Cross-ISA Machine Emulation for Multicores

Emilio G. Cota
Columbia University, USA
cota@cs.columbia.edu

Paolo Bonzini
Red Hat, Italy
pbonzini@redhat.com

Alex Bennée
Linaro, UK
alex.bennee@linaro.org

Luca P. Carloni
Columbia University, USA
luca@cs.columbia.edu

Abstract

Speed, portability and correctness have traditionally been the main requirements for dynamic binary translation (DBT) systems. Given the increasing availability of multi-core machines as both emulation guests and hosts, scalability has emerged as an additional design objective. It has however been an elusive goal for two reasons: contention on common data structures such as the translation cache is difficult to avoid without hurting performance, and instruction set architecture (ISA) disparities between guest and host (such as mismatches in the memory consistency model and the semantics of atomic operations) can compromise correctness.

In this paper we address these challenges in a simple and memory-efficient way, demonstrating a multi-threaded DBT-based emulator that scales in an architecture-independent manner. Furthermore, we explore the trade-offs that exist when emulating atomic operations across ISAs, and present a novel approach for correct and scalable emulation of load-locked/store-conditional instructions based on hardware transactional memory (HTM). By adding around 1000 lines of code to QEMU, we demonstrate the scalability of both user-mode and full-system emulation on a 64-core x86_64 host running x86_64 guest code, and a 12-core, 96-thread POWER8 host running x86_64 and Aarch64 guest code.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code Generation; D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Dynamic Binary Translation, Scalability

1. Introduction

Dynamic binary translation (DBT) is a technique extensively used in virtualization, cross-ISA system emulation, legacy code migration, code instrumentation/replay, and as a front-end for computer architecture simulators. The increasing availability of multi-core machines as both emulation guests and hosts adds scalability to the traditional requirements for DBT engines, namely speed, portability and correctness. Salient examples of this requirement are the emulation of multi-core SoCs, which are integrating increasing numbers of cores [3] and even include chips with different ISAs [30], and computer architecture simulators that fail to scale [6] due to the lack of a portable, robust and scalable front-end. Our goal is thus to enable efficient emulation of multi-core guests on multi-core hosts while maintaining speed, portability and correctness.

Efficient, highly-parallel dynamic binary translation poses two challenges. First, concurrent access to key data structures should avoid contention on the memory hierarchy. Second, guest and host ISAs may differ in the implementation of atomic operations, as well as in the memory consistency model; such mismatches impose additional work on the DBT engine, beyond simply performing instruction-by-instruction translation. This paper proposes a design for a scalable dynamic binary translator that is simple, memory-efficient and correct. Our design, which we call Pico, is implemented on QEMU [7] due to its wide use and support of many different guest and host ISA combinations.

The two main contributions of this paper are:

- A memory-efficient design of a shared code cache for DBT engines. Based on the observation that *code translation* is rare, and that runtime is mostly spent on code execution, we keep the core logic of Pico simple, and achieve scalability through careful tuning of the emulator's data structures. In particular, the code cache is backed by a novel, highly concurrent hash table design that enables fast and scalable lookups (Section 3).
- A scalable, fully correct cross-ISA approach to emulating atomic instructions and reconciling guest-host differences in memory consistency models. We emulate strongly-ordered architectures on top of weaker ones by leveraging the work of Lustig et al. [24]. When possible, atomics are translated to the equivalent operation on the host. Otherwise, they are emulated faithfully either by instrumenting stores or, as a high-performance alternative, by leveraging hardware transactional memory (HTM) on hosts that support it (Section 4).

Pico's design was well received by the QEMU developer community, and we believe that the same ideas are applicable to other emulators as well, for example Valgrind. Furthermore, the implementation only requires modest modifications to QEMU, most of them generic; target- and host-specific changes are on the order of 200–300 lines for each target or host.

978-1-5090-4931-8/17 © 2017 IEEE

CGO 2017, Austin, USA

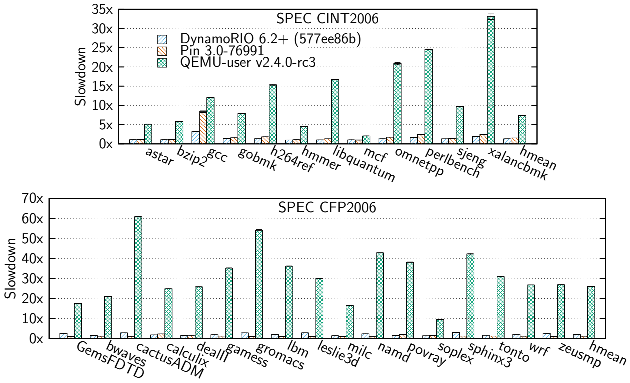


Figure 1: Slowdown for SPEC06 of DynamoRIO, Pin and QEMU-user vs. native execution on an Intel i7-6700 host.

