COREMU: A Scalable and Portable Parallel Full-system Emulator

Zhaoguo Wang Ran Liu Yufei Chen Xi Wu Haibo Chen Weihua Zhang Binyu Zang

Parallel Processing Institute, Fudan University

{zgwang, ranliu, chenyufei, wuxi, hbchen, zhangweihua, byzang}@fudan.edu.cn

Abstract

This paper presents the open-source COREMU, a scalable and portable parallel emulation framework that decouples the complexity of parallelizing full-system emulators from building a mature sequential one. The key observation is that CPU cores and devices in current (and likely future) multiprocessors are loosely-coupled and communicate through well-defined interfaces. Based on this observation, COREMU emulates multiple cores by creating multiple instances of existing sequential emulators, and uses a thin library layer to handle the inter-core and device communication and synchronization, to maintain a consistent view of system resources. COREMU also incorporates lightweight memory transactions, feedback-directed scheduling, lazy code invalidation and adaptive signal control to provide scalable performance. To make COREMU useful in practice, we also provide some preliminary tools and APIs that can help programmers to diagnose performance problems and (concurrency) bugs.

A working prototype, which reuses the widely-used QEMU as the sequential emulator, is with only 2500 lines of code (LOCs) changes to QEMU. It currently supports x64 and ARM platforms, and can emulates up to 255 [1] cores running commodity OSes with practical performance, while QEMU cannot scale above 32 cores. A set of performance evaluation against QEMU indicates that, COREMU has negligible uniprocessor emulation overhead, performs and scales significantly better than QEMU. We also show how COREMU could be used to diagnose performance problems and concurrency bugs of both OS kernel and parallel applications.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Modeling techniques; I.6.0 [*Simulation and Modelling*]: General; D.2.5 [*Testing and Debugging*]: Debugging aids

General Terms Design, Experimentation, Performance

Keywords Full-system Emulator, Parallel Emulator, Multicore

1. Introduction

The continuity of the Moore's Law has shifted the current computing to multicore or many-core eras. Currently, eight cores and twelve cores on a Chip are commercially available. It was predicated that tens to hundreds (and even thousands) of cores on a single chip would appear in the foreseeable future [31].

The advances of many-core hardware also make full-system emulation more important than before, due to the increasing need of pre-hardware development of system software, characterizing performance bottlenecks, exposing and analyzing software bugs (especially concurrent ones). Full-system emulation, which emulates the entire software stack including operating systems, libraries and user-level applications, is extremely useful in serving the above purposes. It is even claimed with evidence that simulators might be inaccurate or even useless if ignoring the system effects [7]. In light of the importance of full-system emulation, there has been a considerable amount of effort to build efficient full-system emulators. Examples include QEMU [21], Bochs [5], *Simics*[15] and Parallel Embra [14].

The many-core or multicore computing also creates tremendous challenges and opportunities to full-system emulation. On one hand, the rapid-increasing number of emulated cores requires full-system emulation to be scalable and be able to handle a reasonable scale of input. On the other hand, the abundant physical cores provide even more resources for full-system emulators to harness.

Unfortunately, most commodity full-system emulators are sequential and only time-slice emulated cores on a single physical core in a round-robin fashion [15, 16, 21, 23], or only support discontinued outdated host and guest processor pairs [14]. Hence, they cannot fully harness the power of likely abundant resources in current CMP architecture, resulting in poor performance *scalability* and *restricted parallelism*.

First, the sequential emulation design indicates linear slowdown when the number of emulated cores grows, thus scales poorly on current multicore platforms. Figure 1 shows the average execution time of processing 10 MB and 100MB input using *WordCount* in log scale, a MapReduce application for shared-memory multiprocessors in the Phoenix testsuite [22], running on an emulated Debian-Linux with kernel version 2.6.33-1 using the recent version of QEMU. The performance degrades linearly with the number of cores. When processing relative large input (e.g., 100MB), QEMU times out for only 32 emulated cores.

Second, the sequential design implies that there is limited parallelism exposed among emulated cores. This significantly restricts the use of full-system emulator to analyze software behaviors, thus sacrifices the fidelity of full-system emulation. This problem is critical as parallelism is crucial to exhibit bugs when running parallel workloads or debugging system software, which are especially important due to the pervasive existence of parallelism and the difficulty in writing correct parallel code.

Figure 2 shows the restricted parallelism problem using a simple *parallel counter* program. The program increases the *counter* using two parallel threads, with each thread increase it 500 times.

[1] The xAPIC specification in x86 supports up to 255 cores.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

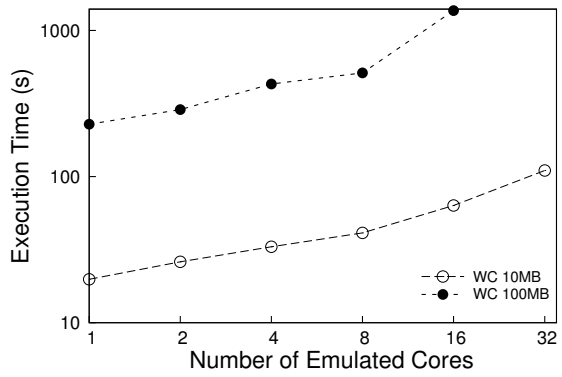

Figure 1: The execution time of WordCount processing 10 MB and 100MB input on QEMU running on a 16-core machine.

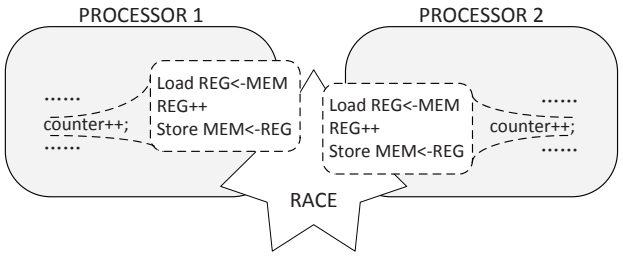

Figure 2: `Bad Counter`: the result will be incorrect as *INC* will not be executed atomically.

The expected output should be 1000 but it fails on a multicore system due to the data race between two threads. Unfortunately, *this program usually behaves correctly* on *full-system emulators that sequentially emulate multicore*. This is because the scheduling among cores only happens at a coarse-grained granularity (e.g., basic block) for the sake of performance, which naturally renders the *increment* atomic, resulting in no data race. In this example, a much subtler problem occurs on a CISC machine where the increment is translated to a single *INC* instruction *without lock prefix*. In such a case, the data race can only appear at *microinstruction* level, hence even scheduling at instruction granularity can hardly expose the data race bug.

Unfortunately, building a *parallel* full-system emulator is usually resource-intensive and requires years to be mature. Full-system emulators, unlike user-mode emulators, need to model the system aspects of a computing platform, including system-ISA, address translation, privilege levels, interrupts and a set of devices. Further, building a portable emulator is even harder because of the dramatic differences of both the user-ISA and system-ISA among diverse architectures. For example, QEMU becomes mature and widely adopted after years of active development and is currently still in active evolvement.

