Science and Technology of Software M1

Advanced Compilation 4I504

Implementation of the ICFP Virtual Machine

Author :
M. Sandu POSTARU

Supervisors :
M. Emmanuel CHAILLOUX
M. Clément PONCELET

Version 1.0 of
18.02.2017

Contents

Chapter 1

The Universal Machine

The Universal Machine (hereafter referred to as "UM") is a Virtual Machine presented at the Ninth annual ICFP Programming Contest. The UM is capable of running the UMIX operating system, which was written for the purpose of the competition.

We will start by presenting a succinct description of the UM, motivate our implementation language choice, explain some of the delicate points concerning our implementation and finish with a full benchmark of our UM.

1.1 Specification

The Universal Machine uses unsigned 32-bit words and has the following components :

- A global memory composed of a collection of arrays of words, each referenced by a distinct unsigned 32-bit identifier. The array having the identifier 0 stores the main program.
- Eight distinct general-purpose registers, capable of holding one word each.
- A program counter that points to the current instruction.
- A 1x1 output console capable of displaying symbols from the ASCII standard.

Using these components the UM is capable of executing each of its 14 supported operators. Each instruction is coded by an unsigned 32-bit word. We use bit manipulation to extract the operator code and the identifier of each register used for the current instruction. Contrary to popular architectures that use little-endian to interpret words in memory, the UM uses a big-endian format for its word manipulation.

1.2 Runtime

In order to start the UM, the name of a file containing the source code of the current program must be provided. The UM then proceeds by importing the source code into memory, converting from big-endian to little-endian each instruction and storing them in the 0 array. Once this operation is successful the main interpretation loop begins:

1. The current instruction is read.
2. The value of the operator is extracted.
3. The value of each register (A, B, C or I) is extracted.
4. The current instruction is executed.
5. The program counter is increased.

The main loop continues until stopped (using the HALT instruction) or until an error occurs.

Chapter 2

Implementation

2.1 Language choice

One important feature of each Virtual Machine is performance, the UM is no different. In our approach we tried to identify the optimal programming language that allows a good balance between performance and data abstraction. Several languages were considered but step by step were discarded until one emerged victorious. The victor was the C language. Below we expose the motivations of our choice.

2.1.1 C

The C language needs no introduction, it is a very powerful and widely used language. It forms the core of most modern languages and allows us to control the very low level aspects of the computer. It allows fine grained tuning when it comes to primitive data types (`uint32_t` `uint64_t`), hardware performance and explicit memory management. There is no better choice when performance is needed, for these reasons we chose to implement the Universal Machine in C.

2.1.2 OCaml

OCaml offers good performance and allows us to abstract the data structure layer, but it has one major drawback for this project. In OCaml the primitive int data type is represented on 31-bits, one bit being used by the Garbage Collector to distinguish between integers and pointers. One option would be to use the `Int32` module, but then the values would be boxed, occupying more memory space, and arithmetic operations would be generally slower than those on `int`.

2.1.3 Java

We also considered Java for its powerful data types library but soon realised that it doesn't have unsigned integers neither. In Java SE 8 and later, we can use the `int` data type to represent an unsigned 32-bit integer, but we must use special methods (`compareUnsigned` `divideUnsigned`) from the Integer class in order to treat the signed int as unsigned int. This induces slower arithmetic operations and a unnecessary verbosity in our code.

2.2 Implementation

2.2.1 Structure

Implementing the UM specification in C required some design choices:

- The global memory is represented by an array containing array pointers (the program array and dynamically created arrays).
- The general-purpose registers are represented by an register array.

The full UM structure is represented by the `vm_t` type:

```
typedef struct _vm {
    uint32_t **arrays;              /* global memory */
    uint64_t *arrays_size;          /* number of arrays in global memory  */
    uint32_t *registers;            /* registers */
    uint32_t *program;              /* program array */
    uint32_t *pc;                   /* program counter */
} vm_t;
```

2.2.2 Initialisation

We start by reading the code into memory, converting each instruction from big-endian to little-endian and storing them into a preallocated `uint32_t` array. Once this operation is successful, the UM is initialised using the function `void init_vm(vm_t *vm, uint32_t *program)`. This function allocates memory for all the UM componenets and sets the allocated memory to zero. Its dual `void free_vm(vm_t *vm)` frees all the resources allocated by the UM. After the UM has been successfully initialised, the main loop begins.

