Scalable Deterministic Replay in a Parallel Full-system Emulator *

Yufei Chen † Haibo Chen ‡

†School of Computer Science, Fudan University ‡Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

chenyufei@fudan.edu.cn haibochen@sjtu.edu.cn

http://ipads.se.sjtu.edu.cn/coremu

Abstract

Full-system emulation has been an extremely useful tool in developing and debugging systems software like operating systems and hypervisors. However, current full-system emulators lack the support for deterministic replay, which limits the reproducibility of concurrency bugs that is indispensable for analyzing and debugging the essentially multi-threaded systems software.

This paper analyzes the challenges in supporting deterministic replay in parallel full-system emulators and makes a comprehensive study on the sources of non-determinism. Unlike application-level replay systems, our system, called ReEmu, needs to log sources of non-determinism in both the guest software stack and the dynamic binary translator for faithful replay. To provide scalable and efficient record and replay on multicore machines, ReEmu makes several notable refinements to the CREW protocol that replays shared memory systems. First, being aware of the performance bottlenecks in frequent lock operations in the CREW protocol, ReEmu refines the CREW protocol with a seqlock-like design, to avoid serious contention and possible starvation in instrumentation code tracking dependence of racy accesses on a shared memory object. Second, to minimize the required log files, ReEmu only logs minimal local information regarding accesses to a shared memory location, but instead relies on an offline log processing tool to derive precise shared memory dependence for faithful replay. Third, ReEmu adopts an automatic lock clustering mechanism that clusters a set of uncontended memory objects to a bulk to reduce the frequencies of lock operations, which noticeably boost performance.

Our prototype ReEmu is based on our open-source COREMU system and supports scalable and efficient record and replay of full-system environments (both x64 and ARM). Performance evaluation shows that ReEmu has very good performance scalability on an Intel multicore machine. It incurs only 68.9% performance overhead on average (ranging from 51.8% to 94.7%) over vanilla COREMU to record five PARSEC benchmarks running on a 16-core emulated system.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Algorithms, Performance, Reliability

Keywords Scalable Deterministic Replay, Full-system Emulator

1. Introduction

Full-system emulation is very useful for a number of usage scenarios, including pre-silicon system software development, characterizing performance issues, exposing and analyzing software bugs. Compared to a real machine, a full-system emulator saves lengthy machine rebooting by completely running in user-mode, can have much richer runtime information via introspection, and can flexibly provide an execution environment completely different from the host platform in scales and even ISAs (Instruction Set Architecture). For these reasons, many system software developers have chosen to use a full-system emulator in the major cycle of system developing and debugging.

Ideally, a full system emulator should provide true parallelism so that it can produce a comparable set of access interleavings as a real machine. This enables it to expose many concurrency bugs in an early stage. Further, by leveraging the abundant multicore resources, the performance and scalability of a full-system emulator can be significantly boosted. Hence, there have been several efforts in building parallel full-system emulators, including Parallel SimOS [13], COREMU [22], PQEMU [6] and HQEMU [9]. Unfortunately, adding parallelism to a full-system emulator makes its execution even more non-deterministic, due to the additionally introduced racy memory accesses. To enable reproducibility of guest software stack, it is vitally important to add deterministic replay features to parallel full-system emulators.

Previous researchers have proposed various schemes for deterministic replay in both application and full-system level. There are a number of software-based approaches that can provide relatively low overhead to replay a multi-threaded application [21]. However, they may not be easily adopted to efficiently support full-system replay as system and device emulation in full-system emulators cannot be trivially rerun and a number of racy execution will significantly degrade performance. SMP-ReVirt [8] incorporates the CREW protocol [14] to replay a virtual machine running atop a virtualized platform (i.e., Xen). However, the difference between a virtualized platform and an emulation platform makes it non-trivial to directly apply their approach to a full-system emulator. Besides, according to our analysis, the previous CREW protocols

* This work was done when Yufei Chen was a visiting student in Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University. This work was funded by China National Natural Science Foundation under grant numbered 61003002, A grant from Shanghai Science and Technology Development Funds (No. 12QA1401700) and A Foundation for the Author of National Excellent Doctoral Dissertation of PR China.

PPoPP'13, February 23–27, 2013, Shenzhen, China.

suffer from performance and scalability issues when replaying a relative large number of cores.

One possible approach to providing deterministic replay to a full-system emulator is treating it as an application for application-level replay systems. However, such an approach loses the capability of incorporating debugging tools to introspect or manipulate internal information regarding guest software stack [19], as this may change the execution behavior of the emulator and the guest software stack.

This paper makes the first attempt to provide deterministic replay capability to parallel full-system emulators. Our goal is to efficiently support scalable record and replay of a relative large number of emulated cores running the entire software stack.

Our analysis indicates that there are several unique differences between a native or virtualization platform and an emulator, which creates both opportunities and challenges. First, there are only software MMUs instead of hardware MMUs in an emulator, which provides more flexibility as well as performance overhead. Second, there is no convenient way to uniquely identify a memory instruction in an emulator[1], making it costly to record the shared memory accesses. Finally, the dynamic binary translator in an emulator can behave differently during record and replay, yet its execution affects the state of the emulated system.

As shared memory accesses are the key factor in a replay system, we carefully study possible schemes that can be adopted in a full-system emulator. We choose the CREW (concurrent-read, exclusive-write) protocol, which was proposed by Courtois et al. [5] and was first used by Leblanc and Mellor-Crummey [14] to replay loosely-coupled message-passing applications. It has been also used by recent systems such as SMP-Revirt [8] and Scribe [12], to record and replay shared memory accesses. Essentially, the CREW protocol serializes racy memory accesses including write-after-write, read-after-write and write-after-read and logs the serialized order of accesses during replay run.

To improve the performance and scalability of ReEmu, we make a comprehensive analysis on possible issues in the previous CREW protocols. First, previous CREW protocols either acquire a huge system-wide lock (e.g., SMP-Revirt [8]) or a spinlock multiple times to serialize shared memory accesses. This severely limits the performance and scalability of a replay system. Finally, adopting a read-write lock scheme [5, 14] may further cause write starvation under heavy contention over a shared memory object. To address this issue, we observe that the design of seqlock[2] essentially matches the design of CREW protocols as both design uses a version number to indicate whether a memory object is contended or not and to serialize the access order to a shared memory object. Based on this observation, ReEmu reuses and extends the design of seqlock to serialize and log access orders of shared memory accesses. The resulting design allows complete lock-free read accesses and non-blocking write access to shared memory objects, thus significantly boost performance and scalability in record and replay runs.

Second, previous CREW protocols usually result in an excessive number of redundant logs, which not only increase the log size, but also cause runtime overhead in record run. Further, some previous CREW protocols requires accesses [8] to remote CPU information, and thus may limit their performance scalability. Finally, some schemes may result in logging overconstained access orders, which may limit parallelism and thus performance in replay run. To reduce unnecessary logs and improve logging efficiency, ReEmu dynamically detects version changes and only takes logs when necessary. Further, it only requires accesses to mostly local CPU information for logging, but instead relies on an offline log processing tool to infer shared memory dependence.