To address the difficulty of building a portable parallel emulator, this paper presents COREMU, a parallel emulation framework for CMP systems that decouples the complexity of supporting parallel emulation from maturing a sequential emulator. The key observation is that CPU cores and devices in current (and likely future) multiprocessors and multicore are loosely-coupled and these cores and devices communicate through well-defined interfaces. Based on this observation, COREMU emulates multiple cores by creating multiple instances of existing sequential emulators, and uses a thin library layer to handle the inter-core and device communication and synchronization, to maintain a consistent view of system resources.

To ensure correctness and provide scalable performance, COREMU also incorporates several techniques to enable efficient parallel emulation. First, efficient and portable *core-to-core synchronization* is achieved through *lightweight memory transactions*, with the only assumption that the host architecture supports compare and swap (*CAS*) primitives [2], and allows the reuse of existing code generation for sequential emulation. Second, COREMU is built with a workload-aware feedback-directed scheduling that avoids situations such as lock-holder preemption and allows balanced scheduling of emulated cores. Third, to improve the scalability of code cache management, COREMU uses a private code cache scheme and addresses the issues with excessive inter-core cache eviction through *lazy cache invalidation*. Finally, efficient *core-to-core communication* is implemented through non-blocking data structures and real-time signals with *adaptive signal control*.

To make COREMU useful in practice, we also provide some preliminary tools and APIs to enable programmers to diagnose performance bottlenecks and bugs of both OS kernels and applications. First, COREMU provides a set of APIs including dynamically instrumenting programs and watching the accesses to user-specified addresses. Second, programmers could also use COREMU to collect memory traces of a program, which could be fed into a cache simulator (e.g., GEMS [17]) to study the cache behavior of a program.

We have built a working prototype that fully supports the x64 and ARM platform, in the form of a thin library composed of 2700 lines of code (LOCs) with only about 2500 LOCs changes to QEMU. Application benchmarks (such as MapReduce, PARSEC, dbench and Kernel Build) show that COREMU scales much better than QEMU: COREMU can emulate up to 255 cores of x64 architecture and 4 cores of ARM platform [3] while QEMU either fails to boot or times out when only emulating 32 cores. Compared to QEMU, the uniprocessor emulation overhead measured by SPECINT-2000 shows negligible performance penalty (within 1%) incurred by the parallel emulation. However, it achieves more than 20X speedup when emulating 16 cores compared to QEMU. The performance benefit is due to the fact that COREMU can leverage multiple available caches on multicore system and increased parallelism.

To demonstrate the usefulness of COREMU, we also use COREMU to debug both Linux kernel and user applications. Our study shows that COREMU can provide precise evidences to uncover several bugs. Cache simulation results using the typical matrix multiply show that COREMU can also accurately observe the cache behavior of a program.

In summary, this paper makes the following contributions:

1. A case for parallelizing full system emulators by reusing existing mature sequential emulator, which decouples the complexity of supporting parallel emulation from constructing and optimizing a sequential emulator.
2. A set of techniques to scale COREMU: lightweight memory transactions to achieve efficient synchronization among cores, feedback-directed scheduling, private code cache with lazy cache invalidation and adaptive signal control for scalable communication.
3. Implementation, evaluation and case studies of COREMU, which demonstrate the performance scalability and usefulness of COREMU.

[2] Other primitives such as *ll/sc* are similar to CAS.

[3] The Cortex-A9 MPCore supports a maximum of 4 cores

The rest of the paper is organized as follows: Section 2 provides background information on full-system emulation and relates COREMU to previous approaches. Section 3 and 4 presents the design and implementation of COREMU. Section 5 evaluates COREMU using various benchmarks. Section 6 demonstrates the usefulness of COREMU using several case studies. Finally, section 7 discusses future work and concludes.

2. Background and Related Work

In this section, we first use QEMU as an example to describe key techniques used in full-system emulation based on dynamic binary translation and then relate COREMU to previous work.

2.1 Full-System Emulation with Binary Translation

Binary translation: The main loop of QEMU translates and executes the emulated code based on basic blocks. Each block has one entry and one exit point and is sequentially executed. QEMU first translates the target machine code into common intermediate code, which is recognized by its *Tiny Code Generator* (*TCG*).

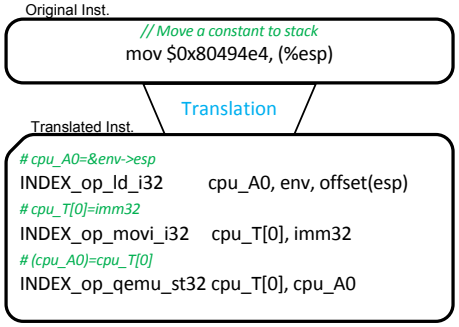

Figure 3: An example binary translation of a *mov* instruction in QEMU.

Figure 3 shows the generated operation code of each micro operation, along with its required parameters, when translating a *mov* instruction. The first item of each line represents the operation code (opcode), which is stored in `gen_opc_ptr` buffer. Other items give the parameters which are stored in `gen_opparam_ptr` buffer. The comment before each line shows the semantics of the line. Here, *cpu_A0* and *cpu_T[0]* are temporary registers allocated through *TCG*. The imm32 is $0x8049424 in this case, and QEMU gets the immediate when disassembling the binary code. After register allocation and machine code generation, the intermediate code will finally be emitted as binary code.

Multiprocessor emulation: QEMU emulates multiprocessor in a round-robin fashion: each emulated core has a time slice to execute, and yields the physical CPU to the next emulated core if the time slice has exhausted. Hence, there is no need to emulate atomic instructions, as the scheduling at basic blocks naturally guarantees the atomic execution of each instruction. Further, the time-slicing of cores will result in bad performance scalability due to the serialization of parallel code. For example, when a core holding a lock has exhausted its time slice, other cores waiting on such a lock will waste their time slices on spin-waiting the lock.

Full-system emulation support: Device emulation is done by providing port and memory mapped I/O callback functions. QEMU implements asynchronous I/O access, such as DMA, through signal mechanism and pipe. Interrupts are handled by setting vector bits in emulated interrupt controller. Before searching and executing translation blocks, QEMU peeks the interrupt controller to see if an interrupt presents, and emulates the interrupt handling procedure accordingly. This typically includes changing privilege level and pointing the emulated program counter to the entry of the interrupt handler.

To support full-system emulation, QEMU implements a soft-MMU that translates the target virtual address to host virtual address. QEMU uses a soft-TLB to speedup target address translation. Soft-TLB caches address results in the same way as the hardware TLB. QEMU places look-up code before the code calling the soft-MMU callback.

