View metadata, citation and similar papers at core.ac.uk brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

# Dynamically Translating x86 to LLVM using QEMU

*Vitaly Chipounov and George Candea*
School of Computer and Communication Sciences
*École Polytechnique Fédérale de Lausanne (EPFL), Switzerland*

## 1 Introduction

QEMU [1] is a system emulator that can run unmodified guest operating systems on a host OS, where the guest and host CPU architecture can be different. For example, QEMU can run x86 guest OSes on a MIPS host, or even x86 on x86 (e.g., a Windows guest on a Linux host). QEMU emulates a complete system including processors, devices, and chipsets. More implementation details regarding QEMU are available in [1].

In this paper, we focus on the design and implementation of the LLVM backend for QEMU. LLVM [5] is a compiler framework which can optimize programs across their entire lifetime, including compile-time and run-time. It also performs offline optimizations. The LLVM backend converts the guest instruction set to LLVM bitcode, optimizes this bitcode, and turns it back to x86, using the JIT capabilities of the LLVM run-time. We build upon an existing attempt to write an LLVM backend for QEMU [6].

The LLVM backend can be used for several purposes. We interfaced QEMU with the KLEE symbolic execution engine to test OS kernel, drivers, and applications [3]; we also use the LLVM backend for device driver testing [4] and reverse engineering [2].

The paper is structured as follows. First, we explain how QEMU runs unmodified guest OSes (§2), then we describe the specificities of the LLVM backend (§3), how it is implemented (§4), evaluate its performance (§5), discuss the limitations (§6), and conclude (§7).

## 2 Dynamic Binary Translation

QEMU has a dynamic binary translator (DBT) which converts binary code from the guest CPU architecture to the host architecture. For example, the DBT can take x86 guest code and turn it into an equivalent sequence of instructions natively executable on a SPARC host. Translation consists of two phases: converting guest code into a sequence of micro-operations and turning these micro-operations into executable code (Figure 1).

First, the DBT converts the guest binary code into a sequence of simple micro-operations, packaged in a translation block. The DBT takes as input the current program counter of the virtual processor and the emulated physical memory. The DBT disassembles all instructions starting at the current program counter until it encounters an instruction that changes the control flow (e.g., call, return, jump, or interrupt). The disassembly process converts each guest instruction into an equivalent sequence of micro-operations. For example, the x86 instruction `inc [eax]` that increments the memory location whose pointer is stored in the `eax` register, is split into a memory load to a temporary register, an increment of that register, and a memory store. The succession of micro-operations forms a translation block.

Second, the DBT turns the translation block into code executable by the host processor. The DBT maintains a dictionary of micro-operations. This dictionary contains a mapping between micro-operations and the equivalent sequences of host architecture binary code. This binary code performs the task of the corresponding micro-operation when run on the host processor. The DBT cycles through each micro-operation contained in the translation block and emits the corresponding sequence of host machine instructions by using the dictionary. The result is a block of host machine code that performs the same operations as the guest OS code.

## 3 An LLVM Backend

We extend the QEMU dynamic binary translator to generate LLVM bitcode from x86 (Figure 2). The translator takes x86 code, disassembles it to micro-operations, maps the micro-operations to their LLVM counterparts, and outputs LLVM bitcode. The generated bitcode can be optionally translated to x86, using the Just-In-Time (JIT) compiler of the LLVM framework.

