Binary Translation — 1

G. Lettieri

22 Oct. 2014

1 Introduction

The idea of *binary translation* is to first translate the guest code into the equivalent host code for the virtual machine, and then jump at the translated code. If the translated code is kept in a cache and reused whenever the the guest is trying to execute it again, the cost of decoding the guest instructions is thus amortized. Moreover, the translated code can be optimized during the translation, since our emulator now looks at more than a single guest instruction at a time. This strategy generally brings great speedups w.r.t. the simpler emulation we have already seen, where each guest instruction is fetched, decoded and emulated in isolation, and this is done every time the guest tries to execute it. Apart from this, our emulator is again a normal, unprivileged program running on the host system, relying on the host operating system for the management of its resources. What we are doing is simply to replace the CPU loop with a more sophisticated one. In particular, the considerations about I/O, virtual memory and multi-threading are essentially the same as before.

Typically, the translation of guest code is not performed all at once, but in smaller units called *dynamic basic blocks*, or *translation blocks* (TB). A dynamic basic block starts with an instruction which is the target of a jump (including the jump to the entry point of the program) and includes all the instructions that follow, stopping immediately after the first branch or jump instruction. One reason for using dynamic basic blocks is that it is otherwise very difficult to identify all the code in the guest memory, since code looks just like data. Dynamic basic blocks start at instructions that the emulated CPU is actually trying to fetch after a jump, and therefore we rest assured that the corresponding bytes in the emulated memory must be interpreted as an instruction. Moreover, as long as the fetched instruction is not a jump (and we don't need to execute the instruction to know this), we are sure that it is followed by another instruction, and so on, until we found a branch or a jump. At unconditional jumps we stop, because we don't know whether the bytes that follow them are for code or for data (the emulated CPU is not going to execute them, for what we now). At branches (conditional jumps) we also stop, because it is possible that one of the two branches may never be taken, and we don't know if this is actually the case. The bytes that live at a dead branch may not be code at all. Dynamic basic

blocks allow us to only translate code that the guest CPU is actually going to execute.

A TB is identified by the guest address of the its first instruction so that, after the execution of a TB, we can use the current value of the guest instruction pointer to find the next TB to execute.

The CPU loop becomes something like

```
for (;;) {
        tb = find_in_cache(CPU->ip);
        if (!tb) {
                tb = translate(CPU->ip);
                add_to_cache(tb);
        }
        exec(tb, env);
}
```

Where CPU is the usual descriptor for the emulated CPU, env is some data structure containing all the information on the state of the guest (including the CPU, but also memory), and tb is a pointer to a translation block descriptor.

2 Examples

2.1 Executing code generated at run-time

To implement the translate () and exec() functions we need two solve two technical problems on the host system:

1. How to find/allocate an area of memory where we can write new executable code;
2. how to jump to the code.

For problem 1, since code is just a sequence of bytes, we might think that a simple array of chars is sufficient. However, the host operating system will generally try to segregate code from data, for security reasons. Therefore, a simple array allocated with the usual means (either statically, on stack, or with **new** or malloc()) will not always work: the corresponding memory area might be protected from execution. On Unix-like systems we can use the mmap() system call as follows:

```
/* get some executable memory */
void *exec_area = mmap(
    NULL,                                   // preferred address: let the
                                            // kernel choose
    4096,                                   // requested size (a page)
    PROT_WRITE|PROT_READ|PROT_EXEC, // protection bits: rwx
    MAP_PRIVATE|MAP_ANON,                   // not backed by file , do not
                                            // share with other processes
    -1, 0                                   // file descriptor and offset:
```

```
                                                    // unused with MAP_ANON
);
if (exec_area == MAP_FAILED) {
        ...
}
```

If the call succeeds, exec_area points to an area of memory that can be used for executable code, because of the PROT_EXEC flag.

Assume, now, that we have stored some machine code in the memory pointed to by exec_area. To actually execute the code, we have at least two options:

1. Cast the pointer obtained from mmap() to a function pointer, then call the function;
2. use an intermediate function written in assembler.

Now let us consider an example. Assume we want to execute the machine code corresponding to the following C++ function:

```
int sum(int a, int b)
{
    return a + b;
}
```

We can obtain the corresponding machine code by compiling the above source and using a disassembler on the executable file (e.g., "`objdump --disassemble`"). In this example, the machine code is the following (stored for convenience in a byte array):

```
char code[] = {
  0x55,                 // pushl %ebp
  0x89, 0xe5,           // movl %esp, %ebp
  0x8b, 0x45, 0x0c,     // movl 12(%ebp), %eax
  0x8b, 0x55, 0x08,     // movl 8(%ebp), %edx
  0x01, 0xd0,           // addl %edx, %eax
  0xc9,                 // leave
  0xc3                  // ret
};
```

As long as we don't try to execute it, the code is just data, so we can simply copy it in the executable area:

```
copy(code, code + sizeof(code), static_cast<char*>(exec_area));
```

(here we are using STL's copy() function, but there is nothing special about it: we are just copying bytes from the code array to the memory pointed to by exec_area).