Translation cache management: QEMU uses a single process to do system emulation and uses a single global translation cache. The central data structure used to manage translated code is `TranslationBlock`, which contains a pointer to the start of a translated block in the cache. Given an emulated program counter (target virtual address), QEMU checks whether its corresponding code has been translated and in the cache and translates the code and puts them into the cache upon a miss.

Physical pages containing translated code (code pages) are write-protected by QEMU to maintain code cache consistency. Specifically, the soft-TLB entries that point to code pages are marked as clean so that any modifications to the page will trap to a callback function, which invalidates the translation block by deleting the corresponding items from the virtual and physical tag hash tables.

2.2 Related Work

2.2.1 Full-system Emulation

The most related work with COREMU is Parallel Embra [14], which extends the original Embra emulator [29]. However, it was designed to support the SGI Origin 2000 with MIPS R10000 processors, which are out-of-date (15 years ago) and not commercially available now. In a contrast, COREMU runs current prevalent Chip-multiprocessors (e.g., x64) and supports emulation of multiple contemporary processors (e.g., x64 and ARM). To decouple the complexity of parallelism from binary translation, it adopts a layered structure and is likely to be easily re-targeted to new host-target architecture pairs. Further, COREMU uses a general and unified approach to handling atomic instruction emulation for weak ordering adopted by modern processors such as x64 and ARM, while Parallel Embra only emulates MIPS processor with a sequential consistency model. Finally, COREMU introduces several new techniques absent in Parallel Embra, such as synchronization with lightweight memory transactions, feedback-directed scheduling and thread-private cache with lazy invalidation. These techniques make COREMU able to run contemporary workloads with reasonable performance.

QEMU/KVM [2] is an accelerator that uses a virtualization layer to accelerate emulation on the same-ISA platform. However, the use of system virtualization loses both portability (e.g., only x86 to x86) and flexibility (e.g., no instrumentation support). Moreover, it requires hardware support.

Sulima [20] and Parallel Mambo [26] (which extends Mambo [6] for PowerPC) are two parallel full system emulators. They achieve parallel emulating through parallelizing the original sequential emulators. However, COREMU adopts a different, *core-per-thread*, organizing strategy and reuses mature sequential emulator for extensible cross-platform emulation. There are several user-level parallel emulators developed (for example, [19, 33]). Similar to COREMU, Graphite [19] provides user-level parallel functional simulation also using a multicore-on-multicore model. Instead of emulating only user-level applications, COREMU focuses on full-system emulation.

2.2.2 Simulation

There has been much research work devoted to fast and faithful simulation of multiprocessors. Broadly speaking, these work can

be categorized to software approaches and hardware-assisted simulations. We discuss their relationships with COREMU in turn.

Software Approaches PTLsim [32] simulates x64 processor using a Virtual Machine Monitor (VMM). By contrast, COREMU supports cross-platform emulation, and uses binary translation for emulation, which provides flexibility such as instrumentation capability and statistics generation.

Most of the software approaches exploit tradeoff between speed and detail to achieve significant speedup, for example, *Statistical simulation* [28], *DiST* [30], AMD SimNow [3] and GEMS [17]. Compared to these systems, COREMU introduces non-determinism and leverages the true parallelism in the underlying processors to accelerate parallel full-system emulation.

Hardware-assisted Simulation The RAMP project from Berkeley investigates the use of FPGA to accelerate simulation of the CMP architecture. Specially, the RAMP Blue [27] models the future architectural features such as message-switching and transactional memory.

ProtoFlex [8, 9] is a hybrid functional emulator that uses FPGAs to accelerate performance-critical parts in emulation. The proposed technique, called transplanting, dynamically selects hot-traces to be emulated in FPGAs, while leaving uncommon traces being emulated in CPUs.

Recently, Chung et al. [10] proposed an approach to solve the emulation of atomic instruction using hardware support for transactional memory. In contrast, COREMU uses a more lightweight solution that only requires compare-and-swap support of the underlying processors, which is readily available on commodity processors.

Compared to these approaches, COREMU exploits abundant multicore resources for full-system emulation, which achieves reasonable performance without special hardware and is easy to deploy and use.

3. The COREMU Parallel Full-system Emulator

This section identifies the challenges in building a scalable parallel full-system emulator, shows the overall architecture of COREMU and presents the solutions to the identified challenges. Finally, we also present the preliminary support in COREMU for debugging and performance diagnosis.

3.1 Challenges in Building a Scalable Parallel Emulator

Compared to building a sequential emulator, there are several challenges in designing and implementing a scalable parallel emulator, due to the inherent differences during execution (i.e., time slicing vs. true parallelism). Here, we identify the key issues in building such an emulator for contemporary CMP architecture:

- *Atomic Instructions:* Unlike sequential emulators that inherently handles atomic instructions by scheduling at the basic-block level, a parallel emulator needs to efficiently emulate synchronization primitives to coordinate concurrent accesses to the emulated shared memory from each emulated core.
- *Scheduling Support:* To emulate a large number of cores with practical performance, it is critical to understand the workload behavior to schedule the emulated cores. For example, lock-holder preemption [25] can easily consume the available limited CPUs, resulting in extremely bad performance.
- *Scalable Code Cache Management:* There could be intensive contentions if adopting a shared code cache as that in sequential emulator when emulating a large number of virtual cores. Thus, an efficient code cache scheme is critical for the performance of parallel emulator.
- *Scalable Communications:* When emulating CPU cores in the scale of hundreds and even thousands, there could be easily excessive core-to-core and core-to-device messages, due to device interrupts (e.g., DMA, timer interrupts) and interprocessor interrupts (such as remote TLB shootdowns). Hence, it is likely that an emulated core could be frequently disturbed for processing messages, limiting its performance.

3.2 Overall Architecture

COREMU is designed based on the observation that cores in modern multicore or multiprocessor machines are loosely-coupled and communicated with well-defined interfaces. For example, each core has its own register file, control logic and separate cache. They independently execute instruction stream assigned to it and the communication channels between cores are *well defined*, such as Inter Processor Interrupt (`IPI`). Such an organization allows the separation of building fine-tuned sequential emulators from efficiently parallelizing it, thus decreases the complexity of building a parallel full-system emulator. It could be much easier to adapt the emulator to different host/target pairs to make such an emulator portable.

Figure 4 depicts the architecture of COREMU. Overall, COREMU is a multithreaded user program running on the hosted operating systems, emulating a cache-coherent shared memory multiprocessor to run operating systems and the applications. Each sequential emulator is essentially a threaded binary translator with its own translation cache holding already translated blocks (TBs). All devices are emulated using a separate thread. There is a thin library layer handling communications and synchronizations among emulated cores and devices through intercepting callouts. The library also maintains the coherence between each translation cache by coordinating the invalidation requests.

Atomic Instructions	Lightweight Memory Transactions
Scheduling	Feedback-Directed Scheduling
Code Cache	Private Cache w/ Lazy Inval.
Communication	Adaptive Signal Control

Table 1: Methodologies in COREMU.

