2011 IEEE 17th International Conference on Parallel and Distributed Systems

PQEMU: A Parallel System Emulator Based on QEMU

Jiun-Hung Ding
MediaTek-NTHU Joint Lab
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan, ROC
adjunhon@sslab.cs.nthu.edu.tw

Po-Chun Chang[1], Wei-Chung Hsu[2]
MediaTek-NTHU Joint Lab
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan, ROC
Pochang0403@gmail.com[1]
hsu@cs.nctu.edu.tw[2]

Yeh-Ching Chung
MediaTek-NTHU Joint Lab
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan, ROC
ychung@cs.nthu.edu.tw

***Abstract*—A full system emulator, such as QEMU, can provide a versatile virtual platform for software development. However, most current system simulators do not have sufficient support for multi-processor emulations to effectively utilize the underlying parallelism presented by today's multi-core processors. In this paper, we focus on parallelizing a system emulator and implement a prototype parallel emulator based on the widely used QEMU. Using this parallel QEMU, emulating an ARM11MPCore platform on a quad-core Intel i7 machine with the SPLASH-2 benchmarks, we have achieved 3.8x speedup over the original QEMU design. We have also evaluated and compared the performance impact of two different parallelization strategies, one with minimum sharing among emulated CPU, and one with maximum sharing.**

Keywords – Multi-core; Emulator; Parallel; Sychronization

I. INTRODUCTION

A full system emulator allows entire software stack running without code modification. It is commonly employed in OS and application development before target hardware is available. Many full system emulators are available today, such as Simics [12], SimOS [11], Embra [15], Bochs [8] and QEMU [6], and most of them adopt Dynamic Binary Translation (DBT) techniques [3] to achieve high emulation speed. Although DBT is effective in increasing emulation speed under single-thread execution environment, it does present a challenge for emulating multi-threaded execution because the DBT engine must be parallelized first. Due to the complexity of parallelizing the DBT engine, many system emulators choose to emulate multi-threaded guest applications sequentially in a round-robin fashion. This common approach fails to take advantage of parallelism existed in the guest multi-threaded application, and parallelism available in the underlying host hardware. In this paper, we discuss the design and implementation of a parallelized QEMU, called PQEMU.

Parallelizing such a system emulator is challenging because both concurrent code generation (i.e. parallelizing the DBT engine) and parallel code execution (i.e. managing thread execution in the code cache) are important. In a parallel system emulator, each guest core can be represented by a host emulation thread, executing dynamically translated codes from guest threads in the code cache independently. However, there are dependences among those seemingly independent emulation threads, which must be handled correctly. For example, any modification to the guest code (as in self-modifying programs) would require a serialization to those emulation threads since the dynamically translated code might be modified. SMC (Self-Modifying Code) may seem like unusual events for application programs. However, the increasingly popular use of JIT techniques in high-level language virtual machines makes SMC more common. Furthermore, for system emulations, SMC happens more often when the guest OS reclaims memory pages (reuse pages containing binary code). The original synchronizations among parallel threads from the guest applications must also be handled correctly in the dynamically translated code. For example, these atomic instructions in the guest binary must be translated into host binary with identical behavior to ensure correctness of emulation.

To increase the parallelism of the DBT engine, it seems straightforward to minimize resource sharing between emulated guest CPU cores. For example, the code cache that stores dynamically translated codes could be separated. This separate code cache design (SCC) can minimize synchronization needs because when one guest core is translating its current guest code, the other does not need to wait for the completion. When two are using the same guest code, it is acceptable or even desirable to have two copies of translated code in their separate code caches. However, this separate code cache design has its own downsides, as it may incur more code translations as well as increased memory requirement for code cache, especially when emulating many guest cores. To further understand the tradeoff between translation overhead, resource utilization and synchronization overhead, our PQEMU prototype explores two alternative implementation strategies, one for the unified code cache design (UCC), which allows all guest cores to share a common code cache, and one for the separate code cache design (SCC), which allows each guest core to have its own code cache. Notice that if the guest application is a parallel program, such as the SPLASH-2 [13] benchmark, they are likely to share a large portion of the code. However,

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.102

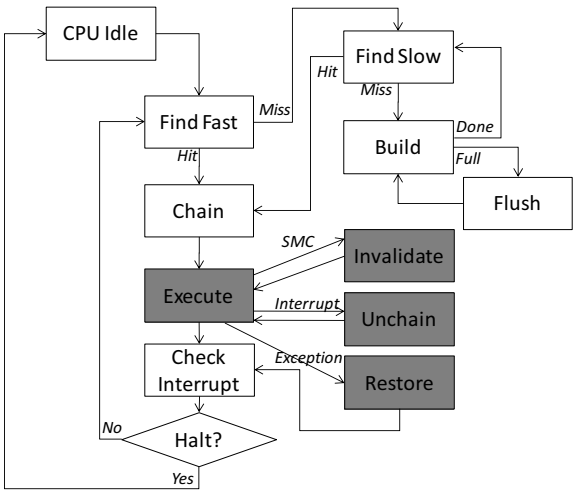

Figure 1. Typical flow of a full system emulator using dynamic binary translation.

if the guest workload is composed of many different jobs, the code sharing will be minimal.