Third, though ReEmu has tried to minimize the amount of lock operations, per-update lock acquisition may still be a limiting factor to performance. To address this issue, ReEmu design and implement a lock clustering mechanism that clusters multiple uncontended memory accesses together into a bulk, which requires only one lock operation. This noticeably reduces expensive lock operations.

To demonstrate the effectiveness and efficiency of our proposals, we have implemented our techniques based on the open-source COREMU and support deterministic replay of full software stack based on both x86 and ARM. Performance evaluation shows that ReEmu incurs around an average of 68.9% (ranging from 51.8% to 94.7%) performance overhead over COREMU to record five PARSEC applications with different characteristics on a 16-core emulated machine. Using the racy benchmark [23], we show that ReEmu can faithfully reproduce concurrency bugs.

In summary, this paper makes the following contributions:

1. The first attempt to add the full-system deterministic replay feature to parallel full-system emulators, as well as a comprehensive analysis of the sources of non-determinism.
2. A comprehensive analysis on issues with prior CREW protocols and a set of novel refinements that provide efficient and scalable record and replay of a full-system software stack wtih contemporary workloads.
3. A working prototype that demonstrates both effectiveness and efficiency of ReEmu.

In the rest of this paper, we first describe the overall design of ReEmu and how ReEmu tracks different sources of non-determinism. Then, we describe the key part of ReEmu by illustrating how ReEmu logs shared memory dependence for faithful replay. Next, we describe how ReEmu is implemented in an open-source full-system emulator. Section 5 then evaluates the performance overhead and scalability of ReEmu and how the refinements to CREW contribute to the performance. We then relate ReEmu to prior work (section 6) and conclude this paper with a brief discussion on current limitation and possible future work (section 7).

2. ReEmu Overview

ReEmu is designed to support record and replay the execution of whole guest software stack running atop, including a hypervisor, an operating system and the applications. ReEmu can checkpoint the states of guest software stack, start execution while logging all sources of non-determinism and then replay the execution from the checkpointed states. Like other full-system emulators such as QEMU, ReEmu runs completely in user space and thus it is very convenient to start and terminate its execution.

ReEmu records and replays sources of non-determinism by using dynamic fine-grained instrumentation. This contrasts with some previous systems [8, 12] by leveraging hardware-based memory protection (e.g., MMU). Using dynamic instrumentation enables ReEmu to choose the granularities of shared memory objects. Fine-grained tracking can avoid false sharing at the cost of increased memory consumption due to the associated metadata, while coarse-grained tracking may just be opposite. Further, not relying on a specific hardware memory protection scheme makes ReEmu easily portable among different platforms (e.g., x86 and ARM).

In the following sections, we will first review the general architecture of full-system emulators and illustrate the sources of non-

[1] Native systems usually leverage performance counters such as program counter and branch counter [7, 8] to uniquely identify an instruction.

[2] Seqlock is short for sequential lock, which is special locking scheme in Linux kernel that allows fast writes to a shared memory location among multiple racy accesses.

determinism. Then, we will describe how ReEmu records and replays them accordingly.

2.1 Full-system Emulator

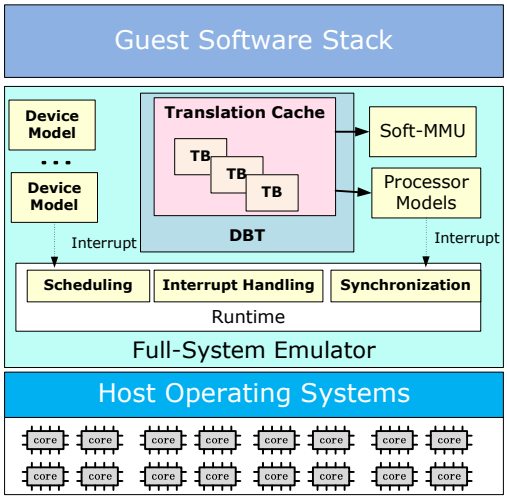

Figure 1. General architecture of full-system emulators using dynamic binary translation.

Figure 1 shows an overview of the general architecture of typical full-system emulators. The emulator usually runs as a user-level program in the host operating system and emulates multiple cores and devices for the guest software stack. State-of-the-art emulators usually use a dynamic binary translator (DBT) to first translates the guest machine code into a common intermediate format, which is then translated into the host machine code. Here, an instruction in guest machine code may be decomposed and reorganized into several host machine instructions. The guest processor states are also mapped into a portion of memory in the emulator. To avoid re-translation of target code, a DBT usually maintains a translation cache that caches translated blocks (TBs). For the sake of performance, a DBT usually supports chaining its translation blocks (TBs) to allow directly jumping from one TB to another TB, without the need to jump out of the translation cache.

Unlike user-level emulators, a full-system emulator also needs to emulate the address translation from guest virtual address to guest physical address and then to host virtual address. A software MMU is usually used to assist such a process. A soft-TLB that resembles a real TLB caching such address translation. Upon misses in the soft-TLB, the emulator will traverse guest page table to do address translation and raise page faults if the address mapping is not present. In addition, the emulator needs to emulate devices such as network interface cards and disks, as well as interrupt mechanism to notify the receiving virtual cores. As a full-system emulator is itself a multi-threaded program, it may also need to handle synchronization and scheduling among virtual cores.

2.2 Sources of Non-determinism

Some sources of non-determinism are common among application-level replay schemes. In addition, the record and replay code of ReEmu lie in and may interact with the full-system emulator, which is non-deterministic. Hence, other than the sources of non-determinism inside the guest software stack, we also need a careful examination on which sources of non-determinism in the emulator may affect a faithful replay of guest software stack. The followings list the sources of non-determinism:

Interrupts: Interrupts in guest software stack are asynchronous events, whose delivery timing will affect the execution of guest software stack. Hence, ReEmu needs to record when an interrupt occurs and to inject the interrupt at the right time during replay.

DMA handling: Handling a typical DMA involves the following steps: 1) CPU issues a DMA command by writing to device registers; 2) The device starts executing the DMA command, which reads or writes memory; 3) When the DMA operation is done, the device will send an interrupt to CPU. Here, the timing of interrupt and the access orders to the DMA memory between the device and CPU cores may both affect guest execution, which need to be recorded as well.

Orders of Guest Page Table Walking: The orders to walking a shared guest page table by multiple cores may affect the execution of guest software stack. The walking order is caused by the internal races among multiple cores walking the same page table. If two cores encounter TLB misses on the same address not present in the guest page table, there will be only one core handling the page fault. Hence, if a different core from that in the record run handles the page fault in the replay run, the execution behavior of guest software stack will diverge, which may lead to either unfaithful replay or even replay failures.

Non-deterministic Instructions: The guest software stack may also issue some non-deterministic instructions, including synchronous accesses to device states (e.g., I/O instructions such as in/out and accesses to memory mapped region), and reading non-deterministic CPU states (e.g., "rdtsc"). These instructions will happen in fixed point in guest instruction stream. However, the execution result is non-deterministic.