In additional to memory consistency, Table 1 summarizes the underlying techniques in COREMU to enable a scalable parallel emulation of a large number of cores with practical performance, which will be presented in detail in the following sections.

3.2.1 Synchronization with Lightweight Memory Transactions

The fact that all emulated cores share a global, cache-coherent memory poses a challenge to scalable parallel emulation. Specifically, microprocessor exports a set of *atomic* instructions executed atomically, which are usually used to implement synchronization primitives. An efficient emulation of atomic instructions is critical to parallel emulation: (1) the emulation of atomic instruction should be fast and correct; (2) the emulation should be portable across a variety of architectures.

An intuitive solution is to perform an identical translation that maps the emulated atomic instruction to one on the host architecture, which is used in Parallel Embra [14]. This solution is fast and correct, but not portable, due to the idiosyncratic nature of different ISAs. For example, the ARM processor has only 2 atomic instructions, while x86 has around 20, which indicates that it is not always feasible for such a direct mapping. Furthermore, an atomic instruction in an emulated x86 core is usually decomposed into several non-atomic micro-operations on host architecture.

Another intuitive solution is to use *lock* to synchronize all parallel accesses, by associating each memory region with a lock to

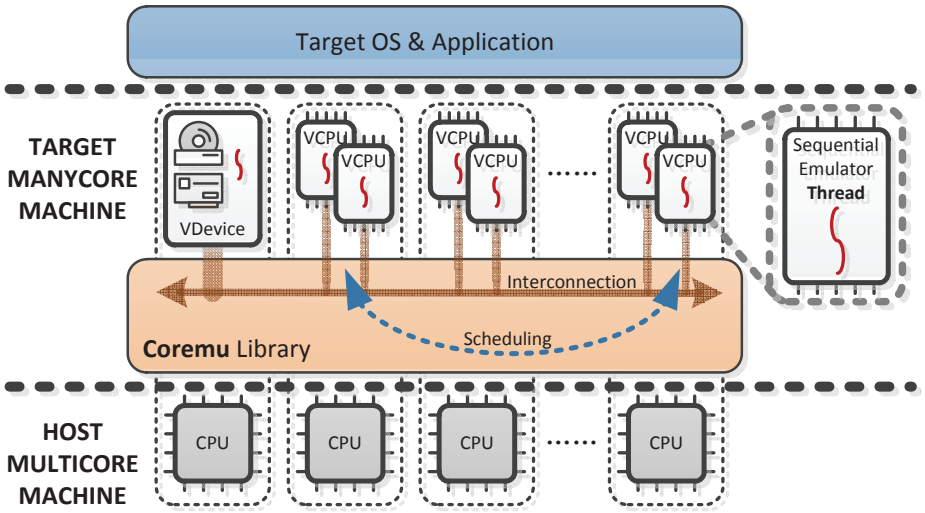

Figure 4: Overall architecture of COREMU, which uses a two-layer parallel emulation organization. COREMU library acts as the bonding agents between different emulated components.

serialize accesses to this region (in our initial implementation, each 64-byte region has a lock). Such a solution avoids a global lock so that parallel accesses to different memory regions are allowed. Unfortunately, while this solution seems plausible, it has correctness issues.

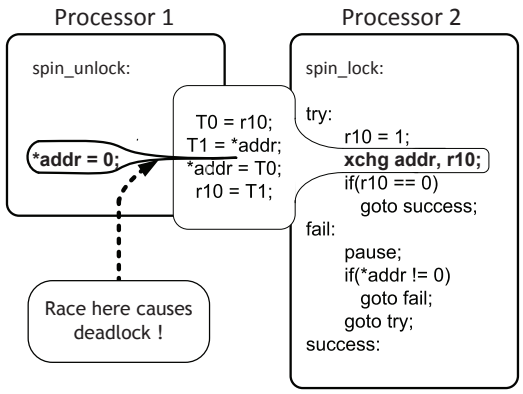

Figure 5: Weak atomicity problem due to partial locking.

For example, consider the deadlock illustrated in Figure 5. This example implements an efficient *spin_lock* and *spin_unlock* operations on Intel/AMD x64 machines. The *spin_unlock* operation simply stores a 0 into a lock. The *spin_lock* exchanges a 1 into the lock using atomic `xchg`, and checks whether the original value is 0. QEMU translates `xchg` by first reading out the two values into temporaries, and then storing them back with swapped order. If we partially protect the exchange with a lock, the store operation in *spin_unlock* can still happen during `xchg`, causing a deadlock afterwards.

Hence, to ensure *strong atomicity*, one must passively protect *all* atomic instructions using lock. However, this solution will incur prohibitive performance overhead as every memory access needs to acquire and release a lock.

COREMU solves the multiprocessor synchronization problem with *lightweight memory transactions* based on the well-known *Multi-Word Compare and Swap* (`CASN`) algorithm [12]. COREMU

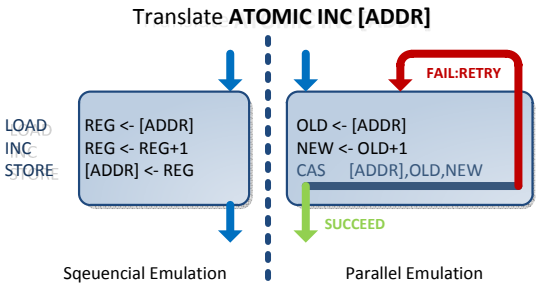

Figure 6: Example translation of an atomic `INC`

supports this operation with the only assumption that the underlying architecture supports `CAS` like synchronization primitives, which holds for most modern architectures. Figure 6 shows an example translation of `INC` instruction where host and target system uses the *same* word size.

Generally, the working flow is as follows: (1) Calculate the result into *tmp*; (2) Use `CAS` to store *tmp* into destination; (3) Proceed to the next instruction on success, or *re-execute* this instruction. Overall, our solution guarantees an efficient emulation of atomic instruction and allows the reuse of most sequential code generation. Further, our memory transactions are much more succinct compared to the general transactional memory due to the simple and clean semantics of instructions. These instructions usually only update a single memory location and the memory state only has one transition, hence very few states are needed to record during such transactions.

To illustrate the effectiveness of COREMU , we start two parallel threads to atomically update a shared counter one million times. The lock-based emulation only uses lock to *partially* protect the emulated `INC`. From our evaluation results COREMU emulation is 8X faster (6.82s vs. 47.69s) than a lock-based solution.