We have already contributed bug fixes and scalability improvements to QEMU's support for multi-threaded user-mode emulation, and are working to include scalable system emulation in QEMU's future releases. This will enable other researchers and practitioners to easily benefit from our work.

We evaluate Pico on a 64-core x86_64 machine running x86_64 code, and on a 12-core, 96-thread POWER8 machine running x86_64 and Aarch64 code. Our results show that scalability of DBT with a shared code cache is comparable with native execution and hardware-assisted virtualization. Further, we quantify the implementation overhead of the different options to handle architectural mismatches between guest and host, exploring correctness vs. performance trade-offs for atomic instruction emulation.

2. Background and Related Work

2.1 Dynamic Binary Translation

Dynamic binary translation (DBT) is the basis for dynamic binary instrumentation (DBI) tools, such as DynamoRIO [11], Pin [23] and Valgrind [29]. DBT is well-suited for code instrumentation: it enables the handling of unmodified binaries (thus removing the need for recompilation/relinking) while covering all executed user-space code without requiring access to the original sources. The overhead of DBI tools is usually low, most of it coming from the analysis performed and not the instrumentation. For instance, Valgrind is significantly slower than Pin or DynamoRIO, in part due to the maintenance of a heavyweight shadow memory.

A major limitation of the above tools is the lack of support for cross-ISA analysis, therefore requiring the instrumented binary's ISA to match that of the host machine. Arguably the most widely used cross-ISA DBT tool is QEMU [7], which currently supports about two dozen different guest and eight host ISAs.

Cross-ISA portability comes at a price, however. As shown in Figure 1, execution of the SPEC06 benchmark suite is significantly slower in QEMU than in DynamoRIO or Pin. Most of the slowdown comes from the lower quality of the output code, as shown by Zhang et al. [35]. Floating point workloads (CFP2006) under QEMU incur an additional order of magnitude slowdown, because QEMU emulates the guest floating point instructions using integer instructions. This is done for portability and correctness: by not relying on any particular FPU, a single code base can support all possible guest-host combinations.

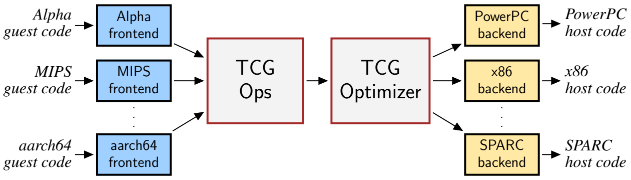


Figure 2: Cross-ISA portability is achieved in QEMU by leveraging TCG, its own intermediate representation.

QEMU leverages TCG, an intermediate representation (IR) into which guest ISA instructions are broken down. After a simple local optimization pass, these are translated into the host's ISA (Figure 2). Translation is more expensive in QEMU than in DynamoRIO and Pin (which do not need the intermediate step of an IR); nevertheless it does not constitute a large part of the run time. The intermediate representation also has a performance cost; this cost is also present in other retargetable DBT systems, such as Valgrind.

Another major difference between QEMU and other DBT tools is that QEMU supports both user-mode and full-system machine emulation. QEMU user-mode (*QEMU-user*) operates only on userspace code, issuing system calls natively on the host by appropriately translating guest system call numbers. Full-system emulation (*QEMU-system*) virtualizes entire machines (e.g. a PC or System-on-Chip), which include one or several processors and various peripherals.

Pico is able to work and scale across ISAs to 64 cores in both emulation modes. The techniques that Pico introduces to achieve this apply regardless of the translation overhead or quality of output code. Thus, we believe that the challenges and solutions proposed in this paper are applicable to dynamic binary translators at large.

2.2 Parallel Cross-ISA Machine Emulation

The increasing core counts in machines with diverse ISAs—from embedded systems to servers—calls for efficient, parallel cross-ISA emulators. Unfortunately, QEMU is not well-equipped for the task: QEMU-user spends large amounts of time sleeping on locks and stops all CPUs every time an atomic operation executes; QEMU-system is only parallel when used in conjunction with KVM[1] [20] otherwise it executes the emulated processors in a round-robin fashion in a single host thread.