Shared Memory Accesses: ReEmu also needs to handle memory accesses from the translation cache. The access orders from multiple cores to a shared memory object may affect the execution of guest software stack, especially for racy accesses.

Unnecessary Non-determinism: The accesses to guest state from the DBT may be different between the record and the replay run. Such accesses include code translation and guest page table walking in soft-MMU. Code translation needs to read guest memory to translate guest instruction stream. However, the code generation strategy may be different due to optimization policies and code cache flush behavior are not the same. It could happen that some guest code is translated only once during recording, but is translated twice during replay. As translation need to read guest memory, this will affect the state of soft-TLB. Hence, the soft-TLB state may also be different between record and replay. Fortunately, code translation only reads guest memory states, and the soft-TLB state should be transparent to guest software stack. Hence, such accesses need not to be handled.

2.3 Record and Replay Non-determinism

Identifying Guest Execution: Previous native or virtualized systems usually use some performance counters to accurately identify the location of guest execution. For example, SMP-ReVirt [8] leverages a combination of the program counter and branch counters to uniquely identify the timing of an interrupt and memory instructions. However, the program counters are not updated upon the execution of every instruction and there is no branch counter maintained to save performance overhead in a full-system emulator.

ReEmu uses two mechanisms to identify guest execution. First, as most full-system emulators using binary translator only inject interrupts in a basic block (i.e., translation block or TB in QEMU) boundary. Hence, ReEmu maintains a per-core counter (BB Counter) that counts the number of executed basic blocks in software and use the BB Counter to record the timing for interrupts. Second, to identify a memory instruction, ReEmu maintains a per-core memory counter that counts the number of memory in-

structions executed in each emulated core. Note that, as these two counters are per-core based, updates to these counters are normal memory operations, instead or being protected using either locks or atomic instructions.

Handling Interrupts: ReEmu logs the timing of interrupts by recording the BB counter of the corresponding virtual core. In addition, ReEmu records the interrupt number as well. In replay run, ReEmu checks whether the number of executed TBs reaches the recorded number before jumping to the successive translated block. If an interrupt should be injected, ReEmu jumps to the corresponding interrupt handler code and unchains the translated blocks.

Handling DMA: The interrupt can be recorded and replayed using the mechanism described above. As DMA also has memory operations, the access order between the device and cores also need to be recorded.

Typically, during the execution of a DMA command, the DMA memory region should only be accessed by the device, otherwise the memory content may be corrupted because of concurrent memory accesses. One way to record memory orders between device and CPU cores is to treat DMA device as a special core, which only accesses memory when it receives DMA commands [8, 23]. This will record the order between memory accesses for CPU and devices. However, this may prevent the opportunity of using a replay scheme to detect DMA related bugs as the memory accesses between CPU and device to the DMA memory region are serialized. If the operating system is buggy and allows concurrent accesses to DMA memory region, such concurrent accesses will not manifest during the record run.

Instead, ReEmu does not enforce such an order but instead add checking code to detect such parallel accesses. ReEmu utilizes the memory order recording mechanism (see section 3) to detect concurrent accesses to the DMA memory region. If the operating system is correct and guarantees no concurrent access to DMA memory region, by correctly replaying the execution of the operating system, the order of memory access for CPU and device should be exactly the same with the original run. Hence, we just need to ensure that the interrupt indicating the completion of a DMA operation happens after the corresponding DMA has completed. If the OS does not enforce such a DMA access order, then the record and replay run may diverge and thus we can detect such a bug.

To detect such bugs during recording, a DMA read request (which writes memory) will first acquire write locks for each shared object which it is trying to access, and mark the shared object as being under DMA operations. Any memory accesses to the shared objects with DMA operations marked are concurrent accesses to DMA memory regions. In such cases, ReEmu will report the problem and continue recording.

To track when a DMA request completes, ReEmu maintains a DMA completion counter for each DMA device. The counter is updated each time a DMA operation completes. When recording a DMA interrupt, we also record the corresponding device's DMA completion counter. During replay, before injecting a DMA interrupt, ReEmu should wait until the corresponding device's DMA completion counter reaches the recorded value.

Access order to MMU: To enforce the same order of page faults during record and replay runs, ReEmu logs each page fault by recording the memory count, the faulting address and the core ID. In replay run, ReEmu checks if the core ID corresponding to a page fault matches the current running core ID when the memory count and fault address match. If the core ID matches, ReEmu will inject the page fault. Otherwise, this core should wait until another core has handled the page fault before it can proceed.

Non-deterministic Instructions: For non-deterministic instructions, ReEmu simply logs the return results of the corresponding emulation functions in the DBT in record run. In replay run, ReEmu just returns the logged results to the calling functions and does not actually execute the functions for most applications. However, there are some functions like reading some device register may have side effects. ReEmu still needs to execute such functions.

Shared Memory Accesses: ReEmu leverages the CREW protocol [5, 14] to record orders of shared memory accesses, but with a notable redesign and optimizations, which will be detailed in the following section.

3. Replaying Shared Memory Systems

A key challenge in a replay system is to replay shared memory accesses. One approach is recording the first read values of memory locations (e.g., BugNet[17]). However, it cannot infer shared memory access orders and thus cannot provide much information for debugging. Another approach is serializing and logging the order of read and write accesses to each shared memory object, so that it is possible to infer the partial or total order among memory accesses to a shared memory object among different cores. The CREW protocol [5, 14] serializes accesses to a shared object by enforcing and logging a total order among writers and a total order of readers with respect to writers. However, it places no constraints among multiple readers. We choose such a protocol as it is very suitable for dynamic instrumentation (e.g., InstantReplay [14]).

However, most previous software-based work only applies the CREW protocol on a few number of cores (e.g., 4 cores in SMP-Revirt [8] and 2 cores (4 threads) in Scribe [12]), without the consideration of scalable record and replay of running on a machine with a relatively large number of cores with non-trivial racy execution[3].

In this section, we first examine the performance and scalability issues with the original CREW protocol and its variants. Then, we propose our refinements and optimizations that significantly boost performance and scalability.

3.1 Previous CREW Protocols

Generally, there are two states for each shared memory object in the CREW protocol: 1) *concurrent-read*, where all cores can read but none can write; 2) *exclusive-write*, where only one core can read and write, while other cores cannot access that shared object. Algorithm 1 shows the original design of the CREW protocol in Instant Replay [14]. The essence of the protocol is the incorporation of the Read-Write Lock with the logging of object versions, and leverage the object versions to infer and enforce access orders of read/write accesses to a shared memory object. This design protects every read and write instruction with a read-write lock. One interesting point of the CREW protocol in Instant Replay is that version information itself is enough to define orders between read-after-write and write-after-write memory accesses.

