Advanced compilation course

Michael Desboeufs
Samuel Mercier

Abstract :

The present report describes the implementations of a copying garbage collector and the super instructions in our minivm (a virtual machine for minisc, a subset of the scheme language). Both subjects are treated separately. First, a little description is given. Then, the implementation and its problems are described.

Precise garbage Collector (phase 1) :

This project is composed of two parts. To adapt the code to make it "precise" and to implement a garbage collector who use copying technique. To render the code precise, we use the tagging, which consists in differentiating pointer from other elements at running time.

GC precise (tagging)

In a first phase, in the Loader, when I parse the code to put it in memory, I test the data in the case of a LINT, LOAD, STOR. If I find in the last argument an integer, I will tag it by left shifting it of 1, and if it is a label I shift it to left of 1 and add 1.

The tagging choice was done as this (the + 1 for pointers), because it avoids to correct most of the operations later. The assumption that the last (most significant bit) of a pointer is always 0 has been made.

Now in the interpreter loop, I test at beginning the PC (R[31]) to see if it is not an integer and if it is a correct pointer. In different cases of the switch, I test if the values are pointers or integers and returns an error in some cases where it should be an integer instead of pointer (or the opposite). I reinforced the tests on the pointer with the help of our tagging.

The main macros like TAG, TAG_INT, IS_PTR are defined in vm.h.

The first idea was to do it in scala, but it was not a good way to do it. Since the addresses are defined only at the time of the loading, the best way I think was to do it at that moment. The second difficulty was to test the code to be sure I had correctly done the tagging.

Copying GC

At first, it seemed simple to code the GC copying, but the management of the pointers when copying from "from space" to "to space", was more complicated than we had anticipated. Therefore, for the algorithm, we used the Cheney's algorithm.

Therefore, I scan from the registers, the pointers who point on the heap. Then I move the corresponding blocks to the "to space" and the pointers from the registers are updated to the new blocks. Then all the pointers to the heap are tested in each block and then they are updated and the new blocks found are moved too.

In case, not enough space is freed then the program exits.

It was hard to test my algorithm with the current mode interactive, so I modified it to have a better view of it.

Conclusion

The garbage collector works greatly. In the example of "hello", we need about 500 bytes of head with it and about 1600 bytes without it. As expected, the mark and sweep gc requires less heap than the copying one.

Super instructions :

The core of a virtual machine can be seen as a c switch statement nested in a for loop :

```
for(; ; )
        switch(opcode)                              1, 4
        {
        case OP_ADD:
                // code for add                     5
                continue;                           6
        case OP_SUB:
                // code for sub                     2
                continue;                           3
        ...
        }
```

normal mode, main loop and control flow of a SUB instruction followed by an AdD instruction.

In such machines if the instructions are simple the overhead caused by the jumps of the switch and continue statements can dramatically slow down the execution. This overhead can be reduced using threaded mode :

```
OP_ADD:
        // code for add                             3
        goto *pc++;                                 4
OP_SUB:
        // code for sub                             1
        goto *pc++;                                 2
...
```

threaded mode, main loop and control flow of a SUB instruction followed by an ADD instruction.

In such mode the opcode is replaced by a pointer to the code of the instruction and the continue statement by a jump to the next opcode. Note that the compiler used to compile the vm must be able to generate "labels as pointers".
Now if an instruction (let say `OP_SUB`) is always followed an other instruction (let say `OP_ADD`) then the native code of both instructions can be coalesced, thus removing the jump between them, and the opcode of the `OP_SUB` instruction (in fact, pointer to the `OP_SUB` instruction) can be replaced by a pointer to the newly created super-instruction `OP_SUBADD`. This process is known as code inlining :

```
OP_ADDSUB:
        // code for sub
        // code for add
        goto *pc++
```

super instruction, code inlining

The idea is now to identify all basic blocks and replace each of them by a single super-instruction.

