Towards a Fully Multithreading Support for QEMU

Tzu-Han Hung, Alex Wauck

Abstract

Multicore computers are the most dominating computer architectures recently. Therefore, for the research field on virtualization technology, it is becoming more and more important to efficiently emulate/virtualize a multicore system on a multicore system. Among many virtualization software tools, QEMU is a binary translation-based emulator and thus enjoys the flexibility of emulating a processor instruction set architecture (ISA) on a host of a different processor ISA. However, QEMU has a poor efficiency of emulating a target multicore system on top of a host multicore system, and this greatly limits the applicability of QEMU. The main reason of the prohibitive emulation performance is because of the single-threaded nature in QEMU implementation, which dates back to 2005, when multicore systems were not as prevalent as today.

In this work, we attempt to add multithreading support into QEMU codebase so that the emulation engine can benefit from using multiple native threads and the emulation time can thus be greatly reduced. We choose to exploit the parallelism at the the parallelism at the bottom-most level of the emulated computer system, the virtual processors. This strategy allows us to take advantage of the inherent parallelism that resides in the hardware, no matter which forms of parallelism (multiple processes or threads) are used by applications running in the emulated environment.

1 Introduction

Multi-core architectures are becoming increasinly common, and this makes it more common and important to emulate a multicore system on a multicore system.

QEMU [6] is a popular processor/system emulator, and yet it does not fully exploit the computation power of the host multicore machine, as it is does not fully support multithreaded execution and emulation. Our goal is to make QEMU not only emulate for target multicore systems, but also powered by host multicore systems.

QEMU is a binary translation-based emulator and works in two different modes: user or system mode. Under user mode, QEMU directly faces to a program to be emulated and treats it as user-level process. This user program cannot spawn threads, as the concept of *threads* does not exist in user-mode emulation and QEMU therefore does not support the use of threading libraries (such as *PTHREAD*) for user-mode emulation. Under system-mode emulation, QEMU is capable of emulating the behavior of an modified operating system, chosen by the user, and together with the operations performed by the applications that run on top of the installed operating system.

We believe that the system-mode emulation is more relevant and interesting. First, it provides the user an environment for testing and evaluating not only an application but also the computer system as a whole. This is the approach other emulators and virtual machines also take and is likely to be the most prevalent use case for virtualization technology. Moreover, the system-mode emulation allows multi-tasking, either through the form of multiple processes or threads, and introduces the interactions and communications between different tasks.

From its introduction back in 2003, QEMU supported only the emulation of a uniprocessor system [8]. With the increasing popularity of multicore systems, QEMU started to support the emulation of a complete multicore system in the recently years. Specifically, the current QEMU implementation can emulate the environment of a shared memory multiprocessor (e.g., an Intel Core 2 Duo), SMP, and a SMP-enabled operating system running on top of it (e.g., an SMP-enabled Linux kernel).

However, in spite of the capability of emulating a target multicore system, the emulation efficiency is

extremely poor and is sometimes prohibitive. For example, it is not uncommon that emulating a dual-core target machine, even with the same single process running in the operating system, introduces a 1.5x – 2x slowdown than emulating a single-core target machine.

The main reason for this inefficiency dates back to the original design goal of QEMU: emulating a target uniprocessor system on a host uniprocessor system. With this goal, QEMU implementation is designed to be mostly single-threaded: since there is one processor in the target system, only one active task needs to be emulated at a time; since there is one processor in the host system, only one job can be run by the underlying hardware at a time.

Until version 0.11, QEMU abided by this design goal and had no multithreading support at all in the codebase. In 0.11 version, QEMU introduced the use of a separate I/O thread that handles the usually time-consuming and sometimes decoupled I/O tasks [6]. However, the execution of CPU operations is not parallelized and is handled by only one thread. That is, no matter how many cores are on the host machine, QEMU will spawn two threads at most, one for handling the I/O tasks and the other for handling the operations of all virtual CPUs. It is obvious that the latter thread can easily be the bottleneck for system emulation, especially there are several CPU-intensive jobs in the guest operating system.

Our work attempts to add the true multithreading support into QEMU so that the emulation engine can benefit from using multiple threads and the emulation time can thus be greatly reduced.

The rest of the paper is organized as follows. Section 2 explains our design and its merits. Section 3 describes the implementation details of both the original QEMU codebase and our modification. Section 4 presents some experimental results and annlyses. Finally, section 5 concludes our work.

2 Design

This sections contrasts our design with several alternative approaches. It first introduces two different ways for enhancing the emulation speed and points out the drawbacks of these techniques. It then describes our approach and discusses several associated benefits other than the improved emulation performance.