3.2.2 Feedback-Directed Scheduling

COREMU creates one thread for each emulated core or device. There might be more threads than physical cores with the number of emulated cores increasing. Hence, thread scheduling is crit-

ical to ensure practical performance when an excessive number of threads co-exist in the system. To address this problem, we propose a flexible feedback-directed scheduling mechanism to provide good scalability and reasonable fidelity for parallel emulation. The scheduling algorithm aims at utilizing the workload information in the emulated environments at the binary translation layer and uses such information as feedback to direct the scheduling among physical cores:

Lock-holder Preemption: Lock holder preemption is one of the limitations for performance scalability and fidelity for parallel emulating large scale virtual cores. The guest parallel workloads (including the operating system kernel) usually use spin-locks to guarantee exclusive accesses to shared data. Such spin-locks are, by design, only held for a short period of time and will be unlikely preempted until the lock is released.

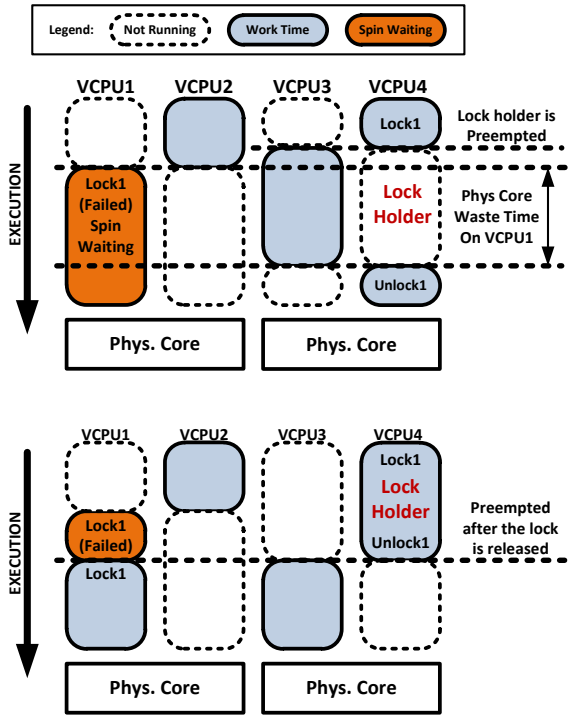

Figure 7: The problem of lock-holder preemption (top half) and the solution in COREMU to remedy the problem (bottom half).

However, when running a commodity operating system on a parallel emulator, the emulated environment may violate such a premise of using spin-locks. The emulated virtual core can be preempted even it is executing critical code protected by a spin-lock, as shown in the top half of Figure 7. Such lock holder preemption could result in a significant increase of the lock holding time. For workloads running on large-scale emulated cores, such a situation could be common and cause a serious performance degradation and poor fidelity.

Avoiding lock holder preemption can be achieved by either modifying (e.g., compiler instrumentation) the guest operating systems or workloads to give hints to the emulator (intrusive), or having the emulation layer to detect when the guest operating systems are not holding a lock (non-intrusive). To retain transparency, COREMU currently uses the latter approach to detecting the spin-locks.

Based on the observation that a spin-lock usually uses a *pause* instruction after the lock prefix instruction, we can pre-translate more translation blocks when we find a lock prefix instruction has been translated. When COREMU detects a spin-lock, it tags the corresponding TBs. Afterwards, COREMU can detect if a virtual core has acquired a spin-lock successfully. To avoid lock holder preemption, COREMU feeds such hints to the underlying scheduler to avoid preemption when a virtual core is still executing a critical section. After the lock is released, COREMU also gives such a hint to the scheduler and the scheduler will decide if re-preempting the virtual cores. Finally, COREMU can also detect if a core is waiting on a lock and feed such a hint to the scheduler, which yields the virtual core until the lock release action is detected, as shown in the bottom half of Figure 7.

3.2.3 Thread-Private Cache with Lazy Invalidation

Like other binary translators, COREMU uses translation cache that caches translated code to improve emulation performance. For a parallel emulator that emulates a number of cores, as each core will access the translation cache, an efficient scheme is vitally important for the performance and scalability.

Basically, there are two design choices: thread-shared cache and thread-private cache. For thread-shared cache, all emulated cores share a single global cache and each piece of code has only one copy in the cache, which is efficient in memory space usage, yet would cause heavy contentions on the cache when emulating a relatively large number of cores or running workloads with poor code locality. For thread-private cache, where each emulated core has its own translation cache, which would result in fewer contentions, yet requires more memory spaces and excessive inter-core communications to maintain cache consistency.

To avoid possible contentions, COREMU uses a thread-private cache scheme because it naturally fits into COREMU's decoupled emulation model, where each core thread independently caches their executed code. However, this design leads to the *code eviction* problem, where a write to a code page must synchronize with other emulated cores to invalidate all cached translation blocks. A typical case for such an invalidation event corresponds to self-modifying code. In full system emulation, this happens more frequently when a code page owned by one process is reused as a data page by another process. For example, after a multi-threaded application exits and its memory pages have been reclaimed by the OS kernel, its pages could be used for holding program data. Unfortunately, such pages might still be in translation cache and other emulated cores are not aware of the status changes of these pages. Consequently, COREMU needs to issue *code eviction* events to other emulated cores to invalidate the code cache. According to our experience, there are typically hundreds of thousands of such events when booting Linux with 4 emulated cores. Furthermore, this number rapidly grows with the increase of emulated cores, which dramatically limits the performance scalability.

To address the scalability problem caused by excessive code evictions, COREMU uses a technique called *lazy invalidation*. Our key observation is: *the invalidated code pages are rarely re-executed later*. Hence, the invalidation could be postponed until the re-execution of stale cached-code (if such code is really self-modifying code or the page will again be used as code page). Specifically, on a code page write, all cached translation blocks are updated to return a value, which indicates that code eviction is needed. If such code is re-executed, COREMU removes the cached block from thread-private translation cache, re-translates and executes it. Lazy invalidation ensures the common case is fast, as the real re-execution of stale code cache implies self-modifying code or reused code, which is rare in practice.

Finally, to implement code page protection similar to QEMU, COREMU uses CAS to implement code page protection. Specifically, the core that maps the code should atomically replace the

soft-TLB entries of other cores to protect the page so that any write afterwards traps into the lazy invalidation callback.

3.2.4 Communication with Adaptive Signal Control

In a sequential emulator, it is easy to handle core-to-core communication and core-to-device communication because it can process these asynchronous events in a synchronous way. For example, to emulate the IPI broadcasting, it simply sets the interrupt vector for each emulated core. However, for a parallel emulator, such a direct modification indicates concurrent or even parallel modifications to internal states of an emulated component. Using locks to provide safe concurrent modification is complicated and time-consuming. Further, it violates the design principle to decouple parallel emulation complexity from optimizing sequential ones.

Asynchronous communication in COREMU is handled using *Real Time Signal* (RT-Signal) [1] and non-blocking data structures. RT-Signal is used as communication primitive for its two useful properties. First, the delivering order is guaranteed to be FIFO, which ensures the fairness of handling the events. Second, the signals with the same type are buffered rather than ignored, thus all asynchronous events will not be lost. To handle asynchronous events from hardware and other cores, each core maintains a non-blocking FIFO queue to hold all these events.