[1] Optionally, instead of performing DBT, QEMU can run unmodified guest code directly on the host through the host processor's virtualization extensions. This is a high-performance option to consider when cross-ISA portability is not required.

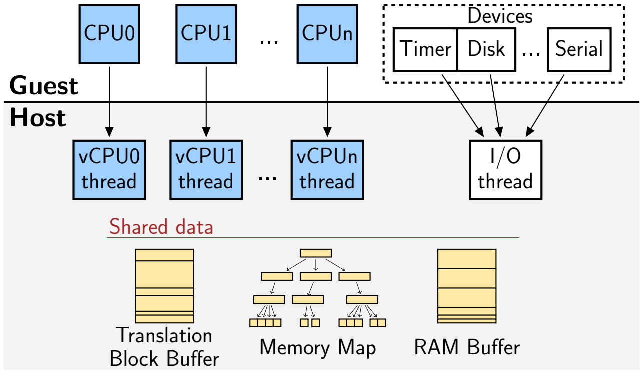


Figure 3: Pico's full-system architecture. Each guest CPU is emulated by a corresponding host "vCPU" thread. Guest devices are emulated by a single "I/O" thread.

The main design choice when developing a scalable emulator is whether to equip each guest CPU with a private translation code buffer, or to share cached translated code among all executing CPUs.

A comparison of shared vs. private code caches was conducted by Bruening et al. [12] using DynamoRIO as the underlying translation system. According to the authors, a private code cache is simpler to manage due to the absence of synchronization for most common operations. However, private code caches have a major downside in their potentially egregious memory consumption: while desktop workloads typically share very little code among threads, typical server workloads (e.g. web servers, databases) spawn hundreds of threads/processes that execute large amounts of shared code, either due to heavy interactions with the kernel (e.g. processing of heavy I/O or network traffic) or due to the high-level languages these workloads are written in (e.g., Java) [16]. Pico's simple shared code cache design is made possible by read-copy-update (RCU), a pattern for efficiently accessing data structures while concurrently updating them [26].

Two works in the literature address the problem of adding multicore support to QEMU: COREMU [33] and PQEMU [14]. COREMU uses private code caches, whereas PQEMU's authors experimented with both private and shared code caches. As an optimization, PQEMU supports a complex state machine to manage the shared code cache.

Neither PQEMU nor COREMU (including a later port for MIPS hosts [19]) address correctness issues specific to cross-ISA emulation. These issues arise due to differences between guest and host in their (1) memory consistency models and (2) atomic operation semantics. The first issue was recently addressed by Lustig et al. [24]. The authors model the insertion of memory barriers between memory accesses as a state machine; their ArMOR framework accepts a description of memory consistency models for the guest and the host, and produces such a state machine for use in an interpreter or dynamic binary translator. The second challenge, which to our knowledge has no prior work in the literature, has to do with the correct and scalable emulation of *load-locked/store-conditional* pairs (*LL/SC*) on hosts that only provide a *compare-and-swap* (*CAS*) instruction which, unlike *LL/SC*, is subject to the *ABA problem* [27]. Section 4 describes Pico's solution to these issues.

In addition, both COREMU and PQEMU require invasive changes to QEMU; they are based on ancient QEMU versions and seem to have aged badly. In COREMU we were only able to boot the Debian image that was released with it, while PQEMU [2] fails to compile on recent compilers.

3. Emulator Design

Pico's architecture, shown in Figure 3, has four main components: the state of CPUs, the memory map, the translation block cache and the guest RAM. In this section we cover the first three components, deferring the discussion of cross-ISA memory access emulation to Section 4.

We do not describe in detail the host "I/O" thread for device emulation, since it has been present in QEMU for several years, supporting concurrent emulation of devices and guest code execution with adequate performance.

