PQEMU: A Parallel System Emulator Based on QEMU

Jiun-Hung Ding\(^1\), Po-Chun Chang\(^2\), Wei-Chung Hsu\(^2\), Yeh-Ching Chung\(^1\),

\(^1\) Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan, ROC
adjunhon@sslab.cs.nthu.edu.tw
ychung@cs.nthu.edu.tw

\(^2\) Department of Computer Science, National Chiao Tung University, Hsinchu 30010, Taiwan, ROC
pochang0403@gmail.com
hsu@cs.nctu.edu.tw

Abstract. In this paper, we propose a novel design of a multi-threaded QEMU, called PQEMU, which can effectively deploy multiple emulated virtual CPUs on the underlying multi-core machine. The main idea of the design is to add a parallel emulation model to the execution flow of QEMU. To evaluate the design, we emulate an ARM11 MPCore by running PQEMU on a quad-core x86 i7 system and use SPLASH-2 as benchmarks. The experimental results show that the performance of PQEMU is, on average, 3.8 times faster than that of QEMU and is scalable on the quad-core i7 system for the SPLASH-2 benchmark suite.

1 Introduction

QEMU is a fast functional system emulator using the Dynamic Binary Translation (DBT) technique. With QEMU, single-core or multi-core computer systems can be emulated to run unmodified operating systems and application programs. However, the current QEMU is not designed for emulating multi-core systems. Since it runs multiple virtual CPUs in a round-robin fashion by using only one emulation thread, QEMU does not take advantage of the parallelism available in the underlying hardware. It incurs linear slowdown when the number of emulated cores is increased.

For computer architecture research, it is critical to have a full system emulator, such as QEMU, parallelized so that emulations of future multi-core architectures can actually run in parallel on the multi-core processors of current host machines. We have designed and implemented a parallel system emulator, called PQEMU, on top of the current QEMU. Although the implementation specifics are tuned for QEMU, the concepts of our work could be applied to other DBT based functional emulators.

The rest of this paper is organized as follows: Section 2 reviews the serial emulation model of QEMU. Section 3 describes the design of PQEMU. Benchmark results for the Serial QEMU and PQEMU are presented in Section 4.
2 The serial emulation model of QEMU

The current QEMU implementation has a severe performance limiter for emulating multi-core machines. Figure 1 presents a computer system emulated by the current QEMU including one emulation thread and one I/O thread. The emulation thread is composed of CPU, Memory and I/O functional models of one guest machine. No matter how many VCPUs (Virtual CPU) are created, only one emulation thread is activated to run all VCPUs in a round-robin fashion. Based on the serial emulation model, all emulated memory operations are considered exclusive and atomic. Hence the current QEMU emulates the hardware atomic operations of guest machines with locks. The I/O thread handles human interface devices (HID) and timer alarm from the host operating system. The host system events will update the status of guest I/O devices and trigger interrupts. The emulation thread can be interrupted by the I/O thread for signal delivery. Such a serial emulation model is simple and can be effectively applied to single-core emulations. However, such a model is inadequate for emulating multi-core systems.

![Fig. 1. A computer system emulated by QEMU](image)

3 The design of PQEMU

To parallelize QEMU, we designed a parallel emulation model so that each emulated VCPU can be deployed on a separate core in the underlying multi-core machine for effective parallel execution. Each VCPU is 1-to-1 mapped to an emulation thread and is scheduled by the host OS independently. If the host machine has multiple cores and the number of cores is greater than or equal to the number of VCPUs, PQEMU can boost its emulation speed and more accurately emulate the characteristics of concurrent execution in multi-core systems. We will briefly explain the parallel emulation model in three dimensions: CPU, Memory and I/O.