2.1 Alternative Approaches

There are some alternatives for speeding up the QEMU emulation. One naive approach is to launch several instances of the single-threaded QEMU on a multicore host platform with the hope that all the real cores can be kept busy at the same time. This approach can be effective when the multiple guest applications are to be run are completely independently and do not need to communicate with each other (at least not at high speed). However, since there are multiple instances of the operating system and its components (e.g., file system) being created and updated, the semantics of the side-effects (e.g., the guest applications all create some files and write back to the file systems) are not easy to reason about. QEMU uses an image file to store the state of an emulated system. As of now, QEMU does not provide a tool or a procedure to merge multiple image files, with appropriate *merge* semantics considered. Furthermore, memory usage is increased substantially, as the OS kernel and resources must be duplicated, and QEMU cannot support merging of identical pages across multiple instances.

Another solution, as is deployed in QEMU version 0.11, is to use KVM [3] to do true virtualization instead of emulation. For each virtual CPU, QEMU allocates an independent KVM virtual CPU. The KVM-based emulation does not rely on QEMU's dynamic binary translation mechanism and therefore very little of the codebase is shared. Signals transmitted between different virtual CPUs are emulated using internal communication mechanisms. The major limitation of this scheme is that KVM requires the instruction set architecture (ISA) of the target and host processors to be the same. Furthermore, when virtualizing x86, KVM requires that the host CPU support AMD or Intel virtualization extensions. This avoids the use of dynamic binary translation techniques and offers a reasonable emulation speed, but it also imposes a serious constraint on applicability. In particular, one increasingly popular application of QEMU, emulating ARM systems on x86 machines (used in Google's Android emulation toolkit [1]), cannot benefit from this strategy.

2.2 Our Approach

We take a different approach by incorporating true multithreading support into QEMU, while still using dynamic binary translation to maintain the flexibility of target and host processor ISAs. From a high level point of view, we create a separate thread for each virtual CPU. (Of course, we recommend that the number of virtual CPUs be less than or equal to the number of real CPUs). Once the threads are initialized (with a *thread function* that contains necessary information for directing the execution of a single core), they can emulate the execution of all virtual CPUs concurrently, with minimal synchronization points.

Our strategy exploits the parallelism at the bottommost level of the emulated computer system: the hardware. One major advantage of this approach is that hardware is inherently parallel. On a multicore system, the execution of the cores are concurrent by nature. This means our approach takes advantage of the universal parallelism and does not depend on other types of parallelism introduced in the operating systems or user-level applications. Another attraction is that no matter what types of parallelism the emulated application is using (e.g., by creating multiple processes or multiple threads), our approach can also take advantage of the user-introduced parallelism, as the SMP-enabled operating system kernel will eventually map the multiple tasks (in the form of processes or threads) down to the multiple virtual CPUs, and our version of QEMU will be able to execute them concurrently with several native threads on the host machine.

Moreover, since our scheme does not prohibit the use of QEMU's dynamic binary translation, it still supports a great range of choices of different target and host processor types and thus covers the recently popular application mentioned beforehand: using ARM as target ISA and X86 as host ISA.

3 Implementation

This section first provides some background information for the QEMU implementation and then turns to the discussion of our code changes for incorporating multithreading support into the QEMU codebase.

3.1 QEMU Introduction

When QEMU is operating with the dynamic binary translation enabled (in contrast to KVM mode), it translates at runtime the target code that has never been seen and translated into host code up to the next block-terminating instruction (e.g., a jump or a CPU state-modifying instruction). In QEMU, this region of code is called a *translation block* and is the basic unit for its binary translation engine, the *tiny code generator* (TCG) [7]. To reuse the translated code for emulation efficiency, QEMU allocates a space, called the *translation buffer*, and puts the newly generated code for the translation block in the buffer. It is reported that a 16 MByte cache is sufficient for holding the most recently used translation blocks. Besides, for the simplicity of implementation, the translation buffer is completely flushed when it is full [8]. Some research aims to improve the code cache utilization and thus the dynamic translation performance [10], but it is out of the scope of our project, although it can be a significant direction for our future work.

3.2 Implementation Scheme

This sub-section explains how we use multiple threads to execute the operations for different virtual processors. Note that in QEMU 0.11, there is already some primitive support for multithreading: QEMU will create two threads at most and have them work for CPU operations and I/O jobs, respectively. In our code, we extend the use of threads by allocating more threads for each virtual processor and by eliminating lots of the shared variables and records in the code.

