# Binary Translation

G. Lettieri

5 Oct. 2018

## 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* (DBB for short). A DBB 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 DBBs is that it is otherwise very difficult to identify all the code in the guest memory, since code looks just like data. DBBs 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. DBBs allow us to only translate code that the guest CPU is actually going to execute.

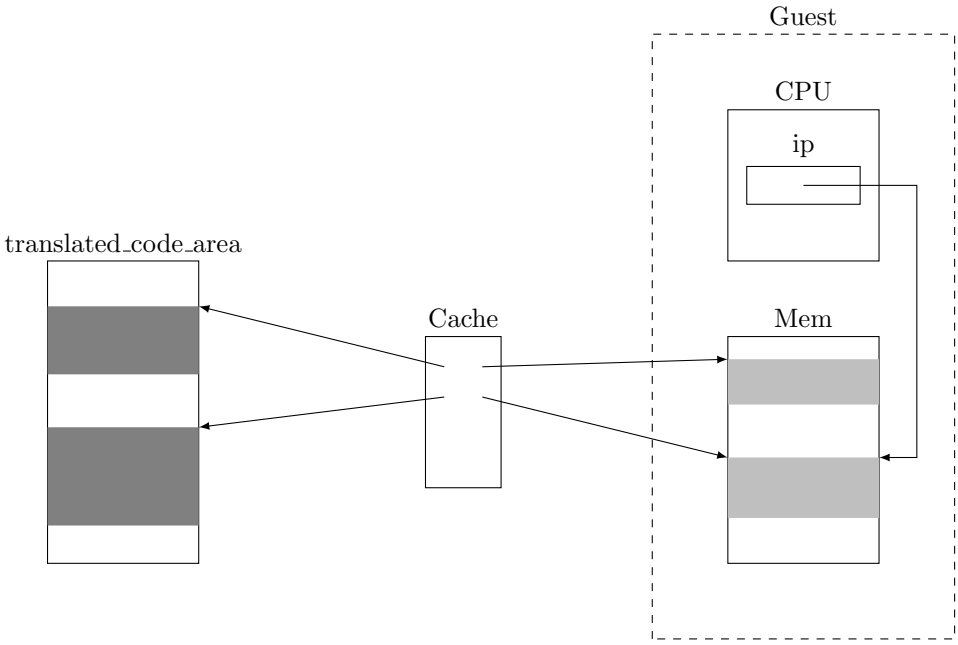


Figure 1: Data structures used in binary translation.

We call Translated Block (TB) the translation of a DBB. DBBs contain target instructions in the target memory (and in its emulation in the virtual machine), while TBs contain host instructions. A TB is identified by the guest address of the first instruction in the corresponding DBB 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 TB 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.

# 2 A binary translator for the Manchester Baby

We now create a complete binary-translator for the Manchester Baby machine, following the general idea outlined above. The task is made easier by the fact that the machine has very few instructions, a very small memory and essentially no specialized I/O. Nonetheless, it is sufficiently complex to understand the main problems and strenghts of this technique. Binary emulation can be activated by passing the `-b` switch to the `mbaby` emulator.

## 2.1 Creating and executing code at runtime

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 from C++.

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 *translated_code_area = mmap(
    NULL,                                   // preferred address: let the
                                            // kernel choose
    MAXHOSTCODE,                            // requested size
    PROT_WRITE|PROT_READ|PROT_EXEC, // protection bits: rwx
    MAP_PRIVATE|MAP_ANONYMOUS,      // not backed by a file,
    -1, 0                                   // file descriptor and offset:
                                            // unused with MAP_ANONYMOUS
);
if (translated_code_area == MAP_FAILED) {
        ...
}
```

If the call succeeds, translated_code_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 translated_code_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.

In the first case, the code stored in the translated_code_area must follow the language linkage-rules.

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):

```
unsigned char code[] = {
    0x55,                   // pushq %rbp
    0x48, 0x89, 0xe5,       // movq %rsp,%rbp
    0x89, 0x7d, 0xfc,       // movq %edi,-4(%rbp)
    0x89, 0x75, 0xf8,       // movq %esi,-8(%rbp)
    0x8b, 0x55, 0xfc,       // movq -4(%rbp),%edx
    0x8b, 0x45, 0xf8,       // movq -8(%rbp),%eax
    0x01, 0xd0,             // addq %edx,%eax
    0x5d,                   // popq %rbp
    0xc3                    // retq
};
```

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<unsigned char*>(translated_code_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 translated_code_area).

Now we have to actually execute the machine code. As we said earlier, an option is to cast translated_code_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>(translated_code_area);

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

For the Manchester Baby the exec function will receive a pointers to the current VM state (containing the CI and A registers, and the M array) and an additional parameter whose purpose will be explained in Section 2.4 below.

## 2.2 Cache of Translated Blocks

For each TB we must remember the range of the corresponding guest instructions (the DBB) and host address where the translated code has been stored. We therefore define a TB descriptor with the following fields:

