How Not To Build A Binary Translator

Aimilios Tsalapatis, Kenneth R Hancock

Binary translation is a common method used to speed up the execution of software components in systems that cannot run them natively. The process consists of dynamically creating native machine code corresponding to the foreign binary. While the concept is straightforward, there is a slew of ways the construction of the translator can go awry, as will be outlined in this paper. We attempt to enumerate difficulties encountered while writing such an assembler from the 6502 to the x86-64 architecture. Examples include dynamically rewriting the emitted code to connect basic code blocks, properly allocating host registers to the guest code, and mixing emulation and binary translation. We discuss the nature of these problems, and offer advice for anyone attempting to use this method.

1. Introduction

1.1. Arguments for Binary Translation

Binary Translation is the process through which binary program which was meant for a specific hardware platform can be run natively in another system. Normally, binary programs that cannot be run have to be interpreted instruction by instruction by an emulator in order to reproduce its behavior. However, this approach is very slow, since the amount of native instructions executed for each foreign one can be exceedingly high.

A way to speed up the execution of programs on arbitrary architectures is using binary translation alongside emulation. This technique takes advantage of the fact that most architectures are composed of common commands, like moves and rudimentary binary operations. Therefore, it is possible to create native executables that perform the same function as the original program. The amount of code needed run is lower, and execution is faster when compared to pure emulation.

Another use case for binary translation is software virtualization. Some programs need to access resources that are not accessible, because they do not exist in the system. Alternatively, the program does not have enough privileges to modify the component, like in the case of userspace applications attempting to change their page table register.

Virtual Machines (VMs) are such a class of programs. The kernel running in the VM is designed to have full access to the hardware, which in this case includes sensitive processor registers and emulated devices. A translator has the responsibility of replacing code that attempts to do such accesses with alternative instructions that divert flow to the translator. The program emulates the resource in such a way that the VM can keep executing, and diverts control back to it. Popular emulators that employ binary translation, such as Bochs[1] and QEMU[2], utilize exactly this kind of scheme in order to emulate devices like disks on demand.

1.2. Binary Translation Internals

Binary translation, much like all forms of compilation, hinges on the concept of basic code blocks. Basic blocks are sequences of machine instructions that do not contain control flow commands like jumps and branches, and are the the smallest unit that binary translation works with. More concisely, we can only jump to the start of basic blocks. Programs are sectioned into multiple basic blocks, which are organized into a graph whose edges denote a possible execution where the destination is run immediately afterthe source.

The concept of basic blocks is useful exactly because all programs enter the block from the beginning and exit from the end. That means that, as long as the sum of its operations produce the same outputs for all possible inputs, it is possible to transform the instructions without losing program correctness. This is the fundamental principle behind compiler optimizations.

An emulator that employs binary translation goes through the program, creating and then executing a sequence of native instructions corresponding to the current block. The key insight is that program correctness is independent of the underlying platform, and is dependent only on creating the proper output for all inputs. Therefore, it is possible to create a native program that does exactly that, using the original binary as a blueprint.

2. Implementation

For the implementation of the binary translator we opted to expand on our existing emulator project, written in C++. The main idea was to have a translation cache in the emulator, which is queried for every basic block. If an existing translated block is found, then the emulator would instead execute that instead of going through the 6502 block instruction by instruction.

Creating executable code is straightforward, if somewhat awkward. First, an area of memory needs to be mmap()'ed as executable. The translator then emits instructions corresponding to the 6502 code by directly writing binary code into the buffer. Unfortunately, it is necessary to hardcode the opcodes in binary, since it is not possible to define code as data in C++. While it would in fact be simple to do so in pure x86 assembly, the fact that we are writing in a high level language actually makes the task more difficult.

x86_64 assembly also has multiple versions of the same instruction, depending on the size and type of the operands that the opcode takes. While the instruction set does succeed in its goal of being backwards compatible with as many past iterations of itself as possible, one disadvantage of this design choice is that there are a lot of opcodes that are obsolete for modern programs. This one to many relation between instructions and opcodes means that a thorough understanding of the platform is necessary when deciding on a representation for a 6502 instruction.

