QEMU, a Fast and Portable Dynamic Translator

Fabrice Bellard

Abstract

We present the internals of QEMU, a fast machine emulator using an original portable dynamic translator. It emulates several CPUs (x86, PowerPC, ARM and Sparc) on several hosts (x86, PowerPC, ARM, Sparc, Alpha and MIPS). QEMU supports full system emulation in which a complete and unmodified operating system is run in a virtual machine and Linux user mode emulation where a Linux process compiled for one target CPU can be run on another CPU.

1 Introduction

QEMU is a machine emulator: it can run an unmodified *target* operating system (such as Windows or Linux) and all its applications in a virtual machine. QEMU itself runs on several host operating systems such as Linux, Windows and Mac OS X. The host and target CPUs can be different.

The primary usage of QEMU is to run one operating system on another, such as Windows on Linux or Linux on Windows. Another usage is debugging because the virtual machine can be easily stopped, and its state can be inspected, saved and restored. Moreover, specific embedded devices can be simulated by adding new machine descriptions and new emulated devices.

QEMU also integrates a Linux specific user mode emulator. It is a subset of the machine emulator which runs Linux processes for one target CPU on another CPU. It is mainly used to test the result of cross compilers or to test the CPU emulator without having to start a complete virtual machine.

QEMU is made of several subsystems:

- CPU emulator (currently x86[1], PowerPC, ARM and Sparc)
- Emulated devices (e.g. VGA display, 16450 serial port, PS/2 mouse and keyboard, IDE hard disk, NE2000 network card, ...)
- Generic devices (e.g. block devices, character devices, network devices) used to connect the emulated devices to the corresponding host devices
- Machine descriptions (e.g. PC, PowerMac, Sun4m) instantiating the emulated devices
- Debugger
- User interface

This article examines the implementation of the dynamic translator used by QEMU. The dynamic translator performs a runtime conversion of the target CPU instructions into the host instruction set. The resulting binary code is stored in a translation cache so that it can be reused. The advantage compared to an interpreter is that the target instructions are fetched and decoded only once.

Usually dynamic translators are difficult to port from one host to another because the whole code generator must be rewritten. It represents about the same amount of work as adding a new target to a C compiler. QEMU is much simpler because it just concatenates pieces of machine code generated *off line* by the GNU C Compiler [5].

A CPU emulator also faces other more classical but difficult [2] problems:

- Management of the translated code cache
- Register allocation
- Condition code optimizations
- Direct block chaining
- Memory management
- Self-modifying code support

- Exception support
- Hardware interrupts
- User mode emulation

2 Portable dynamic translation

2.1 Description

The first step is to split each target CPU instruction into fewer simpler instructions called *micro operations*. Each micro operation is implemented by a small piece of C code. This small C source code is compiled by GCC to an object file. The micro operations are chosen so that their number is much smaller (typically a few hundreds) than all the combinations of instructions and operands of the target CPU. The translation from target CPU instructions to micro operations is done entirely with hand coded code. The source code is optimized for readability and compactness because the speed of this stage is less critical than in an interpreter.

A compile time tool called `dyngen` uses the object file containing the micro operations as input to generate a dynamic code generator. This dynamic code generator is invoked at runtime to generate a complete host function which concatenates several micro operations.

The process is similar to [1], but more work is done at compile time to get better performance. In particular, a key idea is that in QEMU constant parameters can be given to micro operations. For that purpose, dummy code relocations are generated with GCC for each constant parameter. This enables the `dyngen` tool to locate the relocations and generate the appropriate C code to resolve them when building the dynamic code. Relocations are also supported to enable references to static data and to other functions in the micro operations.

2.2 Example

Consider the case where we must translate the following PowerPC instruction to x86 code:

```
addi r1,r1,-16   # r1 = r1 - 16
```

The following micro operations are generated by the PowerPC code translator:

```
movl_T0_r1          # T0 = r1
addl_T0_im -16      # T0 = T0 - 16
movl_r1_T0          # r1 = T0
```

The number of micro operations is minimized without impacting the quality of the generated code much. For example, instead of generating every possible move between every 32 PowerPC registers, we just generate moves to and from a few temporary registers. These registers `T0`, `T1`, `T2` are typically stored in host registers by using the GCC static register variable extension.