Using the SPLASH-2 benchmark as the guest workload, our PQEMU prototype on average performs 3.8x faster than the original sequential QEMU when emulating a virtual ARM11MPCore [7] guest platform on a quad-core Intel i7 based system. This paper made the following contributions:

- It reports required work when paralleling a DBT-based system emulator, in terms of both code generation and execution phases.
- It implements two alternatives (SCC vs. UCC) to investigate the tradeoffs among translation overhead, memory resource utilization and synchronization overhead when parallelizing a DBT based system emulator.
- It shows both implementations can effectively utilize the parallelism existed in the guest application and the parallelism available in the host multi-core system. When emulating an ARM11MPCore on an Intel i7 quad-core based system, both PQEMU implementations can be 3.7-3.8X faster than the original QEMU using the SPLASH-2 benchmark.

The rest of this paper is organized as following: Section II describes the conventional design of a system emulator with DBT as the main acceleration technique, and challenges for parallelizing such designs. Section III provides the parallelization steps toward two alternatives (UCC and SCC designs) and their implementations in PQEMU. Section IV evaluates the performance of PQEMU variants with comprehensive discussion. Section V briefly discusses related work and section VI summaries and concludes.

II. BACKGROUND

For system emulators using DBT to increase emulation speed, the guest binary code must be first translated into equivalent host binary, in unit of basic block or trace. For QEMU, this unit is called a Translated Block (TB). Such TBs will be stored in the *Code Cache* to avoid repeated translations from the same guest binary code. Once the TB is ready, the emulation will be directed to execute the TB. At the end of TB execution, the emulator goes back to the emulation manager. This life cycle of emulation is illustrated in Fig. 1 (dark grey boxes are states in which the emulator executes in the code cache; while others are in the emulation manager). To reduce expensive transitions (architecture states must be saved and restored) between the native execution in the code cache and the emulation manager, the emulator *Chains* subsequently executed TBs to constitute a TB chain. Later code cache execution will be going through a series of TBs, not just one TB, until the chain breaks. The chain of TB will grow longer and longer. Eventually, all important guest codes are translated into TBs and get chained together. When this happens, the emulation will stay executing in the code cache, and rarely come back to the execution manager.

To reuse the codes in code cache, *Find Fast* and *Find Slow* will locate the target TB by the guest PC prior to the *Build* code generation phase. A pointer to the executed TB will be cached in a guest-core-private field to make best use of TB execution locality, and this field will be examined first in *Find Fast* before resorting to a slower but more complete search in *Find Slow*. Code generation occurs in *Build* after all TBs search attempts are failed, and *Flush* is called when the code cache overflows. In QEMU, *Flush* simply removes all translated TBs from the code cache.

Step into *Execute*, the emulator will execute (a chain of) TB in the code cache and make substantial emulation progress. It returns to the emulation manager after executing an unchained TB, or encountering a guest exception. The former results from guest interrupt delivery and Self Modifying Code (SMC) event. The arrival of a guest interrupt will trigger *Unchain* to allow the returning to the emulation manager for guest interrupt handling at *Check Interrupt*; while SMC takes place when guest core tries to modify the memory content which has codes already being translated in the code cache. All offending TBs will be erased in *Invalidate* by removing their indices in the guest-core-private field and a central hash table in *Find Slow* and the emulator leaves the code cache when the target TB is eliminated. The latter guest exception handling in *Restore* requires extra recovery to maintain precise architectural states before leaving the code cache, since exceptions could arise anywhere during TB execution (for example, a guest page fault may arise during the emulation of a guest memory instruction).

Pending guest interrupts are handled in *Check Interrupt* by resetting the guest program counter to a specific vector address, according to the source of guest interrupt. *Halt* and *CPU Idle* are designated for guest instructions waiting for a specific hardware event, such as the ARM *wfi* (wait-for-interrupt).

A. Extend to Emulating a Multi-core Machine

Functionally, emulating a multi-core machine would be as simple as duplicating all guest-core-private data structures to reflect every architectural state of guest multi-core, yet memory and I/O systems are still shared among all guest

cores to mimic the SMP architecture in the real world. This incurs the concurrency problem as guest cores might write the same memory location simultaneously. Conventional emulators adopt a time-sharing scheme to simplify the problem, such that the emulation of guest cores goes in a round-robin fashion, which turns the memory and I/O accesses exclusive to guest cores in emulation. This sequential emulation model also helps in design of I/O emulation (callout functions that perform guest I/O operations for the virtual platform) - no race-condition could possibly happen. To minimize code translation efforts, some DBT engines will generate more versatile code sequences that all TB accesses to guest architectural states go through indirect references, e.g. using base register plus displacement addressing mode.

The design works well in traditional uni-processor environment. However, running such emulators on today's multi-core system is inefficient, because all guest core emulations will be aggregated on a single emulation thread on a single host core, leaving all other cores idle as a thread is the smallest indivisible task unit in the host OS.

B. Toward Multi-core on Multi-core

To fully utilize the power of multi-cores in the host machine, the emulator must create multiple threads so that the host OS could schedule them on the host cores separately. An emulation thread is in the parallel emulator is equivalent to a guest core. Because the emulation spends most of time in code execution, emulation threads would be computation-intensive and distributed evenly on the host cores. The net effect is one guest core is simulated by one emulation thread and is scheduled to run on one host core, and multiple guest cores could be simulated concurrently as running on real hardware. If there are more guest cores than available host cores, we currently have no reliable way to emulate without distortion with respect to real execution, and hence it is not discussed in this paper.