A good example of the peculiarities of the x86 instruction set is the JMP instruction. This operation can execute either a near or a far jump, depending on the opcode used. The differences between these two jumps are nonexistent in 64bit mode, since the distinction originates in the way the older members of the x86 family of processors implemented segmentation. This method of partitioning memory has been rendered obsolete by virtual memory, however the opcodes are still there.

In order to find out which version of the instruction is actually the desirable one, we used a disassembler on compiled programs and inspected the opcodes corresponding to the JMP instructions used there. This method turned out to be moderately useful: The disassembled code was always correct, however finding out the exact version of the instruction needed is not always guaranteed. Looking for a perfect match instead of just a similar version was necessary because of the 'polymorphism' that x86 instructions exhibit, which for example means that some instructions have different opcodes for 32-bit and 64-bit versions.

A major problem we encountered was the linkage of the basic blocks into a self-contained x86 binary. As demonstrated in Figure #, entering the emitted code is done by saving all registers to be used by the generated instructions. As an optimization, the current values of the 6502 registers are always saved in specific registers for easy access. Code generation is also simpler, since 6502 operations that act on the registers always get translated to x86 instructions that have a register as an operand.

```
asm volatile (
    "pushq %%rax\n"
    "pushq %%rbx\n"
    "pushq %%rcx\n"
    "pushq %%rdx\n"
    "pushq %%r10\n"
    "xor %%rax, %%rax\n"
    "xor %%rbx, %%rbx\n"
    "xor %%rcx, %%rcx\n"
    "xor %%rdx, %%rdx\n"
    "xor %%r10, %%r10\n"
    "mov %[X], %%bl\n"
    "mov %[Y], %%cl\n"
    "mov %[A], %%dl\n"
    "callq *%[code]\n"
    "mov %%bl, %[X]\n"
    "mov %%cl, %[Y]\n"
    "mov %%dl, %[A]\n"
    "popq %%r10\n"
    "popq %%rdx\n"
    "popq %%rcx\n"
    "popq %%rbx\n"
    "popq %%rax\n"
    : [A] "+m" (A),
      [X] "+m" (X),
      [Y] "+m" (Y)
    : [code] "r" (code)
);
```

Figure 1: The inline assembly code used when entering and exiting the translated code.

The issues that came up have to do with the interplay of the binary translator with the emulator. Normally, the emulator works at the instruction level, and does not need to be concerned with basic blocks. This means that the existing infrastructure could not be used in combination with the binary translator without modifications: Each time translated code is executed, the emulator is not capable of finding the next instruction that is supposed to be executed.

Monitoring the control flow of the generated code also proved to be a problem when connecting instructions together. When connecting basic control blocks, the main idea is to overwrite the return function that has been attached at the end with a jump to the next block. This modification is especially complex with conditional branches. The basic blocks are disjoint in memory, since they have variable size and are created in a random order.

This means that the code for a relative branch looks like the following:

```
        if (condition)
                jmp to taken
not taken:
        jmp taken_path
taken:
        jmp not_taken_path
```

The main difficulty lies in the way we search for x86 basic blocks that should be linked together, because they originate from adjacent 6502 basic blocks. Conditional branches are particularly tricky, because blocks that end with such instructions have two possible exit points.

When first executing the basic block it is necessary to find out which path should be taken. There are two alternative designs that can resolve this problem. One includes storing an identifier in a specific area in memory, where in can be inspected by the C++ code. The other stores the flags register instead of an identifier, and does so just before executing the branch. The C code then inspects the branch instruction and the flags register, and basically replays the instruction in order to find out the result. The first approach is more straightforward, but is more complex to emit, and, as mentioned earlier, emitting binary code from C++ is extraordinarily unwieldy.

3. A Better Design

3.1. Overview

The general overview of a binary translator goes through the following steps. And can be seen through Figure 2

3.1.1. Retrieval. The translato reads until a jump or a label is found, making note of every use of a label it encounters as it translates. There are two cases in which can occur at this point, either the end label has been encountered or it has not. The easy case being if it has been encountered, in which case we use a label cache and translate that instruction to jump to jump to the code section pointed to by the cache.