Figure 1 shows the control path for native code generation and execution, which is essential to the QEMU dynamic binary translation mechanism. When a QEMU emulator is booted, it initializes all the virtual CPUs. In our case, multiple x86 cores are selected as the target processors and hence `cpu_x86_init()` is called for processor initialization. It in turns calls `qemu_init_vcpu()`, `tcg_init_vcpu()`, and then spawns a native thread for this virtual processor by passing a function pointer `tcg_cpu_thread_fn` to `qemu_thread_create()`, which then invokes `pthread_create()` with the given function pointer.

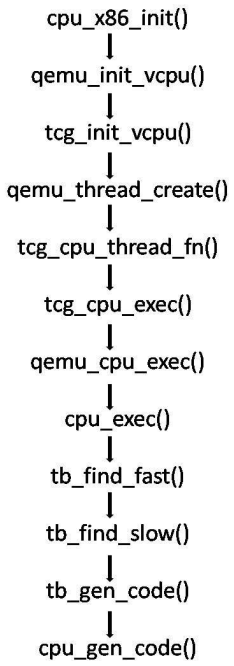

Figure 1: Control path for native code generation and execution.

Each *thread function* `tcg_cpu_thread_fn()` stands for an independent emulation job for a single processor. It invokes a series of functions in turn to get to the native code generation and execution: `tcg_cpu_exec()`, `qemu_cpu_exec()`, `cpu_exec()`. In `cpu_exec()`, it tries to determine if the the code to be executed has been translated before by invoking `tb_find_fast()`. QEMU maintains a cache for the translation block lookup and `tb_find_fast()` depends on this feature to perform a fast lookup. If the fast lookup fails, possibly because of cache miss, `tb_find_slow()` is used for a thorough lookup into the translation buffer. If the slow lookup also fails, the binary translation engine will be invoked and native code will be generated through the calls to `tb_gen_code()` and `cpu_gen_code()`.

If the fast or the slow lookup is successful, or if both lookups fail and the new code is thus generated, a pointer that points to the corresponding native code will eventually be returned to the calling function `cpu_exec()` and the caller will set the native program counter to the starting address of the generated code and start the native execution.

3.3 Implementation Details

Due to time constraints, an actual implementation was not fully achieved. However, an idea was formed. As a first step, we believe that a performance gain could be had by ensuring that every virtual CPU has its own context, complete with its own list of translated code blocks. While this leads to redundant translation when multiple VCPUs execute the same code, we believe that the extra translation will not take enough time to outweigh the benefits of running multiple guest processes/threads concurrently on multiple physical CPUs, at least when ordinary static code (such as a plain C program) is executed.

To further improve performance, we could keep the *translation block list* shared, but make everything else per-VCPU. That is, each VCPU's register file could be modfied at the same time instead of shuffling register values in and out of a shared register file. This way, the global mutex would not need to be held during `tcg_cpu_exec()`. Since that mutex is also used to mediate interactions between the main thread during I/O operations, the main thread would use a separate lock for each VCPU so that when it needs to modify a VCPU's memory for I/O operations, it does not block the other VCPUs unnecesarily. Thus, the only factor limiting concurrency between VCPUs would be the spinlock in `cpu_exec()`, which is only held while checking the translation block list for translated code, translating the guest code, and updating the translation block list. This is already the case in QEMU, so this method should not cause any noticable slowdown for the single-VCPU case.

3.4 Implementation Challenges

The QEMU code could easily be used as a case study for how not to write C code. It does several things that are typically regarded as bad practice, like using global variables and defining macros that use those global variables implicitly. It also abuses the C preprocessor to a truly perverse degree, making it excessively difficult to find the definitions of many commonly-used functions. In fact, the code is so twisted and horrifying that we find it plausible that it was deliberately written this way to frustrate anyone not intimately familiar with the codebase. The fact that the KQEMU accelerator module [2] was closed-

source until KVM superceded it and Mr. Bellard's two victories in the International Obfuscated C Code Contest add weight to this hypothesis. Naturally, the translation code is sparsely commented as well.

On a related note, locks are used in a manner that is difficult to trace, which makes the exact purpose of the global lock unclear. It is clear that it is used in the main thread so that VCPUs are not running during I/O operations, but it is not clear if it actually serializes access to the global VCPU state variable.

Another factor that played against us was our decision to perform the implementation with x86 as the guest architecture. This decision was made due to the availability of pre-made x86 Linux disk images. However, this proved to be a mistake, as the amount of code for translating from guest machine code to the TCG intermediate representation is directly proportional to the number of different instructions in the ISA. Since the global VCPU state variable is used extensively in this part of the QEMU code, this ended up taking an inordinate amount of time to modify. Had we chosen a RISC architecture as the target instead, we may have had enough time to finish our implementation.