Our analysis shows that such a protocol has significant performance and scalability issues when replaying contemporary multi-threaded applications on commodity multicore processors. First, it requires taking a log on every memory access, which will cause a huge per-access overhead for memory instructions. Second, the locks and atomic instructions associated with instrumentation may incur huge runtime overhead and scalability issues when replaying systems with multiple cores. Third, as the writer needs to wait until there is no reader accessing the memory object, it will be easily

[3] Though Instant Replay reported the results on a machine with a relatively large number of processors. The target applications are based on message-passing, which have very little contention on shared memory. Further, it is based on a machine in 25 years ago, at which the memory wall was not a problem.

```
1  Read object begin
2      P(object.lock)
3      AtomicAdd(object.activeReader, 1)
4      V(object.lock)
       // Write to core local log file
5      WriteLog(object.version)
6      Do Actual Read
7      AtomicAdd(object.totalReaders, 1)
8      AtomicAdd(object.activeReaders, −1)
9  end
10 Write object begin
11     P(object.lock)
12     while object.activeReaders ≠ 0 do delay
13     WriteLog(object.version)
14     WriteLog(object.totalReaders)
15     Do Actual Write
16     object.totalReaders ← 0
17     object.version ← object.version
18     V(object.lock)
19 end
```

Algorithm 1: InstantReplay Record Algorithm [14]

starved, especially given the fact that the reader-side critical section in this protocol is quite lengthy.

```
1  Read object begin
2      version ← ReadLog
       // wait write to this object
3      while object.version ≠ version do delay
4      Do Actual Read
5      AtomicAdd(object.totalReaders, 1)
6  end
7  Write object begin
       // Read recorded version and totalReaders
8      version ← ReadLog
9      totalReaders ← ReadLog
       // wait write to this object
10     while object.version ≠ version do delay
       // wait reads to this object
11     while object.totalReaders ≠ totalReaders do delay
12     Do Actual Write
13     object.totalReaders ← 0
14     AtomicAdd(object.version, 1)
15 end
```

Algorithm 2: InstantReplay Replay Algorithm

To reduce the per-access overhead associated with each memory access, recent virtualization-based [8] and application-level [12] replay systems have leveraged hardware MMUs to enforce and track access orders among shared memory accesses. When a memory page is in the concurrent-read state, the page table entry of that page in each core has read-only permission. When a memory page is in exclusive-write state, only one core's page table entry has read-write permission to that page, while all other cores have no permission to that page. A CREW fault will be raised if a memory write has no sufficient permission, where the memory access ordering constraints (i.e., "happen-before" relationships) can be logged.

For example, as shown in Figure 2, a shared object X is first concurrently read by three cores. At time t7, Core0 wants to write the shared object, at which a CREW fault happens as Core1 does

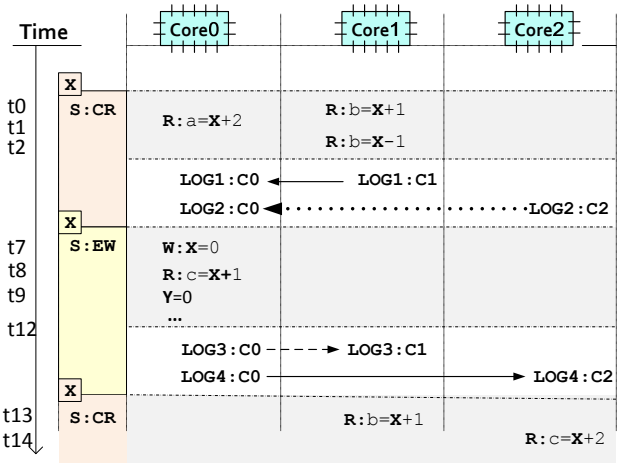

Figure 2. An overview of the CREW protocol based on MMU protection: left squares are a shared object in different states: concurrent-read (CR) and exclusive-write (EW); top squrares are CPU cores accessing the shared object. The solid arrows indicate the captured "happen-before" relationship between two memory operations. Log2 is a redundant log and LOG3 is an overcontrained log.

not have sufficient permission. Hence, the record system has to first decrease permission on Core1 and Core2 and then increase the permission on Core0. The record system then logs that the permission decrease happens before the permission increases and the replay system enforces such orders during replay. Once Core0 has both read and write permission with the shared object, the following read operation at t8 can go without further CREW faults. However, when Core1 tries to read the object at time t13, a CREW fault again happens due to insufficient permission. Here, Core0 will decrease its permission before Core1 can increase their permissions. Core1 needs to notify all other cores to increase their permission after Core0 decreases its permission. As Core2 has already increased its permission in t3, subsequent read access at t4 can proceed without CREW faults.

It should be noted that upon a CREW fault, though only the faulting core needs to acquire a lock and remove other cores' permission, it needs to send IPIs (Inter-Processor Interrupts) to all other cores to take logs and flush TLB, as it is uncertain whether other cores are sharing the current object or not. Hence, such a design may cause scalability issues for a large number of cores and may cause frequent CREW faults upon non-trivial racy execution.

Further, it may also record over-constrained order as well as unnecessary logs. For example, LOG3 in the above example records that the read instruction at t13 of Core1 should happen after the instruction in t12, which actually should happen after t7 precisely. This is because there is no easy way for the MMU to identify the last writer of a shared object. Though the logged order is sufficient to ensure correct replay, this decreases parallelism as instructions between t7 and t13 can be executed in parallel for all cores, but is unfortunately serialized by this overconstrained log.

As the above MMU-based CREW protocol has no idea of which core may access a shared memory object, it may also take a lot of unnecessary logs. For the above example, LOG2 is unnecessary as core2 is not sharing X at that time. Suppose there are 16 cores in the above example and only 3 cores will share X, all other cores will have to be interrupted and take unnecessary logs. This will greatly impact performance.

3.2 A Scalable CREW Protocol

Being aware of the performance and scalability with prior CREW protocols, we propose a scalable and efficient CREW protocol. As ReEmu is based on a full-system emulator, where hardware MMU is not available, we mainly rely on dynamic binary instrumentation to enforce and log orders among concurrent accesses to a shared memory object.

The key idea under the new CREW protocol is that the seqlock used in Linux essentially matches the design of the CREW protocol. Hence, we redesign the CREW protocol by reusing the seqlock.

3.2.1 Data Structures

For each core, ReEmu maintains a *memop* describing the total number of memory accesses in each core. This can uniquely identify the timestamp of a memory access. For each memory object, ReEmu maintains a version number, which remains even and unchanged for a read access, but is increased twice for a write access. The first increment during a write access indicates that a write operation is in progress, while the second increment indicates that the write operation has finished. Thus, the odd or even value of the version number indicates whether there is a write access in progress.

For each memory object on each (virtual) core, ReEmu maintains a *last_seen* structure that describes the last access to a shared memory object, including the version number (i.e., version) and the timestamp (i.e., memop). The *last_seen* structure is used to avoid unnecessary and overconstained logs mentioned in previous CREW protocols.

ReEmu divides log entries into two types: *wait-version* log, and *wait-read* log. As shown in the LogOrder function, each wait-version log item is a tuple of (*memop*, *version*). Each core has such a version log recording every memory access that needs to obey the write/read-after-write order. During replay, when the memop reaches a recorded memop, the virtual core must wait until the accessing object's version reaches the recorded version.