For emulating multiple VCPUs, the main challenge of PQEMU is to parallelize the DBT engine in the current QEMU. The original DBT engine must acquire a big lock to serialize the
emulation of multiple VCPUs. PQEMU is designed to deploy fine-grained locks in the DBT engine to exploit parallelism available in the host machines and achieve high emulation efficiency. It is complex and error-prone to deploy fine-grained locks, we have carefully analyzed the global common resources used in the emulation flow in QEMU and proposed a DBT event synchronization model for two code cache designs: one is a unified code cache (UCC) design and the other is a separated code cache (SCC) design. In the UCC design, the global common resources are shared by all VCPU threads, while in the SCC design, the common resources are private to each VCPU thread. The SCC design prevents most DBT operations from the need of synchronization and thus minimizes synchronization overhead. However, the SCC design requires much more memory for private code caches. Otherwise, the UCC design has only one copy of the code cache, but requires more synchronization overhead for various DBT operations. If PQEMU needs to emulate a large number of VCPUs, the SCC design might be less desirable in terms of the memory overhead.

For emulating the shared memory for multiple VCPUs, PQEMU must handle the translation of guest architecture’s atomic memory instructions correctly and efficiently. PQEMU has implemented two atomic instruction groups based on the *swap* instruction and an exclusive mechanism based on *ldrex* and *strex* (which are similar to the load-link and store-conditional pairs on the MIPS ISA) defined in the ARM ISA. The *swap* instruction is similar to the exchange instruction *xchg* in the x86 ISA, except that *swap* can work on two distinct registers (one source, and one destination register). This design excludes the possibility of using *xchg* to directly emulate *swap*, unless both the destination and the source registers are identical. PQEMU realizes the *swap* instruction by an explicit spin_lock, and has implemented a global monitor to support the *ldrex* and *strex* based exclusion mechanism.

The guest peripheral I/Os are emulated in QEMU by function callouts. A device callout is invoked once a VCPU raises an I/O request via MMIO read/write or IN/OUT instructions. A real I/O request is usually completed and uses an interrupt to inform the completion to the OS. In QEMU, an emulated I/O request is finished before the execution of the next guest instruction. Currently, PQEMU inherits this sequential I/O model from QEMU, based on an assumption that accessing guest MMIO registers by the guest OS is always exclusive, i.e. real I/O requests only appear in the kernel critical sections. Though modern kernels already follow this design convention, PQEMU still suffers when dealing with long-latency I/O requests such as DMA operations or disk accesses. To exploit more parallelism with I/O, we suggest a dedicated host thread other than using VCPU threads to carry out these time-consuming operations. This suggestion is not to replace the original I/O model, but to complement it.

The implementation of PQEMU has been instrumental in identifying deficiencies in guest device emulation. For example, the ARM PL011 UART in PQEMU uses the same emulation callout as in QEMU, which has no Tx FIFO mechanism due to a simplified implementation. The simplified implementation works well in the original round-robin VCPU
emulation environment; but in PQEMU, Tx IRQ could overwhelm the guest interrupt handling routine if all VCPU cores are generating data simultaneously via UART. In such a case, the guest Linux will be forced to shutdown this spurious UART IRQ, and caused the guest console to freeze.

4 Experimental Results and Conclusion

We have used PQEMU to emulate an ARM11 MPcore (with four cores) [1] on a quad-core Intel i7 based host machine. When running the parallelized SPLASH-2 [2] benchmark on the emulated MPcore, PQEMU is 3.8x, on average, faster than the original QEMU. To further understand the performance and the possible limitation to scalability of PQEMU, we have built multiple internal profiling tools to examine the synchronization overheads of the UCC and the SCC designs. For the research community, it is very useful to have such a parallelized full system emulator, such as PQEMU, so that the emulation for future multi-core architectures can run concurrently on the current multi-core computers. PQEMU opens up new opportunities to more efficiently study the issues of parallel applications and system software for future multi-core systems.

![Graph showing benchmark speedup of PQEMU (UCC and SCC) compared to serial QEMU (S-QEMU)](image)

Fig. 2. Splash-2 benchmark speedup of PQEMU (UCC and SCC) compared to serial QEMU (S-QEMU)

References