Such multi-thread emulator designs improve not only emulation speed but also the real concurrent execution behavior. Because a guest multi-threaded program could exploit parallelism on real machines, more intrinsic characteristics about guest multi-threaded program could be observed by such parallel emulators, without turning to real hardware. That gives great flexibility to software development, especially when hardware is inaccessible or not available. However, parallel emulation would add complexity to guest I/O and memory access emulation, since they could be raised at the same time, and to the same location. This could incur race conditions if mutual exclusion is not enforced. Reverting to aforementioned sequential model would be the last choice, since memory instructions are very frequent in typical programs.

Inside the emulator, I/O access from guest cores will be redirected to the I/O emulation functions, which bridge host system calls to functionality of guest peripherals. For example, a common realization of virtual platform timer is a host alarm registered for emulation threads. Alarm is set by a timer period, and a guest interrupt is generated whenever the emulator receives an alarm signal from the host OS. In a multi-threaded emulator, reentrant is a must for I/O emulation function to support concurrent invocations from different guest cores. There will be no memory ordering issue for MMIO (Memory Mapped I/O) access inside the emulator, since calling the I/O emulation function is synchronous to the guest core emulation. Specifically, the I/O function is invoked right after the guest core executes a memory instruction within the MMIO address space. Unless the function call ends, emulation will not proceed to the next guest instruction. In effect, the memory ordering for MMIO accesses follows the guest program order exactly inside the emulator, without relying on guest memory serialization instructions.

For write accesses to the same memory location, the hardware arbitrator determines the order of write requests (and thus final content), which is completely invisible to software. To those software operations sensitive to write sequence, program will use *atomic instructions* instead of plain memory write to guarantee their effects, or at least know whether the write goes as intended (and redo the operation if not). An example for the former case is updating a shared counter via atomic add instruction, where race conditions might happen if implemented in typical read-modify-write instruction sequences; while the latter includes the implementation of a software lock, that all pending candidates tries until the lock is grabbed.

Parallel emulators must enforce atomicity guest program demands, or program will behave incorrectly. Consider the case guest atomic add instruction is translated to a series of read-modify-write host codes. When it is executed on parallel emulator without synchronization beforehand among emulation threads, race-condition could happen. To make best use of host hardware, parallel emulator will generate host atomic instructions for those guest atomic instructions. The difficulty lies in the diversity of semantic transformation between guest and host, because the atomic instructions are architecture-specific.

III. DESIGN AND IMPLEMENTATION

We attempt to describe a uni-core system emulator using DBT as a state machine, where state $S \in \{$*CPU Idle, Find Fast, Find Slow, Build, Flush, Chain, Execute, Invalidate, Unchain, Restore, Check Interrupt*$\}$, and transition function δ is illustrated in Fig. 1. For multi-core emulation, we use notation S_n and δ_n to specify the state and transition function of guest core *n*. In conventional round-robin designs, the emulation goes for each guest core sequentially, one at a time by a single emulation thread, and thus state S_i and S_j for guest core *i* and *j* are totally independent, even if they all use the same transition function δ, i.e. $\delta_i = \delta_j = \delta$.

However, since guest core *i* and *j* might access the same shared component of the emulator from different host cores in parallel emulation, S_i and S_j are partially dependent in states other than *CPU Idle* and *Check Interrupt*. By example of QEMU, shared components are shown in the middle oval shapes of Fig. 2, along with their relationships to emulation states that might access them concurrently. Detailed explanations about these shared components are described as follow:

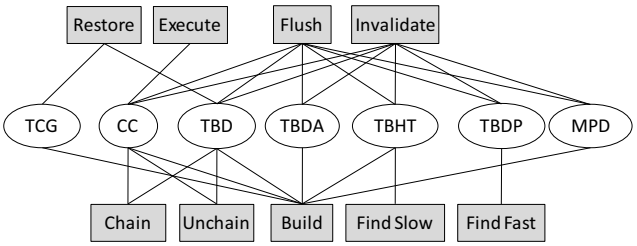

Figure 2. Sharing components (middle ovals) and emulation states that might access them concurrently (grey rectangles) in QEMU.

- *TCG translation engine* (**TCG**): it is the binary translation engine in system emulator, used by *Build* for new TB generation or Restore for guest architecture state recovery.
- *Code Cache* (**CC**): the storage space for TB output after *Build*, code translation phase. *Chain* and *Unchain* will patch the last branch instruction of a TB directly in code cache; while *Flush* and *Invalidate* erase one or more TB.
- *TB Descriptor* (**TBD**): it holds the meta-information of a TB in code cache, e.g. starting guest PC value of TB. It is an identification for TB and being initialized in *Build*, updated at *Chain* or *Unchain* (to the fields of chaining status), and reset in *Flush* or *Invalidate*.
- *TB Descriptor Array* (**TBDA**): to simplify the management of TB descriptors, array of descriptors will be pre- allocated during QEMU initialization phase. *Build* will consume one entry for the new TB. If no entry is available, *Flush* will be triggered to reclaim all descriptors, by dropping all TBs in code cache.
- *TB Hash Table* (**TBHT**): it is the central hash table in key of guest PC value that *Find Slow* searches after *Find Fast* fails. Every in-use *TBD* has an index in this hash table to reference to, and states modifying a *TBD* would update its index, correspondingly.
- *TB Descriptor Pointer* (**TBDP**): it is a field private to each guest core that holds the index (duplicated from previous hash table) to recently-used *TBD*. It speeds up the TB lookup for guest loop code, as *Find Fast* will check this field first before *Find Slow* searches the central hash table.
- *Memory Page Descriptor* (**MPD**): to accelerate the detection of guest SMC activity, emulator must efficiently find all offended TBs for every guest write that changes the guest code already being translated in code cache. QEMU uses this descriptor to record TBs having codes lying in the same guest page. Only the TBs in the same *MPD* will be check for possible SMC write. Again, Build inserts new TB to a descriptor; while *Invalidate* and *Flush* delete them.