Each wait-read log entry contains a tuple of (*object id, version memop, coreid*). For example, a log tuple (3, 141, 59, 2) means that, if there is any write to a shared object with ID 3 at version 141 on other cores, it must wait until core 2 has done 59 memory accesses. Hence, this log ensures the write-after-read order.

3.2.2 Record Phase

Algorithm 3 illustrates the main algorithm of the record phase in ReEmu. Line 1 to line 16 in the *Read* function show how ReEmu logs read accesses to a shared memory object. ReEmu reads the global object version and checks if there is a pending write access by checking if the version number is odd or not. ReEmu repeats this process until the version number becomes even (line 4-7). Then ReEmu performs the actual read (line 8) and checks if there is or has been any write access during the read operation and retries the read if so (line 9).

If the memory object has been updated by other cores since last access in this core (line 10), ReEmu needs to log the shared memory dependence (line 11) and refresh the version last seen by this core for the object (line 12). Finally, ReEmu updates the last access memory operation count (memop) (line 14) and increases the memory operation count (line 15).

To track and log a write access, ReEmu first acquires the per-object lock to prevent concurrent write accesses to an object (line 18) and takes a snapshot of current version of the memory object (line 19). It then increases the object version to an odd value to exclude a potential read access (line 20), performs the actual write (line 21), increase the object version to an even value so that a potential read can proceed, and finally release the per-object lock. ReEmu checks if there is any other write access to this object (line 24) and takes a log on such a write/read-after-write dependence if so (line 25). ReEmu finally flips the memop in the *last_seen* to indicate that this access is a write operation, updates the version number seen by this core to the object, and increases the memory operation count.

The LogOrder function takes logs by writing the memory operation count and version to the per-core wait-version log file (line 32). If the last access to this object is a read operation (line 33), ReEmu takes an additional wait-read log that records the write-after-read dependence as well.

```
1  Read (object, last_seen) begin
       // last_seen contains info about last access, local to core
2      repeat
3          version ← object.version
4          while version is odd do
5              version ← object.version
6              delay
7          end
8          Do Actual Read
9      until version = object.version
       // Version change means modified by other core
10     if last_seen.version ≠ version then
11         LogOrder(object, last_seen)
12         last_seen.version ← version
13     end
       // memop is also local to core
14     last_seen.memop ← memop
15     memop ← memop + 1
16 end
17 Write (object, last_seen) begin
18     SpinLock (object.lock)
19     version ← object.version
       // Odd version locks reader
20     object.version ← object.version + 1
21     Do Actual Write
22     object.version ← object.version + 1
23     SpinUnlock (object.lock)
24     if last_seen.version ≠ version then
25         LogOrder(object, last_seen)
26     end
       // Complement to be negative, mark last access as write
27     last_seen.memop ← ∼ memop
28     last_seen.version ← version + 2
29     memop ← memop + 1
30 end
31 LogOrder(object, last_seen) begin
32     WriteVersionLog(memop, version)
33     if last_seen.memop ≥ 0 then
34         WriteReadLog(object.id, last_seen.version,
           last_seen.memop)
35     end
36 end
```

Algorithm 3: ReEmu Record Algorithm

3.2.3 Log Processing

As ReEmu takes a per-core log scheme and each log only contains local access information, an offline log processing algorithm is necessary to combine log files and infer access orders of a shared memory object. ReEmu only needs to process the wait-read log as each core only needs to read its local wait-version log sequentially.

ReEmu first sorts the wait-read log of each core and then merges all logs together into a single file ordered by object ID and version. ReEmu will also generate another index file that contains the starting location of the wait-read log for each object. During replay, a core performing a write access only needs to do a single sequential search to find the dependent read accesses by other cores to this object.

3.2.4 Replay Phase

Algorithm 4 shows the replay algorithm in ReEmu. For each read access, ReEmu needs to wait until the version of the object has reached to a logged version (line 2). This ensures that a read access can get the same value as in record run. Then, it performs the actual read operation (line 3) and updates the memory operation count (line 4). For each write access, ReEmu needs to first wait until other write accesses before this access have happened (line 7). Then, it needs to ensure all other dependent readers have done the read accesses (line 8). Finally, ReEmu performs the actual write and updates object version and memory operation count.

```
1  Read (object) begin
2      WaitVersion(object)
3      Do Actual Read
4      memop ← memop + 1
5  end
6  Write (object) begin
7      WaitVersion(object)
8      WaitRead(object)
9      Do Actual Write
10     object.version ← object.version + 2
11     memop ← memop + 1
12 end
13 WaitVersion(object) begin
14     version ← ReadLog()
       // wait write to this object
15     while object.version ≠ version do delay
16 end
17 WaitRead(object) begin
       // wait all reads to the object at the current version
18     for each tuple in the form of
       (object.id, object.version, readmemop, readcoreid)
       in read log do
19         while readcoreid's memop ≤ readmemop do
           delay
20     end
21 end
```

Algorithm 4: ReEmu Replay Algorithm

3.2.5 Performance and Scalability Analysis

As shown in the previous algorithms, ReEmu has several good properties that make it efficient and scalable. First, there is only one variable (i.e., object.version) shared among multiple cores, which may reduce unnecessary cache ping-ponging. Second, all logs are taken with only local information (except object.version) to per-core log files, this makes the logging process pretty fast and scalable. Third, there is only one lock to serialize multiple writers and the read side is completely lock-free. Fourth, the logs are taken only when necessary and the logged order contains no overconstained order but is precise to reflect the exact access orders.

3.3 Lock Clustering

Even though ReEmu has been designed to minimize synchronization operations, there is still some overhead related to each memory access, especially write accesses. In some cases, it is possible that for some periods of time, some memory objects are only accessed by one core. In such cases, it would be beneficial to acquire the shared object with exclusive write permission, then do all the following access directly without the need to acquire the lock again.

To enable such an optimization, ReEmu associates an owner information to each shared object. The owner information indicates which core is currently holding the write lock. On acquiring a write lock, a virtual core can hold the lock and set the owner to itself. Each memory access first checks if the owner of the shared object is itself. If so, it then can access the shared memory directly with only the need to update the memop and update last seen information. Otherwise, it goes through the original recording algorithm.

ReEmu needs to avoid possible deadlocks. Deadlock will occur if Core1 holds the lock of shared object A, and then try to access shared object B, whose lock is hold by Core2 and Core2 is trying to access shared object A. To avoid such a deadlock, each core has a mailbox. When a core tries to access a shared object that is hold by other cores, it will send a message to the owner's mailbox with the contending shared object ID, and then wait until the lock is released. Each core will check this mailbox and release the contending shared objects at the end of each basic block or when it is contending shared memory objects with other cores.

A successive access from another core to a shared object cannot release the lock, as the owning core may be doing memory access at that time. However, waiting for the owner to release the lock would incur performance overhead when there are many shared accesses. To reduce this overhead, each core is only allowed to hold a predefined number of shared objects and ReEmu tries to detect contention and disable lazy lock release when there is heavy contention.

ReEmu currently sets the maximum number of shared object each core can hold to 32. Setting this value too low may miss the opportunity to reduce overhead of lock operations, while a large value may incur excessive waiting overhead.