However, naively sending all interrupts using RT-signals would result in excessive signals and cause significant overhead due to the high cost of trapping into and returning from signal handlers. Further, it limits system scalability as the number of interrupts increases rapidly with the increase of cores. For example, our tests indicates that a WordCount application with 10 MB input running on 8 emulated cores with 8 physical cores is even a bit slower than on 4 emulated cores on 4 physical cores (5.78s vs. 5.14s).

To solve this problem, COREMU uses a technique called *adaptive signal control* to reduce the signal-handling overhead by controlling the rate of signal sending. Specifically, COREMU only uses RT-signal to notify the target processor when the number of pending interrupts exceeds a threshold. The threshold is dynamically adjusted in the signal handler according to the frequency of received signals. Otherwise, each emulated core polls for pending interrupts. Using such an optimization, the execution time of WordCount on 8 emulated cores with 8 physical cores decreases from 5.78s to 2.47s.

3.3 Debugging and Diagnosis Support

COREMU is also built with some preliminary mechanisms to assist programmers to debug and diagnose the bugs and performance problems of parallel systems and applications:

Watchpoints: To assist programmers to find memory-related bugs effectively, COREMU is integrated with a watchpoint mechanism that can constantly monitor the accesses to a range of not only virtual addresses but also physical addresses. Programmers can also associate a callback function which will be triggered when a specific type of accesses to a watched address occur. A set of utility functions are also provided to be invoked by the callback function, including dumping the call stack, showing the execution context, showing the content of the stack, which helps programmers to understand the execution context. Programmers could control the execution of the monitoring by specifying the condition that triggers the monitoring, which could save the associated overhead.

Cache Simulation: Cache behavior is critical to program performance. Instead of writing a cache simulator to COREMU, we reuse a state-of-the-art simulator (i.e., GEMS [17]) by collecting memory traces using COREMU and feeding the traces to GEMS. This helps programmers to qualitatively identify the performance problems of some applications and system software.

4. Implementation

COREMU is implemented on x64 processors and currently uses QEMU as the sequential emulator. It is in the form of a multi-threaded application scheduled by the host operating systems. It supports the full system emulation of x64 and ARM processors. The COREMU library only requires around 2700 lines of C code, including the thin synchronization and communication layer and some well-known non-blocking data structures such as Michael and Scott's non-blocking FIFO queue [18].

Portability of COREMU: a Case Study using ARM Given the fully-fledged support for x64 platform, we found it quite easy to port it to other platforms. We chose ARM MPCore as the porting target given the popularity of ARM platform on mobile systems, as well as the readily support of sequential emulation of multiprocessors in QEMU. Two of our developers who are quite familiar with COREMU but are completely new to ARM platform, spend four days to port COREMU for ARM, adding only 150 LOCs.

Parallel Emulator Construction To construct a full-system emulator, we reuse all the high-level abstractions in QEMU, such as devices, processors and interrupt controllers. As we model each processor as a single thread, per-core objects need to be marked with `__thread` specifier. The marking is usually quite straightforward since emulated processor objects are well defined.

The communication interfaces need slight adjustment which typically does not require deep understanding of the internal logic of QEMU. For example, we need to use COREMU interfaces to send I/O requests or interprocessor interrupts, and these interfaces are usually just wrapper functions for original QEMU interfaces. For device emulation, COREMU provides *debug mode* device emulation in case the driver is incorrect. Each emulated device has a lock and the lock is acquired at the entry of its I/O hook functions.

Atomic instruction emulation needs to be adjusted to be aware of their atomicity. This requires inserting calls to COREMU library to use memory transactions. Fortunately, most of the code can still be reused. For example, we can completely reuse all the code in QEMU that generates the decomposed micro operations.

Currently, COREMU allocates a fixed portion of memory for each emulated core as their translation cache. This strategy properly fits into the loose coupling nature of cores, except the disadvantage of linear space overhead with the number of cores. Fortunately, we found a 5 MB cache for each core is enough, even for some large parallel workloads. In future, we plan to implement a two-layer translation cache management scheme that combines the space efficiency of shared cache and the scalability of private cache, to support a larger scale many-core parallel emulation.

COREMU modifies the translated code invalidation callback. Every invalidated cached block is rewritten to just return a value, which indicates if code eviction is needed. Note that, while COREMU needs translated code modification, there is no need to modify the complicated internal states of the emulated core, such as translation block unlink or hash map invalidation. However, this requires a core to see the translation blocks produced by other cores. To achieve such a goal, COREMU maintains a data structure which records, across all cores, all translation blocks on a code page. Further, each translation block is provided with a lock and a flag to indicate whether the block has already been evicted. Upon getting the lock, COREMU checks the flag to see whether the rewriting has been done, hence avoids rewriting the code twice.

When emulating a large number of cores, the excessive time interrupts (i.e., signals) could easily exhaust most of the CPU cycles and starve the user programs. Hence, COREMU adaptively adjusts the rate of delivering time interrupt by emulating an time device (e.g., programmable interval time, PIT) according to the proportion of the number of emulated cores with the physical cores. To associate each emulated core with a signal, COREMU creates a

per-thread local timer which signals the emulated processor based on the thread ID, which is retrieved using *gettid* system call.

5. Evaluation

This section evaluates COREMU by comparing it with QEMU when emulating x64 and ARM MPCore using various contemporary benchmarks that either require a relatively large input size and working set or require relatively long execution time.

5.1 Experimental Setup

All performance evaluation is performed on a 4 Quad-core (1.6 GHZ) Intel x64 system running Debian-Linux with kernel version 2.6.26-2. The guest OS is also a Debian-Linux with kernel version 2.6.33-1 since the kernel version 2.6.26-2 cannot boot when emulated core exceeds 64. The host machine has 32GB memory and the guest is configured with 8 GB memory. We use several different types of applications to study their performance with regard to QEMU and COREMU: (1) SPECINT-2000 [13], a CPU-intensive benchmark; (2) the Canneal benchmark from PARSEC benchmark suite [4] using native input set, whose working set is around 2 GB; (3) WordCount benchmark included in Phoenix MapReduce testsuite for multicore [22], using a 100 MB word file; (4) dbench [24], a file system benchmark; (5) Parallel kernel build, which builds a compacted Linux kernel by specifying the currency level as the number of (emulated) cores.

We ported two MapReduce applications in the Phoenix testsuite to ARM platform to study the performance and scalability of emulated ARM MPCore: Matrix Multiply that multiply two 800 * 800 matrices; WordCount with a 10 MB file. The version of Linux used for emulated ARM is 2.6.28.

QEMU and the one used in COREMU are obtained from its GIT repository on May, 4, 2010. COREMU and QEMU are configured with the same options. All these applications are compiled using gcc-4.3.2. As typical timing mechanism in emulated environments could be inaccurate on a large number of emulated cores, we use *rdtsc* instruction in such cases instead, since such an instruction will read the timestamp registers in the host platform directly in COREMU. For all the tests, we run the test program 5 times to get the average. In many cases, QEMU times out so we omit the results of QEMU.