```
struct tb_t {
    /* range of guest addresses of the DBB */
    int32_t g_beg;
    int32_t g_end;

    /* range of bytes that contain the TB */
    uint8_t* h_beg;
    uint8_t* h_end;

    ...
};
```

The DBB goes from g_beg (included) to g_end (excluded). The TB goes from h_beg (included) to h_end (excluded).

We need a cache to store the TBs. Given a guest instruction pointer, the cache must be able to tell which is the corresponding TB, if any, as quickly as possible. An hash table is usually employed here, but the small size of the Manchester Baby memory (just 32 words) allows us to use a much simpler data structure: an array with an entry for each possible instruction pointer:

```
struct chache_t {
    tb_t* cache[32];

    tb_t* find(int32_t CI)
    {
```

```
        return cache[CI];
    }

    void add(tb_t *);

    ...
};
cache_t cache;
```

Initially, the cache is empty and all entries contain nullptr. To add a Translated Block pointed by tb to the cache we just do

```
void cache_t::add(tb_t* tb) {
    // invalidate the cache is there is no room
    ...
    cache[tb->g_beg] = tb;
    ...
}
```

The host code is stored in a "code area" allocated as described in Section 2.1 above. When there is no room for a new TB, we have to remove some older TB from the cache, to reuse its space in the code area. This is complicated by the fact that the TBs have different sizes. We adopt a very simple solution (also adopted by QEMU): when the code area fills up, we flush the *entire* cache. If the code area is big enough, this will happen sufficiently rarely and will not hurt performance very much.

## 2.3 Finding and translating the DBBs

The translate() function receives a guest instruction pointer. It must find the DBB that begins at that instruction, then create the corresponding TB.

For the Manchester Baby emulator, we structure the generated TB as a function that receives the current VM state and returns the VM state resulting from the emulation of the entire DBB. The general structure is as follows:

```
1  tb_t* translate() /* assume CI, A and M are global variables */
2  {
3      tb_t *tb = new tb_t(); /* allocate a new TB */
4
5      append_prologue(tb);
6      int32_t ci = CI; /* local copy of CI */
7      for (;;) {
8          /* fetch and decode the next instruction */
9          int32_t pi = M[ci];
10         int32_t opcode = (pi & OPCODE_MASK) >> 13;
11         int32_t addr = pi & ADDR_MASK;
12         /* add the translation for this instruction */
```