4 Evaluation

This section provides the preliminary results on the emulation time.

To evaluate the emulation time, we use a parallel program (implemented with OpenMP [5]), Needleman-Wunsh sequence alignment [9], in our experiments. Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments [4]. The parallel implementation of this method creates multiple threads (via OpenMP) for its array-based computation. In our experiments, we tried two different array sizes: 3200 * 3200 and 6400 * 6400. The machine we used in the experiments has an Intel Core 2 Duo processor, which contains two physical cores, sharing the memory and the L2 cache.

Figure 2 shows the *wall-clock running time* of the whole-system, whole-program emulation for three different emulator configurations: the blue bars represent the original QEMU implementation with only one virtual processor; the red bars represent the original QEMU implementation with two virtual processors; the green bars represent our modified QEMU with two virtual processors. The red bars are in general higher than the blue bars, indicating the inefficiency of emulating a multicore system with the current QEMU implementation. Note that the application itself spawns multiple threads (via OpenMP) and the job for each worker thread is essentially halved and therefore the total amount of work remains roughly the same, and yet the SMP emulation in QEMU introduces some overhead and makes the execution time longer.

Unfortunately, our current implementation provides the longest emulation time. Currently, two native threads are created and run simultaneously, but never at peak CPU utilization, even though the application being emulated is obviously CPU-intensive. The main reason for this is there are still too many data structures being shared and there is some non-trivial code region being put into the critical section (i.e., protected by mutexes). The result is that a large part of the code ends up running sequentially and so although two native threads are used, they can only utilize about half of the (physical) CPU time each. The first step of our future work is to further reduce the critical section code and thus expose and exploit more parallelism.

5 Conclusion

While the current QEMU desgin permits emulation of multiple CPUs, it does not make effective use of multiple physical CPUs. By changing the code to keep track of VCPU state separately for each VCPU, we should be able to substantially decrease the amount of code protected by the global mutex. By then running a separate translation thread for each VCPU, we can then take advantage of this change to effectively utilize multiple physical CPUs to emulate multiple virtual CPUs. While this is a worthy goal, it has yet to become a reality, largely due to the large amount of code that needs to be changed and the complicated nature of the code involved.

References

[1] Google Android SDK. http://developer.android.com/sdk/index.html.

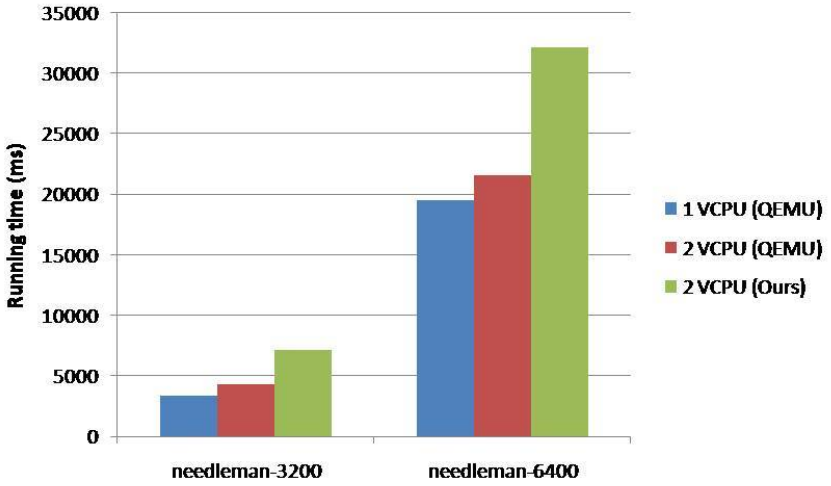

Figure 2: Results on emulation time.

[2] KQEMU: QEMU Accelerator. http://www.qemu.org/kqemu-doc.html.

[3] KVM: Kernel-based Virtual Machine. http://www.linux-kvm.org.

[4] Needleman-Wunsch Method. https://www.cs.virginia.edu/ skadron/wiki/rodinia/index.php/Needleman-Wunsch.

[5] OpenMP. http://openmp.org/wp.

[6] QEMU: Open Source Processor Emulator. http://www.qemu.org.

[7] TCG: Tiny Code Generator. http://svn.savannah.gnu.org/svn/qemu/trunk/tcg/README.

[8] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2008.05.014.

[10] K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 89, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.