Now we have to actually execute the machine code. As we said earlier, an option is to cast exec_area into a function pointer. In this case, it must be a

pointer to a function that takes two integers and returns an integer. We can program this as follows:

```
/* sum_t is a pointer to a function that takes two
 * integers and returns another integer
 */
typedef int (*sum_t)(int, int);
...
/* convert the start address to a function pointer */
sum_t sum = reinterpret_cast<sum_t>(exec_area);

/* call the function */
int r = sum(2, 3);
```

For the second option, we can define an intermediate exec() function in assembler. For this example, we need to pass three parameters to this function: the address of the entry point and the two integers to sum. We can program this function as follows:

```
exec:
  /* standard C++ prologue */
  pushl %ebp
  movl %esp, %ebp
  pushl %ebx
  /* call the sum function */
  pushl 12(%ebp)          // first integer
  pushl 16(%ebp)          // second integer
  movl 8(%ebp), %ebx      // entry point
  call *%ebx              // indirect jump through register
  addl $8, %esp
  popl %ebx
  /* standard C++ epilogue */
  leave
  ret
```

We can call the above function from our C++ source as follows:

```
extern "C" int exec(void *entry_point, int a, int b);
...
int r = exec(exec_area, 2, 3);
```

2.2 Translating code

Our emulator has to tranlate the guest code into the corresponding host code and then execute the latter. In the previous example we have omitted this translation step. Now, assume that the code from the sum() function is actually

guest code. The registers mentioned in the code are not the real registers in the host CPU, but the registers in the *target* machine. These registers are represented by some data structure in the host. This is also true for all accesses to memory: when the guest code pushes something on the stack, this is the *target* stack, which is implemented as some data structure in our emulator.

For example, we can define the following data structures to implement the CPU and the memory of the target system (assume the target system has no MMU):

```
// guest CPU
struct CPU_des {
        uint32_t        EAX;
        uint32_t        ECX;
        uint32_t        EDX;
        uint32_t        EBX;
        uint32_t        ESP;
        uint32_t        EBP;
        uint32_t        ESI;
        uint32_t        EDI;
        uint32_t        EIP;
        uint32_t        EFLAGS;
} CPU;

// guest memory
const int MEM_SIZE = 65536; // 64 KiB
uint8_t  Mem[MEM_SIZE];
```

For convenience, we also define an env_dev structure to tie CPU and memory together:

```
// the guest environment, containing pointers to the
// guest CPU and guest memory
struct env_des {
        CPU_des *CPU;
        uint8_t  *MEM;
};
```

Now assume our guest wants to execute the sum() function of the previous example. Our emulator will have to translate that code into something like the following:

```
.set  EAX, 0
.set  ECX, 4
.set  EDX, 8
.set  EBX, 12
.set  ESP, 16
.set  EBP, 20
```

```
.set ESI, 24
.set EDI, 28
.set EIP, 32
.set EFLAGS, 36

// we assume
// %esi -> address of emulated CPU
// %edi -> address of emulated memory

tr_code:
// PROLOGUE
        pushl %eax
        pushl %ecx
// PUSHL %EBP
        subl $4, ESP(%esi)
        movl EBP(%esi), %eax
        movl ESP(%esi), %ecx
        movl %eax, (%edi, %ecx)
// MOVL %ESP, %EBP
        movl ESP(%esi), %eax
        movl %eax, EBP(%esi)
// MOVL 12(%EBP), %EAX
        movl EBP(%esi), %ecx
        addl $12, %ecx
        movl (%edi, %ecx), %eax
        movl %eax, EAX(%esi)
// MOVL 8(%EBP), %EDX
        movl EBP(%esi), %ecx
        addl $8, %ecx
        movl (%edi, %ecx), %eax
        movl %eax, EDX(%esi)
// ADDL %EDX, %EAX
        movl EDX(%esi), %eax
        addl %eax, EAX(%esi)
        pushf
        popl EFLAGS(%esi)
// LEAVE
        movl EBP(%esi), %eax
        movl %eax, ESP(%esi)
        movl (%esi, %eax), %eax
        movl %eax, EBP(%esi)
        addl $4, ESP(%esi)
// RET
        movl ESP(%esi), %ecx
        movl (%edi, %ecx), %eax
        movl %eax, EIP(%esi)
```

```
        addl $4, ESP(%esi)
// EPILOGUE
        popl %ecx
        popl %eax
        ret
```

Note that the emulator will actually have to create the host *machine code* that corresponds to the assembler above.

In the above code we assume that two registers already point to the guest CPU and memory data structures. We can easily setup these registers if we jump to the translated code using an intermediate function, that we write in assembler. For example, we can use the following function:

```
.set CPU, 0
.set MEM, 4

.globl exec
exec:
        pushl %ebp
        movl %esp, %ebp
        pushl %ebx
        pushl %esi
        pushl %edi
        movl 12(%ebp), %ebx
        movl CPU(%ebx), %esi
        movl MEM(%ebx), %edi
        movl 8(%ebp), %ebx
        call *%ebx
        popl %edi
        popl %esi
        popl %ebx
        leave
        ret
```

We can call the above function from our C++ source as follows:

```
extern "C" void exec(void *entry_point, env_des *e);
 ...
env_des env = { &CPU, Mem };
exec(exec_area, &env);
```

The complete example can be found at
`http://lettieri.iet.unipi.it/virtualization/dynex.tar.gz`.

2.3 Dynamic vs Static Basic Blocks

Dynamic basic blocks differ from (static) basic blocks used by compilers (typically in their optimization phase). A basic block is defined as a sequence of instructions with only one entry point and only one exit point. An entry point is the target of a jump anywhere in the program: compilers have the complete list of these jumps, since they have created them in the first place. Instead, our emulator is not aware of instructions that it has not yet seen, and that may jump in the middle of a previously identified dynamic basic block. The result is that dynamic basic blocks are typically larger than static basic blocks; moreover, it may happen that the same instruction belongs to several dynamic basic blocks. All of this can be shown with a simple example:

```
1        movl $0, %ecx
2  loop: cmpl $8, %ecx
3        jge end
4        incl %eax
5        jmp loop
6  end:  movl %eax, %ebx
```

The static basic blocks are: $\{1\}$, $\{2,3\}$, $\{4,5\}$, $\{6\}$. The dynamic basic blocks (assuming a first jump to instruction 1) are: $\{1,2,3\}$, $\{4,5\}$, $\{2,3\}$, $\{6\}$.