```
13          switch (opcode) {
14          case JMP: append_jmp_translation(tb, addr); goto out;
15          case JRP: append_jrp_translation(tb, addr); goto out;
16          case LDN: append_ldn_translation(tb, addr); break;
17          case STO: append_sto_translation(tb, addr); break;
18          case SUB: append_sub_translation(tb, addr); break;
19          case CMP: append_cmp_translation(tb); goto out;
20          case STP: append_stp_translation(tb); goto out;
21          }
22          /* add the code to increment CI */
23          append_incCI(tb);
24          /* move to the next instruction */
25          ci++;
26      }
27  out:
28      append_epilogue(tb);
29      return tb;
30  }
```

The basic idea is to build the TB piecewise, appending the necessary host code depending on the fetched guest instruction. We start by inserting the C++ function prologue (line 5), then we fetch the guest instructions sequentially (lines 9–26), appending code along the way. We stop when we encounter a JMP, JRP, CMP or STP (lines 14, 15, 19 and 20). Whenever we fetch a new instruction after the first one, we must remember to append the code that increments the guest CI (line 23).

Note that the translation of some guest instructions depend on the value of the addr field. For example, the translation of "LDN 20" must read from M[20]. Therefore, we pass the addr field to the helper functions that need it.

The translation of each instruction is just sequence of bytes that must be appended to the current TB. In our case, the sequence is constant apart from a few bytes that may depend on the value of addr. Examples of the generated TBs can be observed by running the `mbaby` emulator with a verbosity level of 4 (e..g, `mbaby -b -vvvv < afp.snp`). In this way, whenever a new TB is created, the emulator will print the corresponding DBB range, the binary TB contents and its disassembly.

## 2.4 Self modifying code

The most complex problem is how to handle guest code that modifies itself. Assume first that code in one DBB $B_1$ modifies code in another DBB $B_2$. If we have already translated $B_2$, we need to invalidate the corresponding TB. First, we have to detect the fact that the $B_2$ has been altered. Since code is just data somewhere in guest memory, *any* guest write to memory may modify code. We have two main tecniques to detect code modification:

- (software only) remember in some data structure the range of guest addresses containing code that we have already translated and look up the destination address before each guest memory write (e.g., we can keep a bitmap with one bit for each page);
- (with the help of the host hardware) write-protect the guest memory that contains already translated code, then invalidate the relavant cache entries on page fault.

For the latter technique, remember that the binary translator is generally an unprivileged program running on some OS, so it does not have direct access to page tables and it cannot directly intercept page faults. However, in Unix-like system, we can proceed as follows:

1. Allocate the guest memory using mmap() (this is necessary to be able to use mprotect() below);
2. use the signal () system call to intercept the SIGSEGV signal;
3. whenever we translate a DBB, use mprotect() to ask the kernel to remove the PROT_WRITE permission from the corresponding pages in guest memory;

Now, when our process will try to write into the protected pages, the kernel will send it a SIGSEGV signal. Normally, this signal causes the abnormal termination of the process, returning to the shell which then prints "Segmentation fault". However, in step 2, we have asked the kernel that we want to handle SIGSEGV by our own. All we need to do is to call signal () with the number of the signal we are instered in (i.e., SIGSEGV) and a pointer to a function of our own. Whenever the process receives the selected signal, it executes our function instead of terminating. We can use this function to invalidate the translation cache.

Since the Manchester Baby memory is so small, our emulator can use a bitmap with a bit for every word. This bitmap is updated every time a tb is added to the cache, by setting all the bits from tb−>g_beg to tb−>g_end−1. The bitmap is also accessible to each TB. The translation of each "STO $a$" instruction must first check that the $a$ bit is not set in the bitmap. If it is, the instruction is trying to change some previously translated DBB and all the corresponding TBs must be invalidated (note that there might be more than one DBB that contained the modified instruction).

The most complex case occurs when a DBB $B$ tries to modify *itself*. Now we cannot simply invalidate the TB, since this is the very code we are executing. Probably the best thing we can do is to switch to emulation as soon as we detect the write, and continue to emulate one instruction at a time until we get out of the DBB. By "switch to emulation" we mean that our binary translator must also include an emulator (like the one we studied in the first lectures) and, when needed, it must be able to temporarily stop translating dynamic basic blocks and start fetching and executing one guest instruction at a time.

# 3 Optimizations

Recall from the first lecture the basic property of emulation/virtualization: we take snapshots of the guest and target states and guarantee that each guest snapshot is the representation of the corresponding target snapshot; on the other end, we do *not* guarantee any correspondence of guest and target states *between* two snapshots. In binary translation, snapshots are taken just before the execution of each DBB. During the execution of each basic block we are free to rearrange, omit and otherwise optimize the operations, provided that we always obtain the correct snapshot in the end.

There are several optimizations that the binary translator can use during the translation of a dynamic basic block. Here we mention only the most important.

## 3.1 Constant propagation

Constant propagation consists in replacing a register or memory operand with an immediate operand. This can be done whenever the content of the register or memory operand is known to the optimizer. E.g.,

```
movl $0, %eax
...
incl %eax
movl %eax, %ebx
```

If there are not other updates to **%eax** between the inizialization and the increment, then the optimizer can replace the last instruction with **movl** $1, **%ebx**.

## 3.2 Dead code elimination

This optimization consists in removing (host) code that cannot affect the state. A typical example is the removal of the code that computes the condition codes (e.g., the contents of the EFLAGS register in IA32). Assume, for example, that the that the target machine has an IA32 architecture and that the guest code contains something like