To parallelize the QEMU system emulator, we deploy locks to serialize accesses to the shared components, see Fig. 2. We explore two alternative designs for the initial PQEMU implementations, one is to share the same code cache for all VCPUs and the other is to have a private code cache for each VCPU.

TABLE I. SYNCHRONIZATIONS BETWEEN TWO EMULATION STATES FOR UCC DESIGN IN TWO-THREAD PARALLEL EMULATION.

UCC	***N***	***S***	***B***	***R***	***C***	***U***	***E***	***F***	***I***
*Fi**N**d Fast*	I	I	I	I	I	I	I	S	S
*Find **S**low*	I	I	S	I	I	I	I	S	S
***B**uild*	I	S	S	S	D	D	D	S	S
***R**estore*	I	I	S	S	D	D	I	S	S
***C**hain*	I	I	D	D	S	S	D	S	S
***U**nchain*	I	I	D	D	S	S	D	S	S
***E**xecute*	I	I	D	I	D	D	I	S	S
***F**lush*	S	S	S	S	S	S	S	S	S
***I**nvalidate*	S	S	S	S	S	S	S	S	S

A. Unified Code Cache (UCC) Design

In this option, no sharing components are duplicated for minimum memory usage in PQEMU. To minimize serialization overhead, locks will be applied only if necessary. We consider the case using two threads for parallel emulation particularly, since this case could reduce to other configurations with more emulation threads. All synchronization requirements between state S_i and S_j for guest core *i* and *j* are tabulated in Table I , where emulation states are those that might touch the shared components of the parallel emulator. Possible conditions are ***I**ndependent*, ***D**ependent*, and ***S**ynchronous*, in order of synchronization strength required. For example, *Restore* and *Find Slow* are ***I**ndependent* because they never use the same shared component. On the contrary, *Restore* and *Build* are ***S**ynchronous* since *TCG* translation engine is shared among all emulation threads. ***D**ependent* signifies the combination that even though something is shared for state S_i and S_j by Fig. 2, no simultaneous access would happen in real life. For instance, *Build* shares *CC* and *TBD* components with *Chain*/*Unchain*/*Execute*, but they are intrinsically independent because a TB in translation will not be referenced since it is not created yet. The same reason applies to *Chain/Unchain* and *Execute*, assuming branch instruction patching (host memory write) is atomic on the host machine, i.e. no emulation threads would ever observe the branch instruction at the end of TB is partially updated.

To derive lock-applying rules for UCC parallel emulator design, we group emulation states in Fig. 1 as four independent sets:

- ***Construct*** = {*Find Fast*, *Find Slow*, *Build* and *Restore*}
- ***Link*** = {*Chain* and *Unchain*}
- ***Use*** = {*Execute*},
- ***Destruct*** = {*Flush* and *Invalidate*}

All synchronizations in Table I could now be generalized in the following rules for correct and efficient parallel emulation, no matter how many guest cores are being emulated:

- Any two states live in the same set must run sequentially, except those pure read operations like

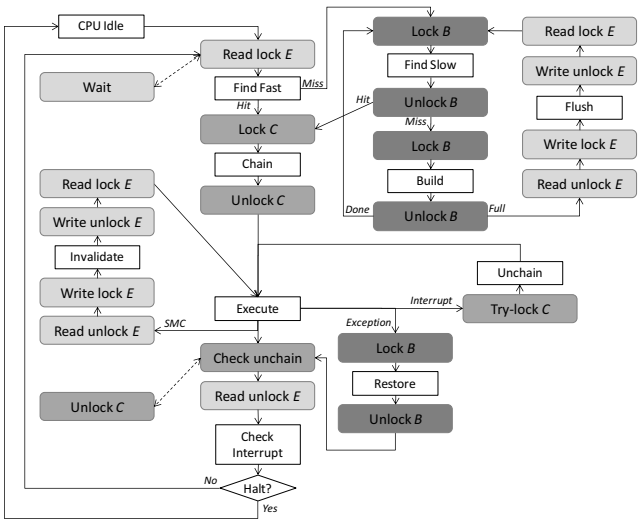

Figure 3. Modified emulation flow for PQEMU using UCC design. Lock C, and B (boxes in light, middle and dark grey) represent exclusive_rwlock chain_lock and build_lock locks, respectively.

Find Fast, *Find Slow* and *Execute*; otherwise they could go parallel.

- ***Destruct*** requires an exclusive access for efficiency reason, since the states will modify most of sharing components all at once.