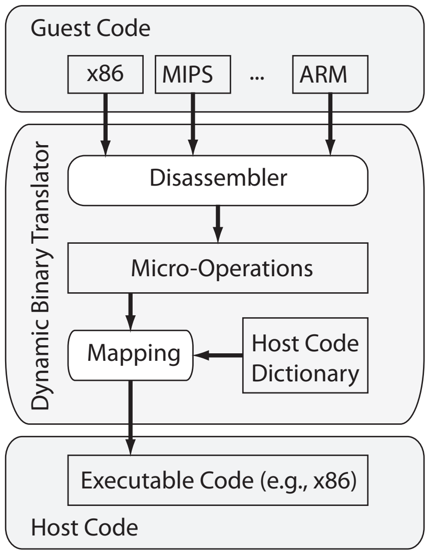

Figure 1: The QEMU dynamic binary translator

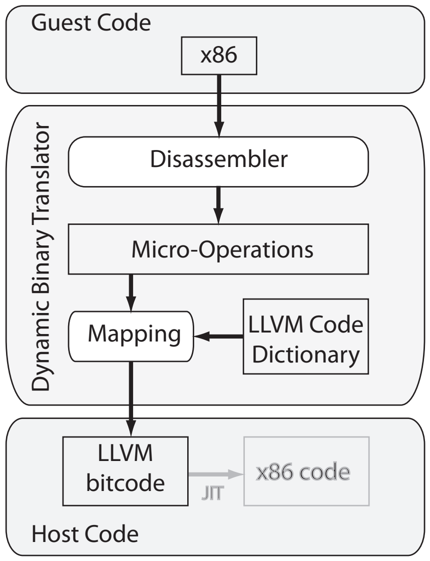

Figure 2: An LLVM backend for QEMU

In this section, we give a brief overview of LLVM (§3.1), show how to convert micro-operations to LLVM code (§3.2), how to optimize the resulting code (§3.3), and how to execute it (§3.4).

## 3.1 The LLVM Assembly Language

LLVM uses a compact RISC-like instruction set and a static single assignment (SSA) code representation. The LLVM instruction set consists of about 30 opcodes. Only the `load` and `store` instructions can access memory, all other instructions operate on virtual registers. LLVM virtual registers resemble registers in traditional instruction set architectures (e.g., x86), except that there is no bound on how many of them can be used by LLVM code, which enables SSA representation. In SSA, each register can be assigned only once. Hence, SSA also provides a `phi` instruction. This instruction assigns a value to a variable depending on the location from where the control flow arrived. Without this instruction, it would be impossible to modify the same variable in two different branches.

The conciseness and SSA-form of LLVM make LLVM programs amenable to complex analysis and transformations. The LLVM code implicitly embeds the program's data flow and def-use graphs. This enables transformations like function inlining, constant propagation, dead store removal, etc. These transformations are a key part of the x86 to LLVM translator, as we shall see in §3.3.

## 3.2 Translating Micro-Operations to LLVM

The DBT maps the micro-operations resulting from the disassembly phase to LLVM functions, by using the code dictionary. Each micro-operation is assigned to one LLVM function that implements the behavior of that micro-operation. The code for the micro-operation is originally written in C and compiled by `llvm-gcc` into LLVM bitcode. The set of LLVM functions implementing micro-operations' behavior is stored in the code dictionary. Listing 1 shows a micro-operation simulating a jump to a code location by assigning a value to the program counter register, and another operation computing the XOR of two registers. Micro-operations operate on the virtual CPU state, which is passed to them as a parameter.

```
void op_jmp_T0(CPUState *env) {
        env->eip = env->t0;
}

void op_xorl_T0_T1(CPUState *env) {
    env->t0 ^= env->t1;
}
```

Listing 1: Examples of micro-operations.

After the DBT has mapped the micro-operations to corresponding LLVM functions, it transforms the translation block into LLVM bitcode. The DBT converts micro-operations to LLVM function calls, as shown in Listing 2. In this example, the translation block is composed of seven micro-operations. The translation block is encapsulated into an LLVM function taking as a parameter a pointer to the processor state. The generated LLVM code is equivalent to the original x86 code in ev-

```
define i32 @dyngen(%struct.CPUX86State*) {
EntryBlock:
 %1 = alloca i32
 %2 = getelementptr i32* %1, i32 0
 store i32 -1, i32* %2
 call void @op_save_exception(%0, i32 -16, i32 0)
 call void @op_movl_T0_im(%0, i32 0xF000)
 call void @op_movl_T1_imu(%0, i32 0xE05B)
 call void @op_movl_seg_T0_vm(%0, i32 96)
 call void @op_movl_T0_T1(%0)
 call void @op_jmp_T0(%0)
 call void @op_movl_T0_0(%0)
 %3 = load i32* %2
 ret i32 %3
}
```

Listing 2: Sequence of micro-operations for `jmp 0xF000:0xE05B` in LLVM format.

ery aspect. In particular, it manipulates the processor and memory state in the same way.