3.1 CPU Execution

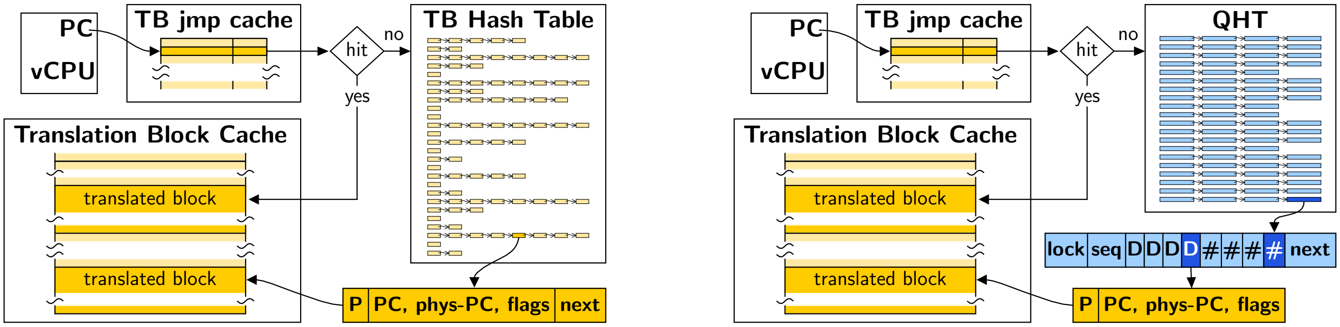
Pico allocates one host thread per emulated CPU; threads can then access the CPU registers without need for synchronization. However, CPUs do communicate with each other, even if sparingly; for example, the ARM architecture has instructions to flush the TLB on *all* cores in a *shareability domain*.

For this purpose Pico uses a two-pronged approach that scales for the common case, in which no communication is ongoing. First, every CPU loop is wrapped in an RCU read-critical section. Second, messages are delivered by setting a "flag" on the *consumer* CPU's state. The use of RCU allows *producer* CPUs to establish when messages have been consumed by waiting for a grace period to elapse. To ensure that the receipt of messages is bounded in time, CPUs check the request flag at the beginning of every translated basic block. Despite the simplicity, the cost of the checks is low: each basic block only needs two or three more instructions, depending on the host architecture—e.g., a load, a compare and a well-predicted branch. Note that QEMU does not attempt to compile multiple basic blocks into a single compiled trace, otherwise the cost could be easily amortized and further lowered by combining it with trace compilation.

3.2 Memory Map

QEMU-system organizes the guest's memory map as a tree of RAM, I/O and "container" memory regions, from which a radix tree is built for efficient lookups. Changes in the memory map are rare once the kernel has booted, and typically happen only in response to device hot-plug and hot-unplug. Given how infrequent these changes are, the radix tree is simply rebuilt from scratch whenever a change occurs.

[2] Available at `https://github.com/podinx/PQEMU`



(a) **(b)**

Figure 4: Translation Block lookup mechanisms in (a) QEMU and (b) Pico. Pico's improved hashing results in a more uniform bucket distribution. Further, QHT has higher performance due to its dynamic resizing and concurrent lookup support.

The memory map radix tree is read on every TLB miss[3] and on every interaction with an emulated memory-mapped device; it is therefore crucial for Pico to provide cheap access to it. The low update frequency and the "rebuild from scratch" approach make the memory map data structure an excellent candidate for RCU. Thus, once the tree is rebuilt, all CPUs are "kicked" out of their execution loop, thereby concluding their RCU read-side critical sections. This guarantees that they will all read the updated state upon resuming execution.

3.3 Translation Block Cache

The translation block cache minimizes code retranslation by buffering already translated code. It consists of *translation blocks* (TBs), i.e., guest basic blocks translated to the host architecture, that are indexed by an associated hash table.

Even though TBs can be invalidated, memory in the block cache is never reused. When the block cache is full, the emulator stops all CPUs and starts over with an empty buffer. Such flushing of the block cache is done for simplicity and performance. For most workloads and with an appropriately sized code buffer, the hit rate is very high and the buffer is flushed rarely—on the order of 2-3 flushes/minute in system emulation and practically never for user mode. Thus, maintaining an eviction policy (such as "least recently used") would likely result in a net loss of performance: the associated bookkeeping would negate the gains of avoiding already rare flushes.

The translation block cache and associated lookup mechanism in the original QEMU is shown in Figure 4a. The hash table points to TB descriptors, and is indexed by guest physical address. The hash table uses separate chaining with a fixed number of buckets. From the CPU execution loop, TB lookups are performed in two steps. First, threads access a direct-mapped, CPU-private cache. On a hit, a pointer to the corresponding TB is returned. On a miss, the shared hash table is accessed after acquiring a global lock. The CPU-private caches are tuned for latency; they are small and are invalidated relatively often (e.g., after every TLB flush). Thus, contention on this lock can be high.