We deploy locks only at state combinations in ***S****ynchronous*, where the modified emulation flow for UCC parallel emulator is shown in Fig. 3. We introduce ***e****xclusive_rwlock*, ***b****uild_lock* and ***c****hain_lock* to satisfy the parallel emulation rules. In the beginning, emulation thread grabs read lock of ***e****xclusive_rwlock* to activate the emulation of particular guest core. This guarantees the exclusiveness of ***Destruct*** (rule 2), that write ***e****xclusive_rwlock* is obtained before *Flush* and *Invalidate* proceed. To reduce the waiting time for write lock, thread in-wait will inject fake guest interrupts to other guest cores in emulation (lock will be released before entering *Check Interrupt*), if wait time expired. The ***b****uild_lock* is dedicated for code generation activity, and ***c****hain_lock* is for TB chaining / unchaining, as its name suggests.

Due to the asynchronous nature of guest interrupts, it is delivered through a host signal in QEMU, which triggers TB unchaining. To avoid deadlock with regular emulation flow, a thread acquires ***c****hain_lock* before entering *Unchain* must go through try-lock. In case the interrupt flag of guest core asserts too long, i.e. try-lock fails, fake interrupts will be injected to those guest cores periodically, for unchaining. The interrupt response time might theoretically get worse, but in practice it is negligible to both user and guest OS codes, as the contention of ***c****hain_lock* is low.

We designate an unchaining check to handle the situation that an emulation thread is trapped in *Execute*. For a guest program with small code footprint, like multi-thread BARNS in SPLASH-2, effect of unchaining is transient that an emulation thread could barely see, let along leaving the code cache. Specifically, one emulation thread is intended to *Check Interrupt* (quit the emulation), while another is ready to do chaining. Incidentally they patch the same TB, which gets chained immediately after being unchained. Because threads share the code cache, the one intended to leave will be stuck if it misses the time TB is unchained. We defer the release of ***c****hain_lock* in *Unchain* till emulation thread exits *Execute* to avoid the problem, because no chaining will be made in-between.

B. Find Slow Optimization for UCC Design

According to rule 1 of UCC design, states in the same set will run sequentially to protect shared components. *Find Slow* and *Build* are such examples in ***Construct*** with rather frequent usage in the parallel emulator (code translation takes longer time than other emulation activities). An obvious optimization for UCC design would be ***I****ndependent* relationship between *Find Slow* and *Build* (see Table Ⅰ). By this optimization, ***Construct*** is redefined as {*Build* and *Restore*}, along with new set definition ***Search*** = {*Find Fast* and *Find Slow*}. We revise the rule 1 as:

- Any two states live in the same set must run sequentially, except ***Search***; otherwise they could go parallel.

This effectively removes the ***b****uild_lock* around *Find Slow*, and code translation will not block code execution anymore. But it also introduces the redundancy problem when two emulation threads are going to execute the same not-yet-translated guest block. Both threads will trigger code translation as none could find corresponded TB in the code cache. Yet the pending one (due to ***b****uild_lock*) does not cancel its translation request after the first is finished, for *Build* never checks TB existence before translation. Because translation output is guest-core independent (TB is shared among all emulation threads in PQEMU by default), redundancy problem here is merely memory waste, not issue of correctness.

C. Separate Code Cache (SCC) Design

To exploit more parallelism, we have SCC PQEMU design which duplicates all sharing components in Fig. 2 for every emulation thread. The only exception is *MPD* descriptor for fast SMC detection, one per guest page. As guest memory is common to all guest cores, we should not localize *MPD* for each emulation thread. Since code cache and TCG translation engine are privatized, most synchronizations in Table Ⅰ for UCC design become unnecessary (see Table Ⅱ). The only serialization appears in

TABLE II. SYNCHRONIZATIONS BETWEEN TWO EMULATION STATES FOR SCC DESIGN IN TWO-THREAD PARALLEL EMULATION.

SCC	***N***	***S***	***B***	***R***	***C***	***U***	***E***	***F***	***I***
FiNd Fast	I	I	I	I	I	I	I	I	S
*Find **S**low*	I	I	I	I	I	I	I	I	S
***B**uild*	I	I	I	I	I	I	I	I	S
***R**estore*	I	I	I	I	I	I	I	I	S
***C**hain*	I	I	I	I	I	I	I	I	S
***U**nchain*	I	I	I	I	I	I	I	I	S
***E**xecute*	I	I	I	I	I	I	I	I	S
***F**lush*	I	I	I	I	I	I	I	I	S
***I**nvalidate*	S	S	S	S	S	S	S	S	S

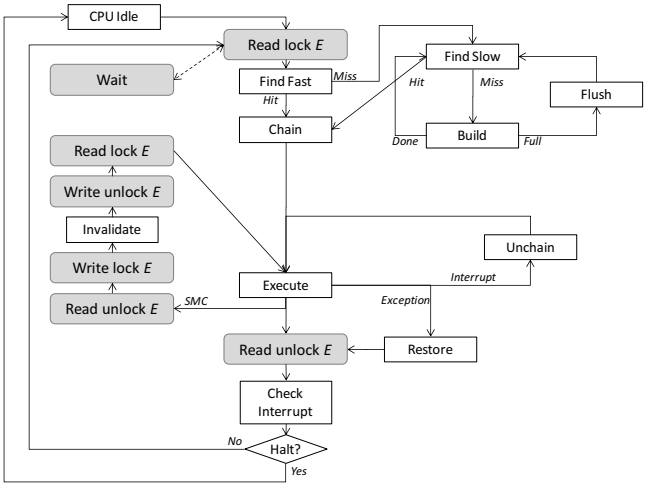

Figure 4. Modified emulation flow for PQEMU using SCC design. Due to sharing components duplication, only lock E exclusive_rwlock is required, in comparison to UCC design.