## 3.3 Optimizing the Generated LLVM Code

The output of the translation phase is suboptimal. Each micro-operation invocation is encoded by a function call. Since micro-operations are small (they often have only one line of code), the overhead of call/return instructions becomes large. Another source of overhead can arise from suboptimal guest code. For example, the guest code could save a value to a register, then overwrite this value without actually using it. This leads to larger than necessary LLVM bitcode.

We supplement the x86 to LLVM translator with an optimization phase to optimize individual translation blocks. The DBT reuses the optimization passes provided by the LLVM infrastructure. To alleviate the overhead caused by many calls to short functions, the DBT applies the inlining pass on the translation blocks to inline all function calls. Listing 3 shows the effect of inlining on the code in Listing 2. Likewise, the DBT applies passes that eliminate redundant instructions.

## 3.4 Turning LLVM Code into x86

The translation phase outputs LLVM code, targeted at a "fictive" LLVM host, just as an ARM-to-x86 translator would target an x86 host. Generating LLVM code can be useful for performing complex optimizations and analysis. For example, the DBT could convert large portions of a program to LLVM dynamically to apply various global analysis and optimizations techniques on the program.

It is also possible to compile the LLVM code into the instruction set of the host platform using the LLVM Just-In-Time compiler. The end result is the same as using original QEMU's translator, except that the translation phase adds the step of translating to LLVM. Turning

```
define i32 @dyngen(%struct.CPUX86State*) {
EntryBlock:
 %1 = getelementptr %0, i32 0, i32 5
 store i32 -16, i32* %1, align 1
 %2 = getelementptr %0, i32 0, i32 0
 %3 = getelementptr %0, i32 0, i32 1
 store i32 57435, i32* %3, align 1
 %4 = and i32 61440, 65535
 %5 = bitcast %struct.CPUX86State* %0 to i8*
 %6 = getelementptr i8* %5, i32 96
 %7 = bitcast i8* %6 to %struct.SegmentCache*
 %8 = getelementptr %7, i32 0, i32 0
 store i32 %4, i32* %8, align 1
 %9 = shl i32 %4, 4
 %10 = getelementptr %7, i32 0, i32 1
 store i32 %9, i32* %10, align 1
 %11 = load i32* %3, align 1
 %12 = getelementptr %0, i32 0, i32 4
 store i32 %11, i32* %12, align 1
 store i32 0, i32* %2, align 1
 ret i32 -1
}
```

Listing 3: LLVM code for `jmp 0xF000:0xE05B` after inlining and optimizations.

LLVM into x86 does not require more than a simple call to an LLVM library function to work.

Listing 4 shows the effect of compiling the LLVM code in Listing 3 to x86 using the JIT engine. It is possible to generate any type of native code that the LLVM JIT supports.

```
mov eax,dword ptr [esp+0x4]
mov dword ptr [eax+0x30],0xfffffff0
mov dword ptr [eax+0x4],0xe05b
mov dword ptr [eax+0x60],0xf000
mov dword ptr [eax+0x64],0xf0000
mov ecx,dword ptr [eax+0x4]
mov dword ptr [eax+0x2c],ecx
mov dword ptr [eax],0x0
mov eax,0xffffffff
ret
```

Listing 4: x86 code for `jmp 0xF000:0xE05B`.

## 4 Implementation

The original QEMU DBT maps micro-operations to machine code by copy/pasting binary code. QEMU stores the micro-operation dictionary in the `op.c` file. This file is compiled by a host platform's compiler in order to generate an executable representation of the micro-operations. This representation is stored in a standard object file (`.o`). At translation time, the DBT parses that object file to extract the binary code associated with each micro-operation. The DBT appends the extracted code to a buffer as it processes each micro-operation, before returning this buffer to QEMU for execution.