Hash table design. Pico increases performance and scalability of the shared hash table in two ways. First, we improve the hashing function used to place TB descriptors in the hash table buckets. Second, we adopt a new hash table design that enables correct, concurrent lookups.

Pico uses *xxhash* [4], a high-performance non-cryptographic hash algorithm, to mix all three parts of the key: the virtual program counter (*PC*), the physical program counter (*phys-PC*) and a set of flags representing the active CPU mode (*flags*). Using all this information leads to a significantly more uniform distribution of TBs into buckets: for instance, the longest observed chain after fully booting Debian "jessie" on ARM is brought down from 550 to 40 TBs.

Our hash table design, called QHT, is highlighted in Figure 4b. Its main feature is support for concurrent reads with optimal scalability. In addition, even though Pico does not need it for the translation block cache, QHT also allows concurrent writes to separate buckets. In QHT, bucket chains are composed of cache line–sized nodes. Each node has a head spinlock for serializing writers, and stores multiple pointer-sized objects ("D" in the figure) along with their precomputed hashes (shown as "#").

QHT is similar to CLHT-LB [13]; however, CLHT imposes a restriction on the memory allocator that can be used with the hash table: the same address cannot be returned twice by the allocator while reads are occurring. To remove this restriction, QHT uses a per-bucket sequence number and the *seqlock* algorithm [21] to synchronize readers against writers. Readers of a *seqlock* need not perform any write, avoiding cache line bouncing and preserving scalability; they only need to retrieve the sequence number with a regular load (plus, on non-TSO hosts, a read fence) before and after traversing each bucket.

Writers update the sequence number before and after a concurrent write; therefore, if the low bit is set, readers know a writer is currently active. Readers wait until the low bit is clear, then access the bucket. If the sequence number changes during the access, the reader might have

[3] The memory map is not used in QEMU-user, since guest addresses are trivially translated to host addresses by adding a constant value.

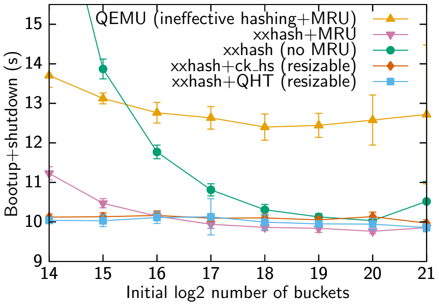


Figure 5: Bootup+shutdown time of Debian "jessie" in an ARM guest running on an Intel Haswell i7-4790K host.

seen an inconsistent state and retries the access. Retries are highly unlikely due to (1) the size of the hash table (it is not uncommon to have several hundred thousand elements), and (2) its low update rate—about 6% when booting Debian "jessie" on ARM, arguably a worst-case workload since most translated code is executed only once.

Figure 5 presents, for several hash table configurations, the time it takes to fully boot Debian "jessie" on ARM and immediately shut down. The original QEMU uses a fixed size hash table together with a most-recently-used (MRU) promotion policy, which moves items to the front of the bucket after every successful lookup. The plot shows that using MRU along with effective hashing (xxhash) *and* appropriate sizing gives optimal performance. On the other hand, MRU writes to memory on every lookup, which hurts scalability due to excessive cache line bouncing.

Furthermore, a fixed size hash without MRU has inferior performance due to excessive bucket chain lengths and increased number of cache misses. QHT virtually matches the performance of an ideally-sized hash table with MRU promotion by supporting resizes concurrent with lookups. Since resizes are rare, concurrent writes spin on bucket locks while a resize takes place. The freeing of pre-resize bucket chains is deferred by using RCU.

We benchmarked QHT against other hash table designs featuring resizing and concurrent lookup. While we did not apply CLHT in QEMU due to the aforementioned memory allocation requirements, Figure 5 shows that when booting a full system QHT has performance on a par with that of *ck_hs*, the hash set implementation from concurrencykit [1].

However, QHT and *ck_hs* show great performance differences in write-heavy scenarios. Figure 6 plots the scalability to 64 cores of QHT, CLHT and *ck_hs*, driven from a hash table microbenchmark operating on 200K elements at different update rates. Due to its use of seqlocks, QHT achieves performance comparable to CLHT's while not imposing restrictions on the memory allocator. On the other hand, *ck_hs* is an open-addressed hash set and therefore takes the same lock around every insert; as a result, it scales poorly even for modest update rates, which limits its general applicability. This limitation is shared with similar hash table implementations, such as the one proposed by Bruening et al. in [12].