5.2 Performance of Emulated x64

5.2.1 Uniprocessor Emulation Overhead

Figure 8 depicts the relative performance overhead to native Linux with 8 applications in SPECINT-2000 benchmark suite. As shown in the figure, COREMU incurs negligible performance overhead compared to QEMU, within 1% for all of these benchmarks. Compared to native execution, COREMU is 11X slower on average. The single core emulation overhead mainly comes from the communication among different components and the use of transactions to synchronize multiple cores. However, as the proportion of communications as well as memory transactions is relatively small in single-threaded applications, the incurred performance overhead is negligible.

5.2.2 Performance and Scalability of Emulated x64

We use four applications with diverse characteristics to evaluate the performance and scalability of COREMU on x64. The characteristics of those benchmarks are described as follows.

- **WordCount: Data-parallel applications** The WordCount (wc) benchmark tries to demonstrate that COREMU can also run data-parallel applications with good performance.
- *Canneal: the case of handling large working set:* To demonstrate that COREMU can handle a large working set, we compared the performance and scalability of the Canneal benchmark from the PARSEC benchmark suites with 2GB input size.
- *dbench: Evaluating file system and I/O Performance:* We use dbench to evaluate file system and I/O performance and scalability of COREMU.
- *Parallel kernel build: evaluating complex workload:* Building a Linux Kernel is a relatively complex workload as it involves creating a number of parallel processes to compile the source file.

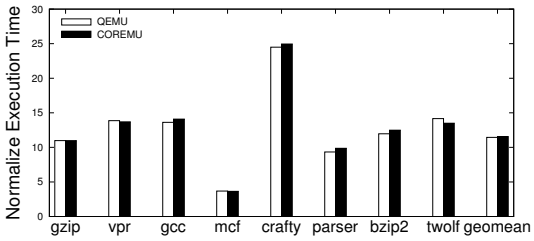

Figure 8: Uniprocessor emulation overhead with SPECINT-2000: the execution time is normalized to native execution time.

To evaluate the performance and scalability, we compare the results of COREMU and those of QEMU with these benchmarks running on 1, 2, 4, 8, 16, 32, 64, 128 and 255 emulated cores. The results are shown in Figure 9, Figure 10, Figure 11 and Figure 12 in log scale. To illustrate the relative performance of COREMU, the native execution time of different applications are also presented.

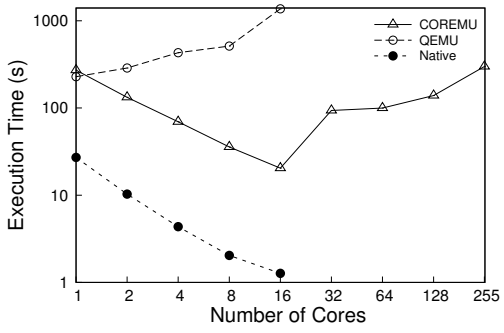

Figure 9: Performance and scalability of the WordCount benchmark with 100 MB input in log scale.

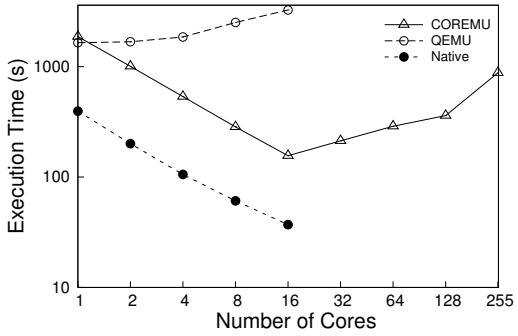

Figure 10: Performance and Scalability with Canneal benchmark from PARSEC testsuite in log scale.

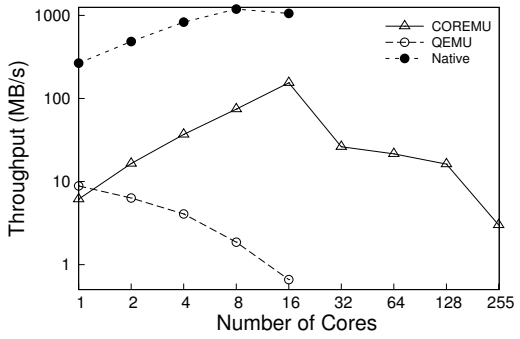

Figure 11: I/O performance results with dbench in log scale.

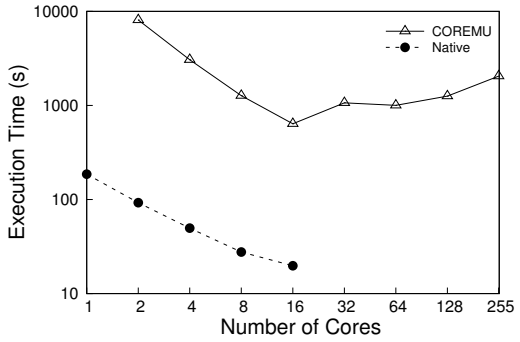

Figure 12: Performance and Scalability of the parallel kernel build benchmark in log scale.

As shown in the figures, COREMU achieves good performance scalability. It scales pretty well when the number of emulated cores increases from 1 to 16. When the emulated cores is larger than physical cores, the performance degradation is still acceptable. It can emulate up to 255 virtual cores with reasonable performance. In contrast, QEMU times out or crashes when emulating more than 16 cores due to cache thrashing or contentions. COREMU achieves a speedup from 20X to 67X when 16 virtual cores are emulated.

There are two reasons for the much better performance and scalability of COREMU. First, when emulating larger number of multiprocessors, there are a lot of synchronizations among threads, and threads frequently fall into spin-wait state. Sequential emulation can only exhaust the time slice of spin-wait. In COREMU, the feedback-directed scheduling can handle this case by detecting lock situation in emulated cores and yielding the control to another thread. Actually, during our process of development and optimization, COREMU can only emulate 32 cores without the feedback-directed scheduling optimization. Second. with the number of virtual core increasing, the implementation of COREMU can also lead to a good data locality and better usage of cache compared to that of QEMU.

As the four mentioned optimizations work together to make COREMU scale beyond 32 cores, we currently failed to identify the contribution of each optimization to the overall performance scalability, which will be our future work.

5.3 Performance of Emulated ARM

We use two applications, Matrix Multiply and WordCount to study the performance and scalability of emulated 1, 2, 3 and 4 ARM cores. Figure 13 shows the performance of the two applications running on QEMU and COREMU. As we currently do not have an ARM MPCore machine in hand, we omit the native data here. Like the performance trend of emulating x64, COREMU has similar performance with QEMU when emulating uniprocessor, but has better performance and scalability when emulating 2 to 4 cores, with the corresponding speedup of 1.67X (11.3s vs. 18.9s), 2.56X (7.2 vs. 16.25s) and 2.5X (6.1s vs. 15.5s) for WordCount and 1.96X (46s vs. 90.5s), 2.9X (30.96s vs. 90.85s) and 4.3X (22.9s vs. 91.17s) for Matrix Multiply accordingly.