Implementation :

the first thing to do is to generate the inlinable code for the instructions. This has been implemented in the `init_vm()` function of the vm. A label has been inserted at the beginning and the end of each instructions :

```
INS_UNKN:
        pprintf("ins_unkn : pc=%i", R[31]);
        pexit_vm();
INS_UNKN_END:
INS_LINT:
        pdecode_args(R[31], OP_LINT, &a, &b, &c);
        R[31]+=4;
        R[a]=*(unsigned int *)R[31];
        R[31]+=4;
INS_LINT_END:
        goto **((void**)R[31])++;
INS_ADD:
        pdecode_args(R[31], OP_ADD, &a, &b, &c);
        R[31]+=4;
    R[a] = R[b] + R[c];
INS_ADD_END:
        goto **((void**)R[31])++;
```

excerpt of the init_vm() function.

Note that :

1. the same code is also used for the threaded mode and that the `goto` statement is not part of the instruction.
2. The temporary variables a, b and c used to store the operands have been made static, because the instructions are called from outside the `init_vm()` function and its frame is not active at this time.
3. Function calls are made through pointers, because calls are implemented using a relative offset on some architecture.

Labels must then be stored in a table :

```
/* creates the table for threaded mode. */
labels_[0]=&&INS_UNKN;
labels_[OP_LINT]=&&INS_LINT;
labels_[OP_ADD]=&&INS_ADD;
...

/* creates the table for super instructions. */
labels_end[0]=&&INS_UNKN_END;
labels_end[OP_LINT]=&&INS_LINT_END;
labels_end[OP_ADD]=&&INS_ADD_END;
...
jump=&&INS_JUMP;
jump_end=&&INS_JUMP_END;
```

creation of the table of labels

Instructions can now be copied using a `memcpy()`, the length being the difference between the end and the beginning of the instruction :

```
memcpy(buffer, labels_[opcode], labels_end[opcode]-labels_[opcode]);
```

The next step is to inline basic blocks. But first we need to identify them. This can be easily done as split points are located at labels and jumps and of course, at the beginning and the end of the program. Note that a basic block needs at least one instruction. Therefore, no basic block is created between two labels if there is no instruction between them.

```
                                           split point, beginning of a basic block
        // initialization code.
                                           split point, end of a basic block.
Loop:
                                           split point, beginning of a basic block.
        ...
        // conditional jump to end.
                                           split point, end of a basic block.
                                           split point, beginning of a basic block.
        // jump to loop.
                                           split point, end of a basic block.
End:
                                           split point, beginning of a basic block.
        // termination code.
                                           split point, end of a basic block.
```

Identification of the basic blocks

in our vm jumps are implemented using the conditional move `CMOV` instruction modifying the register `R31` (in fact any instructions using `R31` as destination operand can be seen as a jump, but the minisc compiler does not use this feature).

Now, two dependent steps must be done : generating the code executed by the vm and the one executed by the computer. The `parser/align_loader` have been modified for that. The idea is to write the address of the block allocated for super-instruction followed by the opcodes of all instructions in the basic block as code for the vm, and to copy the native code of each instruction in the block of the super-instruction. Of course, a jump to the next instruction of the vm must be appended at the end of each super-instruction.

However, this technique does not work and is not practical for some reasons :

1. the size of a basic block is not known as we parse the input file from top to bottom. Therefore, writing directly the address of the block as opcode is not a good idea because a call to `realloc()` can move the block.
2. Blocks allocated by `malloc()/realloc()` are not executable.
3. Allocation of a single block using `mmap()` works fine, but more causes the vm to crash.

Therefore another technique has been used.

1. A single block is allocated for all the super instructions : it is dynamically reallocated whenever it becomes too small. The opcode is now the offset in this block (the block can move but the offset will always be the same) and the address of all opcodes are stored in a table in order to be patched.
2. Once the program has been parsed a single call to `mmap()` to allocate an executable block is issued and the content of the block is copied.
3. All the opcode are patched : the address of the block returned by `mmap()` is simply added.

This concludes our implementation.

Discussion

- The implementation has been tested with some programs and seems to work fine on ICBC07PCxx (it has not crashed). More tests should be done in order to be sure it is really stable. The program crashed on IN3SUNxx. It seems that the block returned by the `mmap()` function is not executable. A newer version has been tested but it does not compile anymore because some variables are defined somewhere else in the libraries. We did not have time to investigate more.
- No benchmark has been run to check the execution speed. However minisc issues a lot of `ALOC` instructions to create the activation frame for the functions. In our opinion, if the heap becomes depleted the gain of speed is overwhelmed by the time spent in the garbage collector. The compiler should be rewritten in order to use a stack to store the activation frames of the functions.