The x86 to LLVM translator follows the same principle, but due to the nature of LLVM code, it requires numerous adaptations. Like the original DBT, the x86 to LLVM translator pastes together fragments of LLVM code. However, LLVM code requires adaptations to the exception handling mechanisms (§4.2). Moreover, LLVM code does not lend itself to dynamic patching. Dynamic patching is used by the original DBT to handle branches (§4.3) by relocating code and chaining translation blocks (§4.4).

## 4.1 Removing Host-Specific Assembly

The x86-to-LLVM DBT does not use any host-specific inline assembly or directives. It generates translation blocks that contain pure LLVM code. The original DBT uses inline assembly to make the generated code fast. The DBT keeps temporary variables used by the micro-operations in host machine's registers and reserves a global register containing the pointer to the CPU state by using the GCC `__attribute__` directive. This adds unnecessary constraints to the LLVM bitcode in the form of host-specific inline assembly. We remove all architecture-specific directives so that the translated blocks only contain native LLVM instructions.

## 4.2 Precise Exception Handling

Another adaptation is related to precise exception handling. Precise exception handling is the ability of the processor to identify which instruction caused a fault, in order to allow resuming execution after the fault is handled. For instance, it is necessary to know the program counter of an instruction that causes a page fault, to rerun it after the fault is handled.

When an exception occurs, QEMU maps the host CPU instruction pointer to the virtual address of the equivalent guest instruction. This allows QEMU to know which guest instruction caused the exception in order to restart its execution later. Since the translated code is a sequence of copy/pasted executable code of known size, it is straightforward to determine the host-to-guest instruction mapping.

However, LLVM does not allow performing this mapping easily. Indeed, the LLVM code loses the relationship between a guest instruction and its micro-operations, because the inlining and optimization phases reorder the operations. The DBT solves this by explicitly updating the program counter at the start of each instruction, to enable precise exception handling. The program counter is updated by the `op_save_exception` micro-operation.

## 4.3 Branches Within Translation Blocks

Translation blocks may have internal branches transferring control flow to any micro-instruction within that same block. These branches are encoded by a special micro-operation, which maps to a machine code absolute jump instruction after the translation. The absolute target of the jump is relocated by the DBT at runtime, during the copy/paste phase. Relocation consists in patching the address of the jump target depending on the memory location of the translated code, similarly to what a dynamic linker would do to absolute addresses in shared libraries used by a program.

The x86 to LLVM translator uses the LLVM branch instruction for internal control flow transfers. The LLVM bitcode does not use patching, because it is machine-independent and does not need to be aware of the actual code location in memory. It is the responsibility of the JIT engine to ensure at run-time that all branches point to correct locations. Listing 5 shows an example of such an internal branch.

## 4.4 Translation Block Chaining

By default, QEMU continuously fetches new translation blocks by querying the translation block cache or requesting the DBT to translate a new block. Although cache querying is less expensive than translating the same block over and over again, it is still slow.

QEMU can bypass the cache by chaining translation blocks to further speed up execution. The DBT chains the blocks using a machine jump instruction. When the host completes executing a translation block, it jumps to the next block directly, without going through a complex fetching logic. The DBT implements this jump similarly to an intra-block jump, by patching it with the address of the next block, if this block is available. Otherwise, the jump exits the translation block and gives control back to QEMU.

While the original QEMU performs chaining *inside* the translation blocks (using a jump machine instruction), the LLVM-enabled version does it *outside*. The translated LLMV function returns a value that QEMU uses as an index in an array of function pointers specifying the next block in the chain. This value can be 0 (take the true branch), 1 (take the false branch), or -1 (no chaining). Listing 5 shows a translated block implementing a conditional jump, with two outcomes (0 and 1).