Invalidate, for guest SMC activity. To minimize cross-thread overhead, sharing components for each emulation thread are duplicated in POSIX manner. Thread could directly manipulate others' duplicates fields, instead of relying on costly inter-process communication mechanism. SCC is expected to have lower contention than UCC when delivering interrupt, because unchaining now is private to each guest core. The ***e****xclusive_rwlock* offers exclusiveness for *Invalidate*, as in UCC design. The modified emulation flow for SCC PQEMU is illustrated in Fig. 4.

D. Memory and I/O Systems in PQEMU

Because QEMU does not emulate hardware cache, the only coherence problem is between code cache and guest memory, which is already included in PQEMU designs, i.e. *Invalidate* for guest SMC activity. For guest ARM atomic instruction *swp* (swap among two registers and a memory location), PQEMU will generate TB with x86 *#Lock XCHG* instruction with some glue codes, since their instruction semantics are mutually transformable. While for *ldrex*/*strex* pair (load- linked and store-conditional on ARM platform), output code will follow the concept of transactional memory. Specifically, PQEMU will keep a table for all on-the-fly *ldrex* addresses, together with its memory content snapshot. Whenever *strex* is executed, its write address will be erased from the table. It succeeds if and only if the write-to memory content is not changed (determined by comparison to previous snapshot), and write address is still on the table. We deploy an additional lock for the table (not those appeared before), for it shares among all guest cores.

For I/O in parallel emulation, initial PQEMU inherits the old sequential model from QEMU, which halts all emulations when performing guest I/O. Later experimental variant removes such serialization since guest OS has already serialized the accesses to the same I/O device. However, it requires a thorough examination about how guest peripheral emulation functions are invoked from guest OS. It will be very complex for peculiar guest architecture like x86, and this feature is currently marked experimental.

IV. EXPERIMENTAL RESULTS

Table III lists the experimental setups and various configurations. Our PQEMU is implemented on QEMU 0.12.1, and Coremu [16] is the most up-to-date result in literature. Each SPLASH-2 [11] program is tested with one, two and four working threads, in measure of total execution time (initial single-thread setup time excluded).

We can see that the measured parallelization overhead of PQEMU designs in Fig. 5, benchmarks with one working thread (upper). On average, we have 5~10% slowdown compared to the baseline ***QEMU*** (as the 100% line). SCC designs usually have higher overhead, because the use of thread-local storage for some guest-core-private fields.

Lower part of Fig. 5 shows the benchmark results using four working threads. For computation-intensive benchmark like SPLASH-2, most guest interrupts are timers for guest OS context switches. Without experimental I/O parallelization, all emulation threads suspend when handling a guest interrupt, and ***P-UCC*** could only achieve 1.81x speedup on average. For ***P-UCC+IO***, the speedup increases to 2.88x over the baseline ***QEMU***; and ***P-UCC+IO+FS*** further advances to 3.72x speed up when *Find Slow* optimization is applied. For the SCC designs, only I/O parallelization matters because the code cache is private to each emulation thread (and no concurrent code generation and execution). Due to less lock contention and overhead, the SCC design will slightly outperform the equivalent UCC designs (***P-SCC*** to ***P-UCC***, and ***P-SCC+IO*** to ***P-UCC+IO+FS***), in around 2~4%. The tradeoffs between UCC and SCC designs are given as follow:

TABLE III. PARALLEL SYSTEM EMULATOR DESIGNS (UPPER), EXPERIMENTAL ENVIRONMENT (MIDDLE) AND SPLASH-2 BENCHMARK SETTINGS.

Parallel System Emulator Designs	
QEMU	Baseline QEMU 0.12.1
P-UCC	*PQEMU* using unified code cache design
P-UCC+IO	*P-UCC* with experimental parallel I/O model
P-UCC+IO+FS	*P-UCC+IO* with find slow optimization
P-SCC	*PQEMU* using separate code cache design
P-SCC+IO	*P-SCC* with experimental parallel I/O model
Coremu	Another parallel emulator design[16]
Experimental Environment	
Benchmark	SPLASH-2 suite using ARM v6 ISA
Guest OS	Linux 2.6.27
Guest machine	ARM11MPCore (x4 ARMv6 processors)
Guest platform	RealView EB board, 256 MB RAM
System emulator	QEMU and various PQEMU designs
Host OS	x86_64 Fedora 12 (Linux 2.6.31.12)
Host machine	Intel i7 920 (4 cores, 8 SMT) @ 2.66 GHz, 12 GB RAM
SPLASH-2 Benchmark Settings	
BARNES	Default
RADIOSITY	-batch -test
CHOLESKY	-B32 –C16384, with input cholesky.tk29.O
WATER-N / -S	Default
FMM	Default
FFT	-m20 –n65536 –l4
OCEAN	-n258 –e1e-07 –r20000 –t28800
LU/LU-NON	–n512 –b16