The micro operation `movl_T0_r1` is typically coded as:

```
void op_movl_T0_r1(void)
{
    T0 = env->regs[1];
}
```

`env` is a structure containing the target CPU state. The 32 PowerPC registers are stored in the array `env->regs[32]`.

`addl_T0_im` is more interesting because it uses a *constant parameter* whose value is determined at runtime:

```
extern int __op_param1;
void op_addl_T0_im(void)
{
    T0 = T0 + ((long)(&__op_param1));
}
```

The code generator generated by `dyngen` takes a micro operation stream pointed by `opc_ptr` and outputs the host code at position `gen_code_ptr`. Micro operation parameters are pointed by `opparam_ptr`:

```
[...]
for(;;) {
  switch(*opc_ptr++) {
  [...]
  case INDEX_op_movl_T0_r1:
  {
    extern void op_movl_T0_r1();
    memcpy(gen_code_ptr,
      (char *)&op_movl_T0_r1+0,
      3);
    gen_code_ptr += 3;
    break;
  }
  case INDEX_op_addl_T0_im:
  {
    long param1;
    extern void op_addl_T0_im();
    memcpy(gen_code_ptr,
      (char *)&op_addl_T0_im+0,
      6);
    param1 = *opparam_ptr++;
    *(uint32_t *)(gen_code_ptr + 2) =
      param1;
    gen_code_ptr += 6;
    break;
  }
```

```
  [...]
  }
}
[...]
}
```

For most micro operations such as `movl_T0_r1`, the host code generated by GCC is just copied. When constant parameters are used, `dyngen` uses the fact that relocations to `__op_param1` are generated by GCC to patch the generated code with the runtime parameter (here it is called `param1`).

When the code generator is run, the following host code is output:

```
# movl_T0_r1
# ebx = env->regs[1]
mov    0x4(%ebp),%ebx
# addl_T0_im -16
# ebx = ebx - 16
add    $0xfffffff0,%ebx
# movl_r1_T0
# env->regs[1] = ebx
mov    %ebx,0x4(%ebp)
```

On x86, `T0` is mapped to the `ebx` register and the CPU state context to the `ebp` register.

2.3 Dyngen implementation

The `dyngen` tool is the key of the QEMU translation process. The following tasks are carried out when running it on an object file containing micro operations:

- The object file is parsed to get its symbol table, its relocations entries and its code section. This pass depends on the host object file format (`dyngen` supports ELF (Linux), PE-COFF (Windows) and MACH-O (Mac OS X)).
- The micro operations are located in the code section using the symbol table. A host specific method is executed to get the start and the end of the copied code. Typically, the function prologue and epilogue are skipped.
- The relocations of each micro operations are examined to get the number of constant parameters. The constant parameter relocations are detected by the fact they use the specific symbol name `__op_paramN`.
- A memory copy in C is generated to copy the micro operation code. The relocations of the code of each micro operation are used to patch the copied code so that it is properly relocated. The relocation patches are host specific.
- For some hosts such as ARM, constants must be stored near the generated code because they are accessed with PC relative loads with a small displacement. A host specific pass is done to relocate these constants in the generated code.

When compiling the micro operation code, a set of GCC flags is used to manipulate the generation of function prologue and epilogue code into a form that is easy to parse. A dummy assembly macro forces GCC to always terminate the function corresponding to each micro operation with a single return instruction. Code concatenation would not work if several return instructions were generated in a single micro operation.

3 Implementation details

3.1 Translated Blocks and Translation Cache

When QEMU first encounters a piece of target code, it translates it to host code up to the next jump or instruction modifying the *static CPU state* in a way that cannot be deduced at translation time. We call these basic blocks Translated Blocks (TBs).

A 16 MByte cache holds the most recently used TBs. For simplicity, it is completely flushed when it is full.

The static CPU state is defined as the part of the CPU state that is considered as known at translation time when entering the TB. For example, the program counter (PC) is known at translation time on all targets. On x86, the static CPU state includes more data to be able to generate better code. It is important for example to know if the CPU is in protected or real mode, in user or kernel mode, or if the default operand size is 16 or 32 bits.

3.2 Register allocation

QEMU uses a fixed register allocation. This means that each target CPU register is mapped to a fixed host register or memory address. On most hosts, we simply map all the target registers to memory and only store a few temporary variables in host registers. The allocation of the temporary variables is hard coded in each target CPU description. The advantage of this method is simplicity and portability.