## 5 Evaluation

In this section, we evaluate the overhead of the x86 to LLVM translation phase. We compare the time it takes to boot different OSes with two variants of QEMU 0.9.0. The first variant uses the original x86 to x86 DBT, while the second uses the x86 to LLVM DBT. We evaluate OSes of increasing complexity: MS-DOS 6.22, RedHat Linux 2.2.5, Debian Linux 2.6, and Windows XP SP2. The measurements are performed on an AMD Turion64, 2 Ghz and 2GB of RAM.

```
define i32 @dyngen(%struct.CPUX86State*) {
  %1 = alloca i32
  %2 = getelementptr i32* %1, i32 0
  store i32 -1, i32* %2
  call void @op_save_exception(%0, i32 1040544, i32 0)
  ... omitted instructions ...

  /* Conditional jump    */
  %3 = call i32 @op_jnz_ecxw(%0, i32 0)
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %5, label %7

; <label>:5
  .....
  store i32 0, i32* %2  /* Return index 0 */
  call void @op_movl_eip_im(%0, i32 57511)
  call void @op_movl_T0_im(%0, i32 15669600)
  %6 = load i32* %2
  ret i32 %6

; <label>:7
  store i32 1, i32* %2 /* Return index 1 */
  .....
  %8 = load i32* %2
  ret i32 %8
}
```

Listing 5: Translation block chaining in LLVM.

Table 1 shows the boot time when using the original x86 to x86 translator or the x86 to LLVM translator. We observe that QEMU runs out of memory while booting Linux and Windows crashes during logon (the Winlogon process terminates unexpectedly shortly after the login screen appears).

| | Vanilla QEMU | LLVM QEMU | Slowdown |
|---|---|---|---|
| MS-DOS 6.22 | 4s | 80s | 20x |
| Linux 2.2 | 20s | 10min | 35x |
| Linux 2.6 | 70s | - | - |
| Windows XP | 75s | 45min | > 35x |

Table 1: Boot time and overhead for various OSes.

Most of the time is spent in code translation, because turning guest code into LLVM is a complex operation. For example, when booting Linux up to the GRUB menu, 33 seconds are spent inside the translator from a total of 39 seconds. The DBT takes 5 ms on average to translate a bock of guest code to LLVM. Converting LLVM code to x86 using the JIT engine further adds 1 to 2 ms.

Although QEMU has a translation cache which avoids that the same executed blocks get translated over and over again, the translation overhead outweighs by far any benefit of caching. It is about three orders of magnitude slower than the original translator, that only copy/pastes bits of precompiled code using `memcpy`. In contrast, the LLVM engine has to inline all operations, which builds instructions from scratch and uses memory allocation.

## 6 Discussion

The current implementation of the x86 to LLVM translator suffers from two problems: non-negligible overhead and large memory consumption.

Translating guest code to LLVM brings a large overhead, which considerably slows down the emulated system. This can be alleviated by making the code cache persistent. QEMU currently flushes this cache when it is full. Moreover, the contents of the cache are lost when the virtual machine is turned off. One approach towards this issue is to make the cache persistent, e.g., by saving it to disk. This can be accomplished by leveraging the LLVM framework, which allows saving dynamically generated LLVM code to files.

The second problem is the large memory consumption caused by LLVM (the 2.4 version used by the DBT). One cause of it is that the hundreds of thousands of translation blocks generated during execution use a large number of constants (e.g., program counter values). LLVM keeps these constants in a cache and does not destroy them when no more basic blocks reference them.

## 7 Conclusion

In this paper we presented the implementation of an x86 to LLVM dynamic binary translator for QEMU. This translator turns x86 binaries into LLVM bitcode and enables the use of numerous code optimization techniques provided by the LLVM compiler framework. We also demonstrated the use of the LLVM Just-In-Time capabilities to turn LLVM code into x86 that can be run on the host machine.

## References

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX*, 2005.

[2] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys*, 2010.

[3] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *HotDep*, 2009.

[4] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX*, 2010.

[5] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

[6] T. Scheller. LLVM-QEMU Google Summer of Code. http://code.google.com/p/llvm-qemu/, 2007.