```
ADDL %EAX, %EBX
SUBL $1, %EBX
```

The flags computed by the first addition will be overwritten by the following subtraction, so there is no need to compute them.

Another example is the value of the guest instruction pointer register: this is updated after the execution of each guest instruction, but its value is only used during a relative jump (to sum it to the jump offset) or a subroutine call (e.g., to store it on the stack). The translated code can avoid computing the value of the guest instruction pointer and only update it at the end of the dynamic basic block. In the example translation of the previous lecture, we already used this optimization: in that case, the guest instruction pointer was updated only

at the end, by loading it from the guest stack during the emulation of the **RET** instruction.

### 3.3 Register allocation

Instead of repeatedly reading and storing the guest registers from the guest CPU data structure in memory, we can allocate some host register to store the content of a guest register for the entire duration of a TB. At the beginning of the TB we load the contents of the guest register into the selected host register, we use the host register during all of the translation, then we store the value of the host register back into the guest register at the end of the TB. Note that registers are generally faster than memory, even than fist level caches. A modern CPU may need 4 cycles to access its cache, but registers are always read and written in one cycle.

There is no need to permanently map a guest register to the same host register. The mapping can be optimized on a per-block basis. However, imporant registers that are used very often in most blocks may be always kept in the same host register, thus saving the loads and stores at the beginning of each translated block. In IA32 this may be the case for the guest ESP register, for example.

### 3.4 Lazy condition code computation

Even if we can omit most condition code computations using dead code elimination, we still have to compute the final value of the flags register ad the end the TB. This is because we do not know if the next TB will need it. However, we can do better: instead of computing the flags, we store the operands and the result of the last guest instruction that would have updated them. We will use these values to compute the flags later, but only if we fetch a guest instruction that needs them. In most cases we will never compute the flags, since we will first fetch a guest instruction that overwrites them.

### 3.5 Translated block chaining

In the standard binary translation algorithm we return to the the main loop after the execution of each TB; then, we use the current value of the guest CPU instruction pointer to look up the translation cache, eventually translating a new DBB in case of miss. Once we have the pointer to the new TB, we can patch the previous block so that, the next time it will be executed, it will directly jump to the new TB, instead of returning to the main loop. This can be done even if a TB ends with a conditional jump: the TB will jump to one of two possible next TBs.

# 4 Problems

Depending on the target architecture, the binary translator also have to solve a set of complex problems, in order to correctly emulate the target states.

## 4.1 Handling interrupts

The target will check for interrupts after the execution of each instruction. The binary translator can emulate this by inserting the code that checks for interrupts after the translation of each guest instruction.

However, since there is generally no guarantee on the exact timing of interrupt arrival, we can adopt a more efficient strategy and only check for interrupts whenever the host returns to the main loop. This is acceptable if basic block chaining is not used, since each translation block will contain no loops. If chaining is used, however, the emulator may spend an unbounded amount of time in the exec() function, without returning to the main loop. In this case, chaining must be disabled whenever there is a pending interrupt This is essentially equivalent to putting the interrupt checking code at the end of each TB, before the jump to the next block.

Note that this optimization may have a confusing effect: since we are artificially disabling interrupts for the duration of a DBB, we may be hiding synchronization errors in the guest code.

## 4.2 Handling faults

The problem with faults is that they may occur in the middle of a DBB and, unlike interrupts, they typically cannot be delayed. We can implement this with a lonjmp() to the main loop, where the corresponding setjmp() will then read the guest interrupt table and change the guest instruction pointer accordingly. The problem, however, is that the guest fault handler may need the contents of all guest registers *at the time of the fault*, while the translated code has been optimized assuming that the guest state was needed only at the end of each TB. Since we certainly do not want to remove optimizations, we need to reconstruct the exact state of the guest whenever a fault is generated. This is still good performance-wise, since faults are rare.

The complexity of the guest state reconstruction depends on the optimizations that have been used during the translation of the block.

- If we have used register allocation, we can store a table with the mapping between guest and host registers together with each TB. If a fault is generated during the execution of the TB, we can use the table to update the contents of the guest registers before jumping to the guest fault handler.
- We have to consider possible faults when we apply dead code elimination. Essentially, faults may insert code between two instructions, thus creating users for otherwise dead values. If we can identify before hand all the instructions that may cause a fault (e.g., division by zero can only be

generated by a division), then we can make sure that the guest state is updated before the execution of any such instruction.

- In the most complex cases we may need to return to a known consistent state and *switch to emulation* until we hit the fault again.

## 4.3 Virtual memory

In the example translation of the previous lecture we also omitted to consider the guest virtual memory. In a complete translation, we need to translate every guest address before using it to access the guest memory. This can be done by inserting the code for the translation in all the guest instructions that have memory operands. The code to be added is essentially the same code that we already considered when we talked about virtual memory in emulators, and can benefit from the same optimizations (e.g., a software TLB). If a page fault needs to be generated, we need to apply all the considerations that we have already discussed for the general fault case.