The future versions of QEMU will use a dynamic temporary register allocator to eliminate some unnecessary moves in the case where the target registers are directly stored in host registers.

3.3 Condition code optimizations

Good CPU condition code emulation (`eflags` register on x86) is a critical point to get good performances.

QEMU uses lazy condition code evaluation: instead of computing the condition codes after each x86 instruction, it just stores one operand (called CC_SRC), the result (called CC_DST) and the type of operation (called CC_OP). For a 32 bit addition such as $R = A + B$, we have:

```
CC_SRC=A
CC_DST=R
CC_OP=CC_OP_ADDL
```

Knowing that we had a 32 bit addition from the constant stored in CC_OP, we can recover A, B and R from CC_SRC and CC_DST. Then all the corresponding condition codes such as zero result (ZF), non-positive result (SF), carry (CF) or overflow (OF) can be recovered if they are needed by the next instructions.

The condition code evaluation is further optimized at translation time by using the fact that the code of a complete TB is generated at a time. A backward pass is done on the generated code to see if CC_OP, CC_SRC or CC_DST are not used by the following code. At the end of TB we consider that these variables are used. Then we delete the assignments whose value is not used in the following code.

3.4 Direct block chaining

After each TB is executed, QEMU uses the simulated Program Counter (PC) and the other information of the static CPU state to find the next TB using a hash table. If the next TB has not been already translated, then a new translation is launched. Otherwise, a jump to the next TB is done.

In order to accelerate the most common case where the new simulated PC is known (for example after a conditional jump), QEMU can patch a TB so that it jumps directly to the next one.

The most portable code uses an indirect jump. On some hosts (such as x86 or PowerPC), a branch instruction is directly patched so that the block chaining has no overhead.

3.5 Memory management

For system emulation, QEMU uses the mmap() system call to emulate the target MMU. It works as long as the emulated OS does not use an area reserved by the host OS.[2]

In order to be able to launch any OS, QEMU also supports a *software MMU*. In that mode, the MMU virtual to physical address translation is done at every memory access. QEMU uses an address translation cache to speed up the translation.

To avoid flushing the translated code each time the MMU mappings change, QEMU uses a physically indexed translation cache. It means that each TB is indexed with its physical address.

When MMU mappings change, the chaining of the TBs is reset (i.e. a TB can no longer jump directly to another one) because the physical address of the jump targets may change.

3.6 Self-modifying code and translated code invalidation

On most CPUs, self-modifying code is easy to handle because a specific code cache invalidation instruction is executed to signal that code has been modified. It suffices to invalidate the corresponding translated code.

However on CPUs such as the x86, where no instruction cache invalidation is signaled by the application when code is modified, self-modifying code is a special challenge .[3]

When translated code is generated for a TB, the corresponding host page is write protected if it is not already read-only. If a write access is made to the page, then QEMU invalidates all the translated code in it and reenables write accesses to it.

Correct translated code invalidation is done efficiently by maintaining a linked list of every translated block contained in a given page. Other linked lists are also maintained to undo direct block chaining.

When using a software MMU, the code invalidation is more efficient: if a given code page is invalidated too often because of write accesses, then a bitmap representing all the code inside the page is built. Every store into that page checks the bitmap to see if the code really needs to be invalidated. It avoids invalidating the code when only data is modified in the page.

3.7 Exception support

longjmp() is used to jump to the exception handling code when an exception such as division by zero is encountered. When not using the software MMU, host signal handlers are used to catch the invalid memory accesses.

QEMU supports precise exceptions in the sense that it is always able to retrieve the exact target CPU state at the time the exception occurred.[4] Nothing has to be done for most of the target CPU state because it is explicitly stored and modified by the translated code. The target CPU state S which is not explicitly stored (for example the current Program Counter) is retrieved by re-translating the TB where the exception occurred in a mode where S is recorded before each translated target instruction. The host program counter where the exception was raised is

used to find the corresponding target instruction and the state S.

3.8 Hardware interrupts

In order to be faster, QEMU does not check at every TB if an hardware interrupt is pending. Instead, the user must asynchronously call a specific function to tell that an interrupt is pending. This function resets the chaining of the currently executing TB. It ensures that the execution will return soon in the main loop of the CPU emulator. Then the main loop tests if an interrupt is pending and handles it.

3.9 User mode emulation

QEMU supports user mode emulation in order to run a Linux process compiled for one target CPU on another CPU.