ReEmu tries to detect contention by recording the memop in a contending point for each shared object for each core. The contending memop is recorded whenever a core sends a shared object ID to the mailbox of the owning core, or when the owning core releases contending objects. This contending memop acts like a timestamp of last contention. A core performing write access will release the lock lazily only if the current memop is greater than the contending memop by a predefined number (e.g., 10 in our implementation). This means that some certain time has elapsed since last contention.

4. Implementation

We have implemented ReEmu based on COREMU 0.1.2 [22], an open-source parallel full-system emulator based on QEMU [2]. The implementation adds around 2500 SLOCs to COREMU. ReEmu checkpoints the guest stack by leveraging the qcow [16] support in QEMU to checkpoint guest states. ReEmu divides memory into equally-sized memory chuncks, each chunk maps to an object ID. The memory access recording algorithm uses this ID to identify a shared object. The chunk size can be changed at compile time. As object ID is accessed very frequently, ReEmu uses fixed mapping from memory address for fast object ID calculation. The object ID is calculated as $(address >>$ `CHUNK_BITS`$)\&$ `OBJID_MASK`. For a 4 KB chunk size, the `CHUNK_BITS` is 12. To bound the number of object id used, ReEmu fixes the `OBJID_MASK` to be an integer with the lowest 21-bit set. When chunk size is small, it is possible that multiple memory chunk may map to the same ID. However, this will not affect the correctness of the order recording algorithm as those chunks will be considered as a single combined shared object.

Port to ARM: To demonostrate the portability of ReEmu, we further port ReEmu to support deterministic replay of ARM. As the algorithms and instrumentation code is almost the same for ARM and x86, only around 120 SLOCs are added, which mainly lie in the atomic instruction emulation in ARM to support memory order recording.

Validating Correctness: During development, ReEmu records the PC of each executed basic block. This log defines an execution path that must be identical between record and replay. During replay, ReEmu reads the recorded PC before executing a basic block and checks if the PC are the same. Hence, we can check if the replay run is the same with the record run.

ReEmu also uses memory value verification that records the value of each read/write instruction and verifies the value during replay, which validates that shared memory accesses are recorded correctly.

5. Evaluation

Performance evaluation is done on a 20-core Intel x64 machine (2 GHZ, 2 processors with each having 10 cores with 32KB L1, 256KB private L2 and 24MB shared L3 cache) running Debian 6 with Linux kernel version 2.6.38.5. The guest OS is also a Debian 6 with kernel version 2.6.32.5. The host machine has 64GB memory and the guest is configured with 2GB memory.

For the x86 guest machine, we use `int $0x77` as a backdoor instruction marking the start and end of timing. For the ARM guest machine, we use the atomic instruction `swp` as the backdoor instruction because it's not used normally. (Neither Linux kernel or the tested application uses `swp` instruction.) All tests were executed at least four times and we report the arithmetic average of them.

5.1 Workload Characteristics

For x86 architecture, we choose five applications from the PARSEC benchmark suite (version 2.1) [3]: *blackscholes*, *bodytrack*, *canneal*, *fluidanimate*, *swaptions*. All applications are tested with the simlarge input size. PARSEC can be treated as worst-case applications due to pervasive shared memory accesses, causing larger overhead and log size. (For system applications like kernel build, ReEmu has about 50% overhead on average compared to COREMU.)

application	#sync	#shared memop	working set
blackscholes	very few	a few	grow with #core
canneal	a few	a few	large
swaptions	a few	many	small
bodytrack	medium	large	small
fluidanimate	large	very few	large

Table 1. Selected PARSEC applications characteristics

As shown in Table 1, the five applications are chosen based on their differences in the amount of synchronization primitives, shared memory accesses and the working set size: blackscholes has very few synchronization primitives, a number of shared memory accesses among two threads and a small working wet; canneal has a few number of synchronization primitives, a few amount of shared memory access and a relatively large working set; swaptions has a few number of synchronization primitives, many shared memory accesses, and a small working set; bodytrack has a medium amount of synchronization primitives, a large amount of shared memory accesses and a small working set; fluidanimate has a very large number of synchronization primitives, very few shared memory access and a large working set.

Unless otherwise noted, we use 1KB as the default memory chunk size. As we assign 2 Gbyte memory to the guest, and use the 21-bit object ID mask, there is exactly one memory chunk mapped to the same shared object ID. As some applications require the number of threads to be the power of 2, we only report the performance results for an emulated system with the number of cores be power of 2.

5.2 Performance Slowdown

Slowdown in Record Run: Figure 3 shows performance slowdown in record run compared to COREMU. The average slowdowns are 60.2%, 74.8%, 77.7%, 74.5%, 68.9% for recording these applications on 1, 2, 4, 8, and 16 cores. Among them, swaptions has a relatively large overhead (94.7% at 16 cores) due to the relatively large amount of shared write memory accesses [3], which is a limiting factor to performance. For other four applications, the performance overhead is relatively small and mostly less than 80%. The small overhead is attributed to the efficient and scalable design in the recording algorithm of ReEmu.

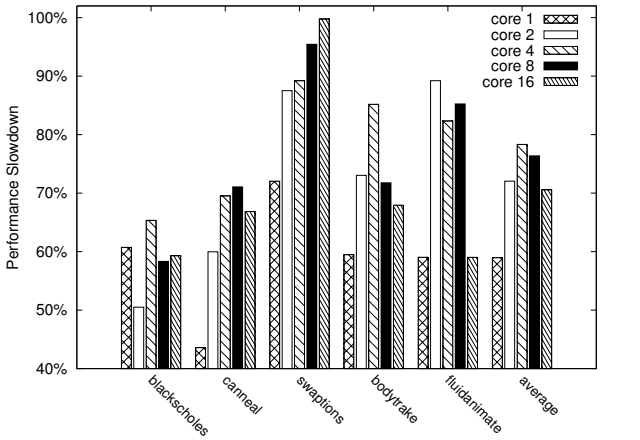

Figure 3. Record slowdown compared to COREMU

Slowdown in Replay Run: Figure 4 shows replay performance slowdown compared to COREMU. Currently, we have not done any optimization for replay. Hence, some applications are still with a very large performance overhead during replay. The main reason may be that a relatively large number of waiting operations get aggregated together to degrade the replay performance. Swaptions is with a relatively large overhead on 16 cores, probably due to the fact that a large number of shared write accesses need to wait for its dependent readers to finish.

Scalability: To see whether ReEmu has a good performance scalability on a multicore machine, we also plot the execution time of ReEmu and COREMU with the increasing amount of cores. As shown in figure 5, we can see that ReEmu in record run has similar performance scalability with COREMU. ReEmu in replay run also has good performance scalability before 8 cores but stops scaling in 16 cores due to performance degradation caused by aggregated waiting, which we will optimize in our future work.