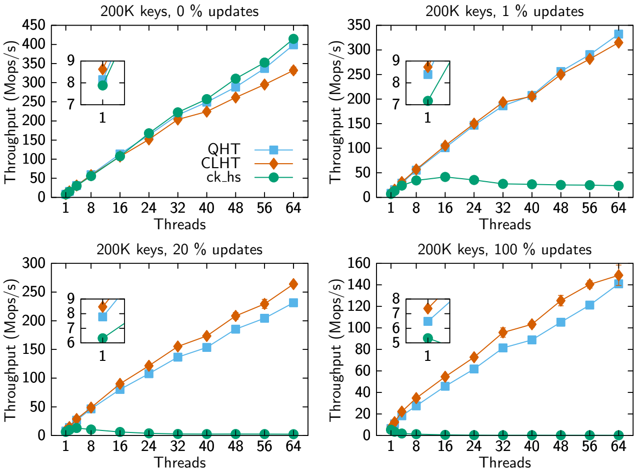


Figure 6: QHT, CLHT and *ck_hs* performance.

4. Correct, Cross-ISA Memory Accesses

Two cross-ISA issues are specific to multi-threaded emulators. First, the emulator has to handle mismatches in the memory consistency models of the source and target ISAs. Second, it has to correctly emulate the semantics of atomic operations. The latter problem, in turn, has to be attacked differently for the two families of atomic operations: *compare-and-swap* and other read-modify-write operations on one hand, and *load-locked/store-conditional* on the other.

4.1 Mismatches in the Memory Consistency Model

The memory consistency model of an ISA is made of a set of implicit ordering guarantees that the ISA promises to respect. These could be, for example, which memory accesses (loads, stores, atomic read-modify-write sequences) can be reordered in front of older accesses, or whether memory ordering obeys causality (also known as *transitive visibility*).

Performing DBT on a host that only allows reordering loads before older stores (such as x86_64) is trivial, because the emulated code cannot have an implicit ordering guarantee that the host does not respect. Unfortunately, the opposite case is problematic: performing DBT of guest code on a host whose ISA is *weaker* than that of the guest means that certain reorderings need to be explicitly forbidden by the translator.

Recent work by Lustig et al. [24] deals with exactly this problem: given two memory consistency models, they provide a framework, called ArMOR, that generates a state machine for the translator that guarantees correct execution of code from stronger ISAs on weaker ones.

We evaluate the impact of ArMOR's state machines in Pico's performance in Section 5.5.

4.2 Compare-and-Swap (CAS)

Compare-and-swap in the guest can be easily mapped to compare-and-swap in the host. Pico implements this similarly to how it appears in COREMU's downloadable code[4].

[4] Available at `http://coremu.sf.net/`

Multi-word *CAS* is required to emulate processors with 64-bit words on top of 32-bit ones. Fortunately, most 32-bit processors do provide 64-bit *CAS* or *load-locked/store-conditional* operations; the only exception is PowerPC processors (until POWER8, which can implement it using hardware transactional memory).

This approach cannot portably implement the x86 `cmpxchg16b` instruction, which performs a *CAS* operation on a 128-bit quantity. This operation is not available natively on 32-bit hosts, as well as on 64-bit PowerPC processors until POWER8. Fortunately, this instruction is rarely used by operating systems (e.g., Linux), and the x86 `cpuid` instruction marks its presence with a separate feature bit. We therefore either hide this feature bit from the guest, or fall back to the strategies used for emulating load-locked/store-conditional instructions, which are described below.

Bus-locked atomics. Architectures such as x86 support the atomic execution of certain instructions via a prefix (e.g., *LOCK xadd*). The same infrastructure used for *CAS* can also be used to implement other instructions such as atomic fetch-and-add or test-and-set; they can be trivially reduced to a *CAS* loop, as done in COREMU. However, for increased efficiency, Pico leverages the equivalent bus-locked instruction on the host whenever possible.

4.3 Load-Locked/Store-Conditional (*LL/SC*)

Rather than providing *CAS*, most RISC processors implement atomic read-modify-write operations through two instructions, *load-locked* (also known as *load-link*) and *store-conditional*. The first returns the current value of a memory location; the second stores a new value only if no updates have occurred to the location since the load. Unlike *CAS*, these instructions detect the case where a location has been changed to a different value and then back to the original.