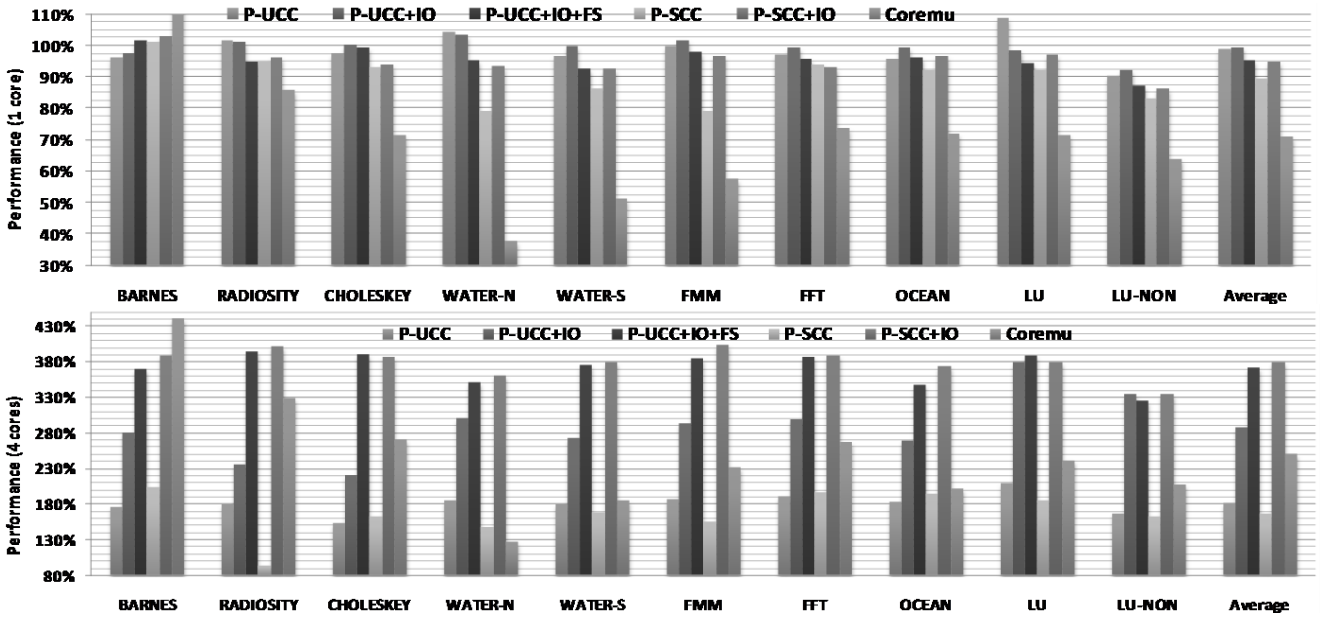

Figure 5. Performance by computation time of SPLASH-2 benchmarks using one (upper) and four (lower) working threads. We treat baseline QEMU as 100% basis.

- SCC needs more memory space and translation time, but it eliminates most synchronization except SMC *Invalidate* and emulation of guest atomic instructions in *Execute*.
- *Invalidate* in SCC incurs more overhead, because update has to apply to all duplicated sharing components. It is currently observed only at guest Linux boot-up, possibly because the memory pressure of SPLASH -2 is too small.
- Latency of guest interrupt in UCC is slightly worse than SCC, because of the contention for TB chaining and unchaining.
- UCC and SCC have the same *Restore* counts, since it depends on memory exception counts of guest program, not PQEMU implementation. But SCC experiences 1.15 to 3.00 times more *Build* than UCC, the downside of duplicating code cache– more code translations will be called for.
- Both will significantly re-shape the traffic of host cache. UCC is expected to have more cache coherence traffic, while SCC tends to experience higher cold-misses (due to duplicated yet identical TBs).
- SCC may be too costly in terms of the memory overhead when emulating a many-core guest machine. Ideally, SCC design is best for running different applications (throughput benchmarks), while UCC is for parallel applications with massive code sharing. A hybrid implementation which can be adaptive to the guest applications may be worth pursuing in the future.

Coremu [16] is another parallel emulator design, based on QEMU also. The parallelization comes from invoking many sing-core emulators at a time, one for each guest core. It resembles SCC in host process level, yet the inter-guest-core update (SMC for example) would be costly since it relies on the inter-process communication, not direct manipulation as in our designs. Coremu currently uses big lock to implement sequential I/O model, without all emulation threads fall back to emulation manager as in ***P-SCC*** and ***P-UCC***. Such design exploits 80% more parallelism between I/O emulation and guest code execution (***P-SCC*** to ***Coremu***), while duplicated code cache only introduces 12% overhead (***P-SCC*** to ***P-UCC***). For UCC design, such disadvantage could be compensated using *Find Slow* optimization, where 90% more speedup is feasible (difference in ***P-UCC+IO*** and ***P-UCC+IO+FS***). In short, I/O would be the greatest obstacle in parallel emulator designs. Merely parallelizing the core computation part would not be sufficient to efficiently exploit multi-core capabilities.

V. Related Work

Architectural simulations include micro-architectural and functional simulations. Well-known examples of micro-architectural simulation include SimpleScalar [5] for cycle simulations and Wattch [4] for power consumption simulations. To further observe the interactions between application threads and the OS, some full system simulators incorporate micro-architectural simulation capabilities, for example, RSIM [14], SimOS [11], Simics [12], Mambo [2] and M5 [1] support system simulations and selective micro-architectural simulations.

