num = pc[GET_CONST2_ARG1];
                const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
                ... // get_const2 code
                pc += GET_CONST2_SIZEOF;
                NEXT_INSTRUCTION;
            ...
        case JUMP: // aligned version
            jump_label:
                pc = get_4_bytes(&pc[JUMP_ARG1]);
                NEXT_INSTRUCTION;
            ...
    }
}
```

```
static void *label_tab[] = {
    ...&get_const2_label,
    ...&jump_label,
    ...
}
#define NEXT_INSTRUCTION \
    goto **(void **) (pc)
```

```
#define GET_CONST2_SIZEOF 8
#define JUMP_SIZEOF 8
#define GET_CONST2_ARG1 5
#define GET_CONST2_ARG2 6
#define JUMP_ARG1 4
```

Νηματικός Διερμηνέας με Χρήση Prefetching

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            get_const2_label:
                source_reg_num = pc[GET_CONST2_ARG1];
                const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
                pc += GET_CONST2_SIZEOF; // prefetching
                ... // get_const2 code
                NEXT_INSTRUCTION;
            ...
        case JUMP: // aligned version
            jump_label:
                pc = get_4_bytes(&pc[JUMP_ARG1]);
                NEXT_INSTRUCTION;
            ...
    }
}
```


Subroutine Threaded Interpreter

- The only portable way to implement a threaded interpreter in C is to use **subroutine threaded code**
- Each VM instruction is implemented as a function and at the end of each instruction the next function is called

Stack-based vs. Register-based VMs

- A VM can either be *stack-based* or *register-based*
 - In a stack-based VM most operands are (passed) on the stack
 - The stack can grow as needed
 - In a register-based VM most operands are passed in (virtual) registers
 - These registers are often implemented using an array rather than physical machine registers
 - The number of registers is limited
- Most VMs are stack-based
 - Stack machines are simpler to implement
 - Stack machines are easier to compile to
 - Less encoding/decoding to find the right register
 - Unless virtual registers are mapped to physical registers, virtual registers are not faster than stack slots

Virtual Machine Interpreter Tuning

Common VM interpreter optimizations include:

- Writing the interpreter loop and key instructions in assembly
- Keeping important VM registers (pc, stack top, heap top) in hardware registers
 - GNU C allows global register variables
- Top of stack caching
- Splitting the most used set of instruction into a separate interpreter loop

Instruction Merging and Specialization

Instruction Merging: A sequence of VM instructions is replaced by a single (mega-)instruction

- Reduces interpretation overhead
- Code locality is enhanced
- Results in more compact bytecode
- C compiler has bigger basic blocks to perform optimizations on

Instruction Specialization: A special case of a VM instruction is created, typically one where some arguments have a known value which is hard-coded

- Eliminates the cost of argument decoding
- Results in more compact bytecode representation
- Reduces the register pressure from some basic blocks