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

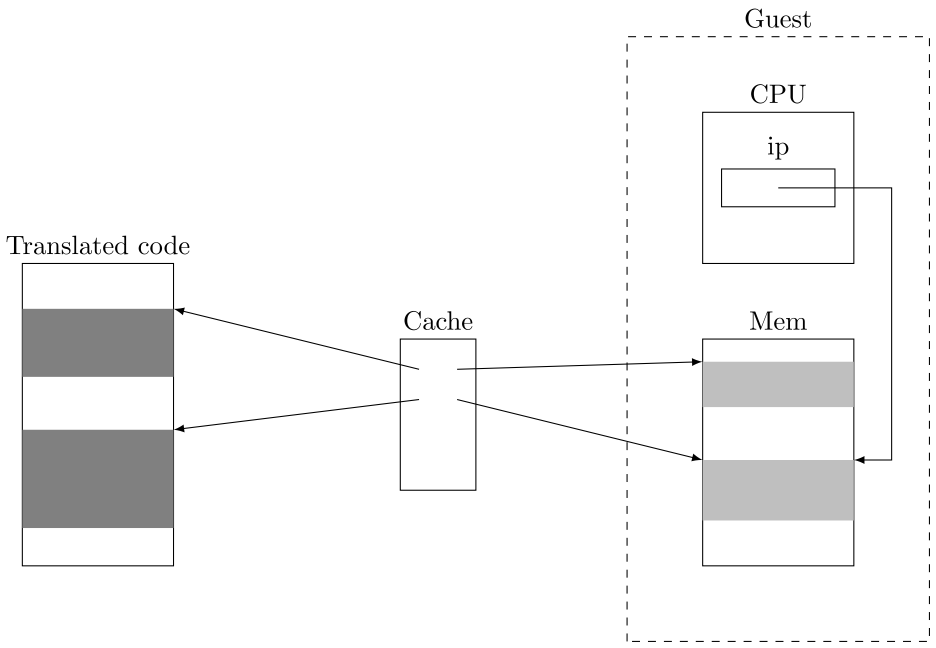

Figure 1: Data structures used in binary translation.

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.

Fig. 1 shows the main data structures used in binary translation. The guest system is represented by the usual data structures implementing the target CPU and memory. The light gray areas in the Mem data structure represent dynamic basic blocks. The cache binds together each (currently translated) dynamic basic block with its translation (dark gray areas). Translations are kept in a memory area suitable for host execution (e.g., allocated via mmap()).