For some applications, such as validating an application in a different ISA, functional simulations alone would be sufficient. Both QEMU [6] and Bochs [8] are examples of full system emulators, and SimOS [11], Simics [12] have mode for fast functional simulation. Functional simulations also allow the interactions among processors, memory and peripherals to be observed. Recent functional emulators

usually equip with dynamic binary translation [3] for increased simulation efficiency. In today's multi-core environment, parallelism exploitation becomes a major issue in emulator designs. For example, IBM Mambo [9] and Parallel Embra [10] are parallel version of Mambo [2] and Embra [15] respectively. Mambo [2] regards the emulation as a series of hardware operations, and their execution are scheduled by *tsim* inside the simulator. Embra [15] then focuses on the parallelization of such user-space schedulers. Parallel Embra [10] leaves such scheduling work to the host OS, and uses the round-robin scheduling if there are more guest cores than the number of physical cores in the host machine. The authors also give a brief discussion of challenges in designing a parallel emulator for the MIPS machine.

Coremu [16] is the latest research that shoots for the same target of this paper – supporting parallel emulation with QEMU. However, its parallelization approach comes from a quite different direction by lunching multiple emulators at the same time. This "multi-emulator" design is similar to the SCC design at process level in PQEMU. However, the synchronization overhead between processes in Coremu is greater, even with their optimized message passing interface. Portability to new architectures is the main concern of Coremu while PQEMU is targeting at greater simulation efficiency.

VI. Conclusion And Future Work

Full system emulators have been widely employed in software development cycle, especially before hardware is available. To fully utilize the processor-level parallelism of recent multi-core systems, emulators must also go parallel. In this paper, we have identified the challenges in designing and implementing such parallel emulators, and prototyped a parallel QEMU called PQEMU. The concept to parallelize a dynamic binary translator centric simulator is generalized as an emulator-neutral mathematical model, and can be applied to other system emulators than QEMU. The implementation of PQEMU takes care of architectural dependent features (in this study, the guest architecture is ARM11 MPCore) such as the handling of atomic instructions and I/O requests. We have experimented with two design alternatives, notably the Unified Code Cache (UCC) design and the Separate Code Cache (SCC) design, to explore the tradeoffs between memory space and emulation speed. Intuitively, SCC requires less synchronization overhead. However, our experiments show that the difference is not significant since the emulation of typical guest programs do not spend majority of time in dynamic code translation – once the code is translated and stored in the code cache, the emulation will remain in the native execution in the code cache. In addition, since typical multi-threaded programs share a large portion of code among threads, the increased memory space requirements of SCC may become a major issue as the emulation scales up to many cores. Most existing parallel system emulator implementations are based on the SCC design; hence, our UCC design offers an attractive alternative.

Acknowledgment

The authors wish to thank their anonymous referees for all of their invaluable suggestions. The authors would also like to thank MediaTek Co. Ltd for their help in financial support. The work presented in this paper was supported by "MediaTek embedded systems technology research and personnel training program", Project No. 100F2211EA.

References

[1] N.L. Binkert, R.G. Derslinki, L.R. Hsu, K.T. Lim, A.G Saidi, S.K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro. 26*, 4 (July-Aug. 2006), pp. 52-60.

[2] Bohrer, P., Peterson, J., Elnozahy, M., et al, "Mambo: A Full System Simulator for the PowerPC Architecture," *SIGMETRICS Perf. Eval. Rev*. 31, 4 (Mar. 2004), 8-12.

[3] Cmelik, R.F., and Keppel, D, "Shade: a fast instruction set simulator for execution profiling," Technical Report UWCSE-93-06-06, Dept. Computer Science and Engineering, University of Washington.

[4] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," In Proc. of *the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[5] D. C. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *Computer Architecture News*, 25(3): 13–25, June 1997.

[6] F. Bellard, "QEMU, a fast and portable dynamic translator," In In Proc. of the *USENIX Annual Technical Conference*, pages 41–46, April 2005.

[7] K. Hirata and J. Goodacre, "ARM MPCore; the streamlined and scalable ARM11 processor core," *ASP-DAC* '07, pages 747–748, Jan. 2007.

[8] K. P. Lawton, "Bochs: A portable PC emulator for Unix," *Linux Journa*l, vol. 1996, no. 29, p. 7, 1996.

[9] Kun Wang, Yu Zhang, Huayong Wang, Xiaowei Shen, "Parallelization of IBM mambo system simulator in functional modes," ACM *SIGOPS* Operating Systems Review, v.42 n.1, January 2008

[10] Lantz R.,, "Fast functional simulation with parallel Embra," In Proc. of the 4th Annual Workshop on Modeling, Benchmarking and Simulation, 2008.

[11] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "The SimOS approach," *IEEE Parallel and Distributed Technology*, vol. 4, no. 3, 1995.

[12] Peter S. Magnusson et al. Simics: "A Full System Simulation Platform," IEEE *Computer*, 35(2):50–58, February 2002.

[13] S. C. Woo, M. Ohara, E.Torrie, J.P. Singh and A. Gupta, "The SPLASH-2 Characterization and Methodological Considerations," *22nd Annual Int. International Symposium on Computer Architecture*, June 1995.

[14] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve., "RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors," In Proc. of *the Third Workshop on Computer Architecture Education*, February 1997.

[15] Witchel, E. and Rosenblum R., "Embra: fast and flexible machine simulation," *In Proc. of the SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems.* 68-78Yourst, MT. 2007.

[16] Zhaoguo Wang, Ran Liu,et,al., "COREMU: A Scalable and Portable Parallel Full-system Emulator," PPoPP 2011