Slowdown to Native Execution: We also compare the performance of ReEmu to native execution. The average record and replay slowdown for 1, 2, 4, 8 and 16 cores are 17X, 16X, 14X, 14X, 18X and 16X, 15X, 14X, 17X and 33X accordingly, while the slowdown caused by COREMU is 11X, 10X, 8X, 8X and 12X accordingly. Though the slowdown is still relatively large, it is still much smaller than prior user-level replay system such as PinPlay [19], which more than 80X even for recording an application with a small number of cores. Further, the performance slowdown can be further reduced if the overhead of COREMU/QEMU has been reduced. Actually, recent work over QEMU shows that, by integrating an

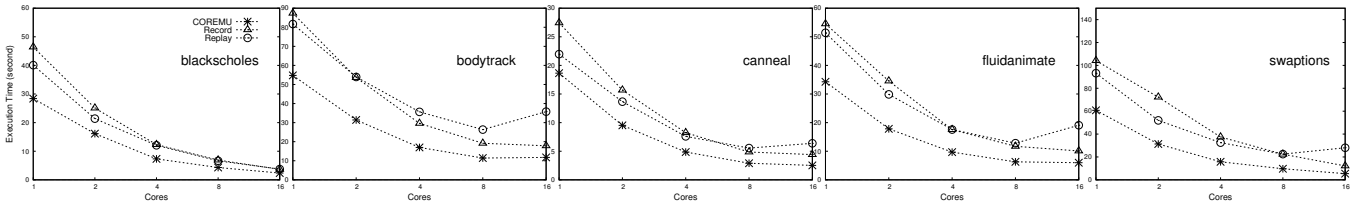

Figure 5. Performance and scalability of the PARSEC benchmark

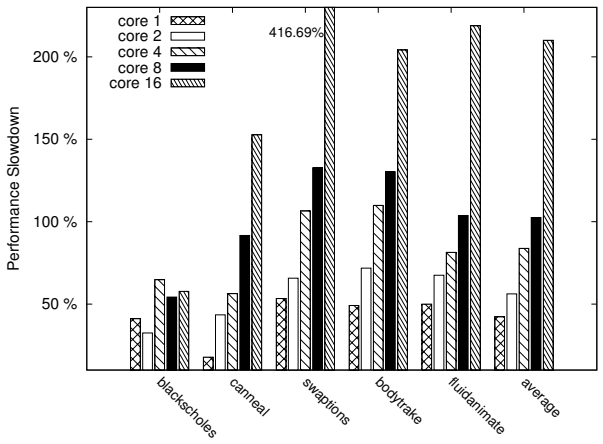

Figure 4. Replay slowdown compared to COREMU

optimized backend compiler (e.g., LLVM) with optimizations, the performance of QEMU can be reduced by 2.4X to 4X. With a more optimized DBT, the overhead of ReEmu can be similarly reduced.

5.3 Benefits of Performance Optimization

Compared with Prior CREW Protocols: To compare the CREW protocol in ReEmu with prior ones, we implement the CREW protocol in SMP-Revirt [8] in ReEmu with a few changes: 1) retrieving remote core information for logging is done by directly accessing the fields in remote cores, instead of sending IPIs; 2) a read-write lock is used to guarantee that the remote core information is unchanged, instead of using a system-wide lock (i.e., shadow_lock). We tried to implement the original CREW protocol in Instant Replay [14], but it turns to be too slow to boot Linux.

ReEmu with this CREW protocol takes around 122.6s, 343.6s 912.3s 1343.5s to record the execution of blackscholes on 1, 2, 4 and 8 cores. When using 16 cores, the system runs too slowly that it still fails to boot Linux after 1 hour. In contrast to the CREW protocol in ReEmu, it has significantly worse performance and scalability.

Benefits of Lock Clustering: Due to space constraints, we present three applications that are with performance speedup and slowdown due to the lock clustering algorithm. As shown in Table 2, the execution time of swaptions is reduced from 27.3% to 19.2% when the number of cores increased from 1 to 16. When there is contention on an object, lock clustering will not be applied for that object. This explains why the improvement is reduced when the number of cores increases. blackscholes also gets some speedup with lock clustering, but not that much as swaptions. canneal is the only tested application with performance slowdown with lock clustering. Though lock clustering can avoid repeatedly acquiring the same lock, it needs extra work like recording which locks are hold by self and releasing hold locks when there is contention. To avoid holding too much locks and hurt performance when there's contention on many shared object, each emulated core can hold at most 32 locks. When this number increases to 64, canneal will also show a little speedup running 1 emulated core, but more slowdown when running more emulated cores.

app	#core	w/o opt	with opt	reduction
swaptions	1	143.38	104.18	27.3%
	2	75.10	59.42	20.9%
	4	38.56	30.76	20.2%
	8	24.23	18.39	24.1%
	16	13.08	10.57	19.2%
blackscholes	1	52.57	46.43	11.7%
	2	27.28	25.08	8.1%
	4	13.72	12.30	10.3%
	8	8.12	6.90	15.0%
	16	4.02	3.67	8.5%
canneal	1	26.92	27.44	-1.9%
	2	14.56	15.68	-7.7%
	4	7.61	8.24	-8.4%
	8	4.56	4.86	-6.6%
	16	4.02	4.41	-9.9%

Table 2. Optimization effect of lock clustering

5.4 Impact of Object Size

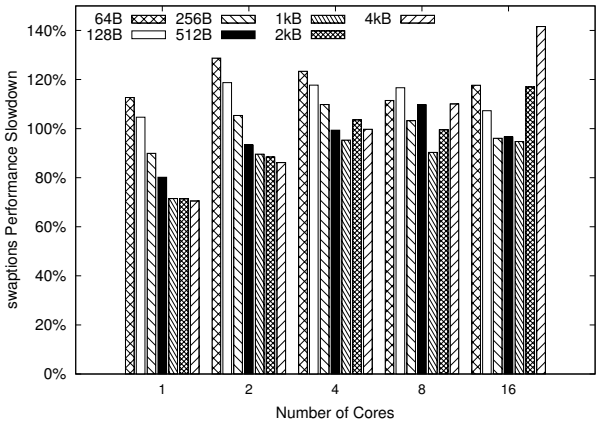

Figure 6. Impact of memory chunk size (record performance slowdown compared to COREMU)

Figure 6 shows the performance impact of different memory chunk sizes for the swaptions application. (swaptions has many shared memory operations thus the impact would be bigger.) This result is tested with lock clustering enabled. For 1 core configuration, a larger memory chunk size reduces the number of lock operations thus setting it to 4K has the smallest slowdown (70.5%),

while 64byte has the largest slowdown (112.7%). For 16-core configuration, as there are more concurrent accesses to the same page, the performance of 4 Kbyte memory chunk size degrades, and the slowdown increases to 141.6% (while the lowest slowdown is 94.7%). To get a consistent performance, we set the default memory chunk size to 1 KByte, which shows good performance from small number of of emulated cores to larger number of emulated cores. We will apply an adaptive approach by assigning the chunk size according to the number of cores in future.

5.5 Log Size

To collect log size for an application, we modified ReEmu to start take log after the timing backdoor has triggered. So the data here are only for the log taken when executing region of interest.