2.2.3 Interpretation loop

The interpretation loop has the following form :

```
while(true){

    // the current instruction is read
    uint32_t instr = arrays[0][*pc];

    // the value of each register is extracted
    uint8_t c = (instr >> 0) & 0b111;
    uint8_t b = (instr >> 3) & 0b111;
    uint8_t a = (instr >> 6) & 0b111;

    // the value of the operator is extracted
    uint8_t opcode = (instr >> 28) & 0b1111;

    switch(opcode){

        case MOVEIF :
            if(DEBUG){
                fprintf(stderr,"MOVEIF: ...);
            }
            if(r[c]){
                r[a] = r[b];
            }
            break;
            .
            .
            .
        }

        // the program counter is increased
        *pc = *pc + 1;
    }
```

All the necessary information is extracted from the instruction. Based on the operator code, one specific branch will be executed and then the program counter is increased. The interpretation loop continues until stopped (by the HALT instruction) or until an error arises.

2.2.4 Memory management

The global memory model is composed of an array of arrays of unsigned 32-bit integers (`uint32_t **arrays`). A new slot can be allocated using the `ALLOC` operator and can be freed by the `FREE` operator. When initialised, the UM has `INITIAL_ARRAY_SIZE` available slots.

Since performance is the key element of the UM, this array based approach allows us to have a fast `O(1)` access to any part of the memory. The drawback of this approach is that `C` arrays have a static size and are non-extensible. In the case of memory saturation a `realloc` call is made and the current size is doubled similar to the Java's `ArrayList<T>` module. The call can potentially recopy `uint32_t **arrays` in another memory segment and can therefore be quite costly. Nevertheless, performance-wise this approach is superior to a linked-list approach, which has the advantage of not needing a reallocation but can have a much slower `O(CURRENT_ARRAY_SIZE)` access.

When an existing array is freed, the unique identifier associated with the array becomes available for future use. A free list (implemented as a linked-list) could have been used to store the free indexes, but this approach would have slowed the execution down, since most `ALLOC` calls are shortly followerd by a `FREE` call, so new cells for the free-list would need to be allocated/freed very often during runtime.

Instead we used a different approach. The last allocated index `last_i` is stored, when a new `ALLOC` takes place, the next index `i = last_i + 1` is being used. This process takes place in circular manner, meaning that if `i == INITIAL_ARRAY_SIZE` then `last_i = 0`, so the process starts from the beginning. When a full tour has been made (`i == last_i`) a `realloc` call takes place.

No compaction method for the memory can be used, since there is no way to modify all the array index references during runtime. No distinction can be made between an array containing another array index or just an integer value that resulted from another operation, like an addition for example.

Chapter 3

Benchmarks

The UM has been successfully tested with both `sandmark.uzm` (benchmark for the Universal Machine) and `codex.umz` (a mini operating system) which uses the following decryption key:

$$(\backslash b.bb)(\backslash v.vv)06FHPVboundvarHRAk$$

All the benchmarks have been effectuated on machines from Pierre and Marie Curie University.

3.1 Sandmark

The stress test provided on the ICFP website has been used to test the performance of our implementation. Using `gcc 6.3.1` with `-O3` optimisations we obtain `31.192s`. This result is pretty good, considering the fact that 100+ costly operations are being made, in order to prove the well-functioning of the machine.

3.2 Codex

The mini operating system prompts for a decryption key, once entered it starts quite fast totaling at `12.876s`. This program performs some initials tests from the sandmark package and a decryption routine so the result is again coherent.

Chapter 4

Conclusion

Implementing the Universal Machine requires a full understanding of its internal behaviour. Comprehending the intricacies of the system is required in order to guarantee good performances.

While it's true that most high level languages such as Perl, Ruby, Python offer a pleasant abstraction level, they are quite slow for implementing the Universal Machine. A language like C allows us to control the very low level aspects of the computer and is perfect for such a task. Comparing the pros and the cons of each considered language was a good exercise, since it proved us that there is a right tool for each task.

Even if the Universal Machine memory model is quite simple, it allowed us to get a better understanding of the difficulty associated with efficient memory management and appreciate more the complexity of modern garbage collection mechanisms.