At the CPU level, user mode emulation is just a subset of the full system emulation. No MMU simulation is done because QEMU supposes the user memory mappings are handled by the host OS. QEMU includes a generic Linux system call converter to handle endianness issues and 32/64 bit conversions. Because QEMU supports exceptions, it emulates the target signals exactly. Each target thread is run in one host thread[5].

4 Porting work

In order to port QEMU to a new host CPU, the following must be done:

- `dyngen` must be ported (see section 2.2).
- The temporary variables used by the micro operations may be mapped to host specific registers in order to optimize performance.
- Most host CPUs need specific instructions in order to maintain coherency between the instruction cache and the memory.
- If direct block chaining is implemented with patched branch instructions, some specific assembly macros must be provided.

The overall porting complexity of QEMU is estimated to be the same as the one of a dynamic linker.

5 Performance

In order to measure the overhead due to emulation, we compared the performance of the BYTEmark benchmark for Linux [7] on a x86 host in native mode, and then under the x86 target user mode emulation.

User mode QEMU (version 0.4.2) was measured to be about 4 times slower than native code on integer code. On floating point code, it is 10 times slower. This can be understood as a result of the lack of the x86 FPU stack pointer in the static CPU state. In full system emulation, the cost of the software MMU induces a slowdown of a factor of 2.

In full system emulation, QEMU is approximately 30 times faster than Bochs [4].

User mode QEMU is 1.2 times faster than `valgrind --skin=none` version 1.9.6 [6], a hand coded x86 to x86 dynamic translator normally used to debug programs. The `--skin=none` option ensures that Valgrind does not generate debug code.

6 Conclusion and Future Work

QEMU has reached the point where it is usable in everyday work, in particular for the emulation of commercial x86 OSes such as Windows. The PowerPC target is close to launch Mac OS X and the Sparc one begins to launch Linux. No other dynamic translator to date has supported so many targets on so many hosts, mainly because the porting complexity was underestimated. The QEMU approach seems a good compromise between performance and complexity.

The following points still need to be addressed in the future:

- Porting: QEMU is well supported on PowerPC and x86 hosts. The other ports on Sparc, Alpha, ARM and MIPS need to be polished. QEMU also depends very much on the exact GCC version used to compile the micro operations definitions.
- Full system emulation: ARM and MIPS targets need to be added.
- Performance: the software MMU performance can be increased. Some critical micro operations can also be hand coded in assembler without much modifications in the current translation framework. The CPU main loop can also be hand coded in assembler.
- Virtualization: when the host and target are the same, it is possible to run most of the code as is. The simplest implementation is to emulate the target kernel code as usual but to run the target user code as is.
- Debugging: cache simulation and cycle counters could be added to make a debugger as in SIMICS [3].

7 Availability

QEMU is available at

`http://bellard.org/qemu`

References

[1] Ian Piumarta, Fabio Riccardi, Optimizing direct threaded code by selective inlining, Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[2] Mark Probst, Fast Machine-Adaptable Dynamic binary Translation, Workshop on Binary Translation 2001.

[3] Peter S. Magnusson et al., SimICS/sun4m: A Virtual Workstation, Usenix Annual Technical Conference, June 15-18, 1998.

[4] Kevin Lawton et al., the Bochs IA-32 Emulator Project, `http://bochs.sourceforge.net`.

[5] The Free Software Foundation, the GNU Compiler Collection, `http://gcc.gnu.org`.

[6] Julian Seward et al., Valgrind, an open-source memory debugger for x86-GNU/Linux, `http://valgrind.kde.org/`.

[7] The BYTEmark benchmark program, BYTE Magazine, Linux version available at `http://www.tux.org/~mayer/linux/bmark.html`.

Notes

[1] *x86* CPUs refer to processors compatible with the Intel 80386 processor.

[2] This mode is now deprecated because it needs a patched target OS and because the target OS can access to the host QEMU address space.

[3] For simplicity, QEMU will in fact implement this behavior of ignoring the code cache invalidation instructions for all supported CPUs.

[4] In the x86 case, the virtual CPU cannot retrieve the exact `eflags` register because in some cases it is not computed because of condition code optimizations. This is not a major concern because the emulated code can still be restarted at the same point in any cases.

[5] At the time of writing, QEMU's support for threading is considered to be immature due to locking issues within its CPU core emulation.