Table 3 shows low size for all the benchmarks applications and Linux kernel boot. For single core recording, as no memory ordering needs to be logged, the log size is very small. The log size grows as number of cores grows. The log size for bodytrack and swaptions grows much slower than other applications. These 2 applications has small working sets, they touch less memory chunks and thus less memory ordering logs are required. While canneal and fluidanimate has very large working set, as the number of cores grow to 16, the log size grows to more than 20MB. blackscholes' working set size grows with core number thus the log size also grows a lot.

Note that working set size is not an accurate indicator of log size. Memory access pattern is also a deciding factor to log size. In the worst case, two cores writing to a single shared object in turn, every write needs to take log. So even for working set as small as a single page, the runtime overhead and log size could still be big.

application	1 core	2 cores	16 cores
blackscholes	123K	398K	18M
canneal	346K	1.5M	26M
swaptions	67K	149K	2.3M
bodytrack	12K	209K	2.7M
fluidanimate	54K	687K	24M
kernel boot	1.2M	1.5M	6M

Table 3. Log size of ReEmu (compressed with gzip)

5.6 ARM Performance

We ported one MapReduce application in the Phoenix test suite [20] to ARM platform to study the performance and scalability of ReEmu on ARM record and replay. The guest version of Linux is 2.6.28 which is provided by ARM corporation. We tested WordCount with 10 Mbyte file running on a 1, 2, 4 core configuration[4]. The average slowdowns caused by ReEmu in record and replay run are 2.1X, 1.9X, 1.9X and 3.3X, 3.4X, 3.8X accordingly. It also exhibits good performance scalability: the execution time is 32.1s, 12.6s, 5.8s and 50.5s, 21.8s, 11.4s to record and replay WordCount under 1, 2, and 4 core configuration accordingly.

5.7 Bug Reproducibility

To evaluate the bug reproducibility of ReEmu, we record and replay the racy benchmark [23] on ReEmu. We validate the signature generated from record and replay runs and find that the signature is the same. Hence, ReEmu can faithfully reproduce concurrency bugs.

[4] The maximum number of cores supported in ARM Cortex-A9 is 4.

6. Related Work

Replaying in the Virtual World: There has been some prior work in supporting execution replay of virtual machines. Bressoud et al. [4] are the first to provide execution replay for virtual machines, but their techniques target at single-core VM and are used for fault-tolerance in between a primary machine and the backup. Revirt [7] logs external inputs and source of non-determinism in a single-core VM and replays the execution mainly for intrusion analysis. The time-travel VM [11] and ReTrace [24] leverages execution replay of single-core virtual machines to debug operating systems and collect execution trace respectively, which can also be similarly applied to ReEmu. SMP-Revirt [8] is the first to provide execution replay of multiprocessor VMs by leveraging the CREW protocol [5] to record orders of shared memory accesses. ReEmu also leverages the CREW protocol, but with notable refinements that enable ReEmu to be scalable and efficient to replay multicore systems.

Leap [10] and ORDER [25] are two recent systems supporting deterministic replay in Java virtual machines (JVMs). Like ReEmu, replaying in JVM can do fine-grained instrumentation and track memory accesses in fine-granularity. For example, ORDER takes an object-centric approach that records shared memory accesses in object-level. However, execution in a full-system emulator has no such good locality with objects, for which reasons ReEmu tracks memory objects at a fixed size.

Replaying Natively in the User Land: Similarly to ReEmu, Scribe [12] also leverages the CREW protocol to record shared memory accesses. However, Scribe defers page ownership transition at sync points (e.g., system calls), which is not appealing for ReEmu as the delayed memory access interleaving may hide many data races that occurs in a real machine. Compared to Scribe, the CREW protocol in ReEmu has been refined to be scalable and efficient by precise and scalable tracking of access orders to shared memory objects. Finally, tracking memory accesses in page granularity in Scribe may suffer from false sharing, while ReEmu is more flexible in choosing tracing granularity. DoublePlay takes an approach called uniparallelism that uses serialized execution to check the memory ordering of parallel execution. However, uniparallelism may be hard to be efficient for a full-system emulator, which may cause frequent rollback and it is not easily to efficiently rollback the execution of a full-system stack.

There are also efforts in trading determinism and recording overhead. ODR [1] reduces the overhead by only guaranteeing output deterministic at the benefit of ignoring the outcomes of data-races, while takes more time and space to search the execution path during replay to ensure output-determinism. PRES [18] also only records a "sketches" during record run, and instead leverages the replayer to explore possible execution spaces to reproduce bugs. Respec [15] combines speculative execution for logging and replies on the replayer to detect and recover if the replay session diverges. There techniques should also be able to be integrated into ReEmu to further reduce the overhead of ReEmu, which will be our future work.

7. Conclusion and Future Work

In this paper, we made the first attempt to provide deterministic replay features to parallel full-system emulators. Based on a comphrensive analysis on issues with prior CREW protocols, we make several refinements that made ReEmu scalable on multicore platforms and efficient. Evaluation showed that ReEmu incurs modest runtime and space overhead and can faithfully replay the whole software stack.

There are still plentiful research opportunities for us to explore in future. First, we plan to incorporate debugging and analysis tools into ReEmu so that it can seamlessly work with existing tools like gdb. Second, we will study the performance and scalability of

our refined CREW protocol to record and replay user-level system such as user-mode QEMU and Pin, which should bennefit from the scalable design. Third, as we currently only evaluate ReEmu on a small amount of cores, we plan to study the performance and scalability of ReEmu on a machine with hundreds of cores.

References

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multi-core debugging. In *Proc. SOSP*, 2009.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proc. USENIX ATC*, 2005.

[3] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. PACT*, 2008.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proc. SOSP*, 1995.

[5] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Comm. of the ACM*, 14(10):667–668, 1971.

[6] J. Ding, P. Chang, W. Hsu, and Y. Chung. PQEMU: A parallel system emulator based on QEMU. In *Proc. ICPADS*, 2011.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI*, 2002.

[8] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proc. VEE*, 2008.

[9] D. Hong, C. Hsu, P. Yew, J. Wu, W. Hsu, P. Liu, C. Wang, and Y. Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proc. CGO*, 2012.

[10] J. Huang, P. Liu, and C. Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *Proc. SIGSOFT FSE*, 2010.

[11] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX ATC*, 2005.

[12] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. SIGMETRICS*, 2010.

[13] R. Lantz. *Parallel SimOS - Performance and Scalability for Large System*. PhD thesis, Stanford University, 2007.

[14] T. Leblanc and J. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *Computers, IEEE Transactions on Computers*, C-36(4):471–482, 1987.

[15] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Proc. ASPLOS*, 2010.

[16] M. McLoughlin. The qcow image format, 2008.

[17] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proc. ISCA*, 2005.

[18] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proc. SOSP*, 2009.

[19] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proc. CGO*, 2010.

[20] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. HPCA*, pages 13–24, 2007.

[21] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *Proc. ASPLOS*, 2011.

[22] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, Z. W., and B. Zang. Coremu: a scalable and portable parallel full-system emulator. In *Proc. PPoPP*, 2011.

[23] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proc. ISCA*, 2003.

[24] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[25] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: object centric deterministic replay for java. In *Proc. USENIX ATC*, 2011.