If it has not, then we translate it such that when the assembly is executed it moves a unique identifier for that label into a piece of shared memory and returns. This allows us so when we return from the assembly execution (which may have been running through multiple different code sections), we know exactly the label it requires, and also that it requires it immediately. We can then scan the original binary for this label and start a new execution at that label.

Code sections are just mapped pieces of memory that are executable.

3.1.2. Edit. Once a basic code block has been fully translated, if the block itself started with a label then it must notify and edit every block that is currently in the request label map under said label. And retranslate those blocks (we know there is possible optimizations for this, but for simplicity we will just force a retranslate).

3.1.3. Execute. Execution occurs through the boot-strapped assembly code seen previously. Code sections are either executed seamlessly through each other or upon exit we use unique identifier.

References

[1] K. P. Lawton, “Bochs: A portable pc emulator for unix/x,” *Linux Journal*, vol. 1996, no. 29es, p. 7, 1996.

[2] F. Bellard, “Qemu, a fast and portable dynamic translator.,” in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.

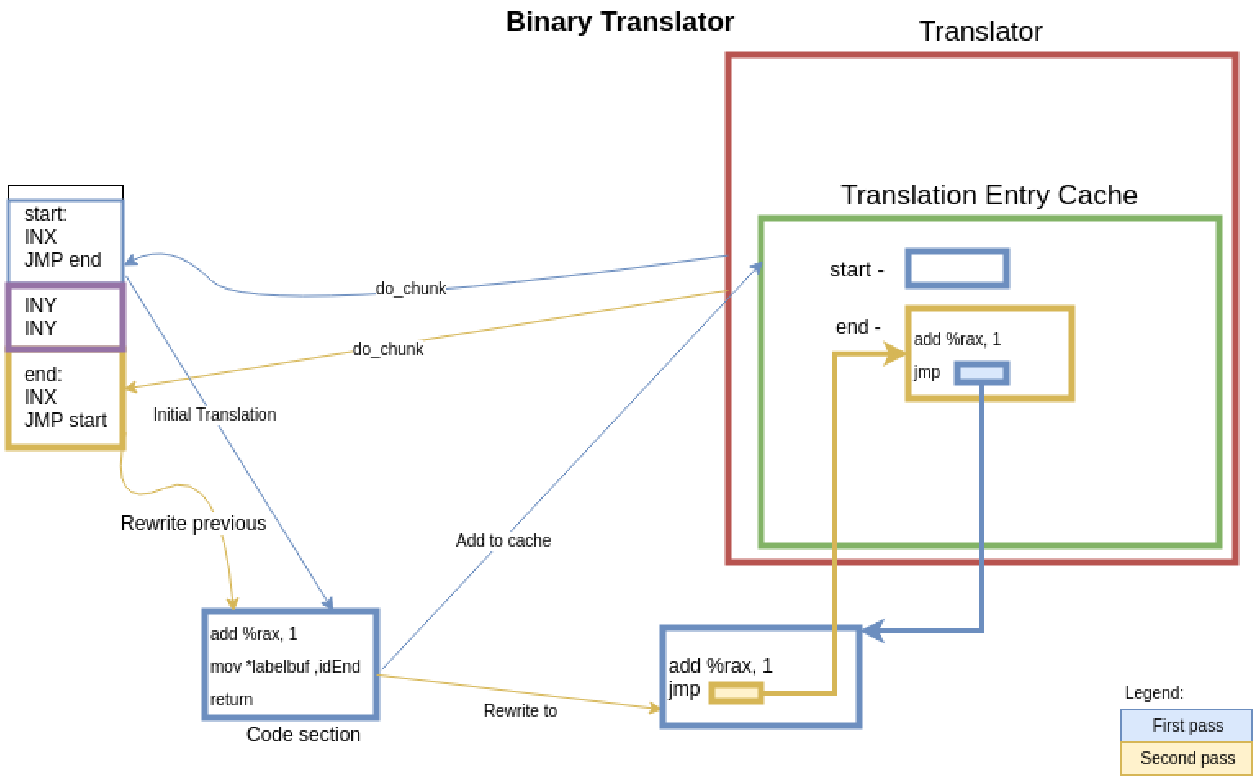

Figure 2: The general overview of a simple binary translation