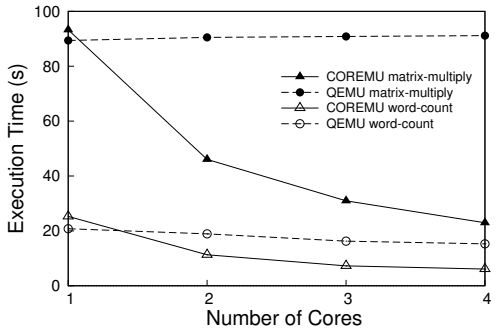

Figure 13: Performance results with Matrix Multiply and Word-Count on QEMU and COREMU.

6. Case Studies of COREMU

To demonstrate the effectiveness of COREMU, we make several case studies by using COREMU to diagnose and debug the performance problems and (concurrency) bugs in both OS kernel and user applications.

Cache Simulation: We use the matrix multiply (mm) application from the Phoenix MapReduce framework [22], which is a parallel version of matrix multiply written using the MapReduce programming model. By collecting the memory traces using COREMU and replaying them in GEMS, we found that even when using 4 cores, the default version in COREMU incurs more than 26% L1 cache miss rate with the input size to be *500 X 500*. By transposing the input matrices before executing the MapReduce tasks, we observe that the L1 cache miss rate degrades to only 5%, leading to a performance speedup of more than 2X.

Debugging using Watchpoints: We use one kernel bug and one user bug to demonstrate how COREMU could be used for debugging. The kernel bug is a NULL pointer dereference bug caused by incorrect concurrent updates to the *inode->i_pipe* variable in Linux kernel version 2.6.21 [4]. After one thread has freed *inode->i_pipe* and set it to NULL, another thread tries to deference it. To detect such a bug, we inserted a watchpoint on updates to that variable and log accesses that write a NULL to that variable. Using COREMU, we quickly located the function and execution context nullifying that variable.

The user-level bug is from pbzip2 [5], which is an order violation concurrency bug. There are still accesses to the *fifo->mut* variable from the consumer threads after the variable has been freed by the main thread, which causes a segmentation fault. With COREMU, we diagnose the root cause of this bug similarly by inserting a watchpoint on *fifo->mut* and logging the accesses.

7. Conclusion and Future Work

We have presented the open-source COREMU, a scalable and portable full-system emulator for CMP systems. COREMU clusters multiple mature sequential emulators using a thin library layer,

[4] https://bugzilla.kernel.org/show_bug.cgi?id=14416

[5] http://www.eecs.umich.edu/ jieyu/bugs/pbzip2-094.html

hence decouples the complexity of supporting parallel emulation from building an optimizing sequential emulator. Experimental results show that COREMU has negligible uniprocessor performance overhead and scales much better than sequential emulators, and is orders of magnitude faster. From our experiences of building COREMU, we found that efficient emulation of synchronization primitives, efficient scheduling, scalable code cache management and efficient communication mechanism are the key to the performance and scalability of a parallel full-system emulator. We hope that our experiences could be useful for building other similar systems.

We plan to extend our work in several directions in future. First, while currently COREMU trade the determinism for performance by parallelizing the emulator, determinism is extremely useful to replay uncovered bugs. Hence, we plan to add record and replay support in COREMU, to support the execution replay of the full emulated multiprocessors [11]. Second, though there is no fundamental limitation to support other Host/Emulated processors pairs, we currently only tried a few. We are now trying to add more processors pairs to make it more portable. Finally, we are also providing more debugging and instrumentation support in COREMU to enable a more wide range of usages in performance debugging and diagnosis.

8. Acknowledgments and Availability

We thank the anonymous reviewers for their insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 61003002 and 60903015, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100, China National 863 program numbered 2008AA01Z138, a research grant from Intel as well as a joint program between China Ministry of Education and Intel numbered MOE-INTEL-09-04, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

The source code and OS images of COREMU can be downloaded from sourceforge (http://sourceforge.net/p/coremu/home/). They are distributed under the Lesser GNU General Public License.

References

[1] http://davmac.org/davpage/linux/rtsignals.html.

[2] Kvm/qemu. http://wiki.qemu.org/KVM.

[3] R. Bedichek. SimNow: Fast Platform Simulation Purely in Software. In *16th Hot Chips Symp*, 2004.

[4] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. PACT*, pages 72–81, 2008.

[5] Bochs. http://bochs.sourceforge.net/.

[6] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, et al. Mambo: a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.

[7] H. Cain, K. Lepak, B. Schwartz, and M. Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workload*, 2002.

[8] E. Chung, E. Nurvitadhi, J. Hoe, B. Falsafi, and K. Mai. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. In *Proc. FPGA*, pages 77–86, 2008.

[9] E. Chung, M. Papamichael, E. Nurvitadhi, J. Hoe, K. Mai, and B. Falsafi. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2009.

[10] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *IEEE HPCA*, 2008.

[11] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08*, pages 121–130, 2008.

[12] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proc. DISC*, pages 265–279, 2002.

[13] J. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

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

[15] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, pages 50–58, 2002.

[16] P. Magnusson and B. Werner. Efficient memory simulation in SimICS. In *Proc. Annual Simulation Symposium*, pages 62–73, 1995.

[17] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):99, 2005.

[18] M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. PODC*, pages 267–275, 1996.

[19] J. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *Proc. HPCA*, 2010.

[20] A. Over, B. Clarke, and P. E. Strazdins. A comparison of two approaches to parallel simulation of multiprocessors. In *Proc. ISPASS*, pages 12–22, 2007.

[21] QEMU. http://qemu.org/.

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

[23] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, 1995.

[24] A. Tridgell. Dbench filesystem benchmark. http://dbench.samba.org/.

[25] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology*. USENIX Association, 2004.

[26] K. Wang, Y. Zhang, H. Wang, and X. Shen. Parallelization of IBM mambo system simulator in functional modes. *SIGOPS Operating System Review*, 2008.

[27] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A practical FPGA-based framework for novel CMP research. In *Proc. FPGA*, pages 116–125, 2007.

[28] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[29] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):68–79, 1996.

[30] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical sampling of microarchitecture simulation. *ACM Trans. Model. Comput. Simul.*, 16(3):197–224, 2006.

[31] D. Yeh, L.-S. Peh, S. Borkar, J. A. Darringer, A. Agarwal, and W. mei Hwu. Thousand-core chips [roundtable]. *IEEE Design & Test of Computers*, 25(3):272–278, 2008.

[32] M. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proc. ISPASS*, pages 23–34, 2007.

[33] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *IPDPS*. IEEE Computer Society, 2004.