Implementing *LL/SC* primitives is trivial in a sequential emulator. Whenever a *store-conditional* instruction is concurrent with one or more regular stores, however, a parallel emulator has to order the conditional store against each regular store. This is a *consensus problem* of order two, whose solution requires an atomic instruction (such as a test-and-set instruction) in both regular and conditional stores [17]. A trivial, non-scalable solution is to stop all other CPUs while executing *store-conditional* instructions. This is exactly what QEMU does; performance however drops very quickly even with very few concurrent threads. A solution, in order to scale, should thus avoid atomic instructions whenever the store does not conflict with *LL/SC* operations. We now present three different solutions, exploring the trade-offs between correctness, scalability and portability.

Pico-CAS: ***a (slightly) incorrect and scalable solution.*** The simplest, but nonetheless practical solution is to not implement exact *LL/SC* semantics; instead, a *store-conditional* operation can simply perform a *CAS* from the value fetched by *load-locked* to the argument of *store-conditional*.

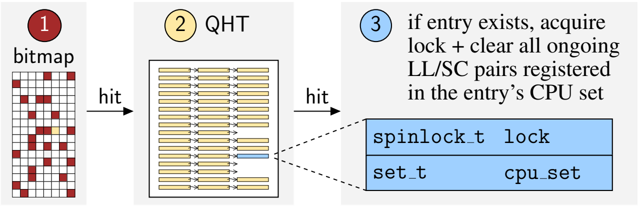

1
bitmap
hit
2 QHT
hit
3 if entry exists, acquire lock + clear all ongoing LL/SC pairs registered in the entry's CPU set
spinlock_t lock
set_t cpu_set

Figure 7: Instrumentation of stores in *Pico-ST*. Stores execute while holding the appropriate lock iff an atomic instruction has previously been performed on their target cache line.

Of course, this suffers from the *ABA problem* [27]: if the location were modified twice between *load-locked* and *store-conditional*, and the second write restored the original value, then the *store-conditional* would incorrectly succeed.

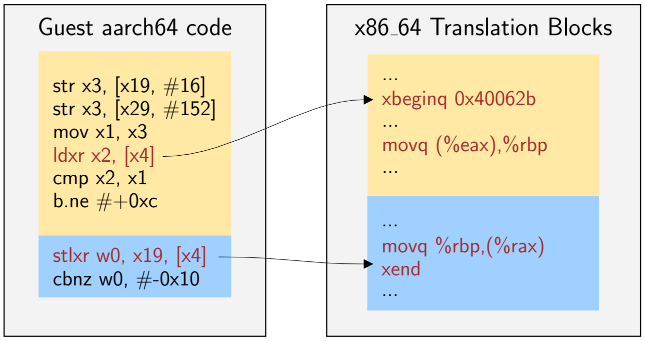
At the same time, in practice this is rarely problematic. Portable code using C/C++11 atomics only has access to *CAS*, and the entire Linux kernel does not employ *LL/SC* in ways that would break when emulated with *CAS*. Therefore, synchronization algorithms and lock-free data structures avoid the ABA problem using techniques such as reference counting, RCU or hazard pointers [28]. Nevertheless, it is worth looking further for fully correct solutions.

Pico-ST: ***a correct, scalable and portable solution.*** It is possible to avoid the ABA problem if one accepts a slowdown in single-threaded performance. This requires instrumenting stores to check whether the physical address to store to has *ever* been accessed atomically. If so, ongoing *LL/SC* pairs to the same cache line are canceled.

For each cache line that an *LL/SC* operates on, a corresponding entry is kept. An entry has two fields: a lock and a set of CPUs to track ongoing *LL/SC* operations. Entries are looked up by coupling a scalable hash table (we use QHT, see Section 3.3) and a bitmap. While each entry in the hash table represents a single cache line, each bit in the bitmap can represent a configurable number of cache lines, thus keeping its storage overhead practical. A bit set in the bitmap means that it *is possible* but not guaranteed that a lock might have to be taken for a given address; to dispel the uncertainty, the hash table is checked with the full address of the cache line. The instrumentation preceding emulated stores is depicted in Figure 7.

An additional measure is necessary to avoid races between the first *load-locked* operation on a cache line and a concurrent store to it by another CPU. This has to be done in three steps: (1) insert the appropriate entry in the hash table and bitmap, (2) kick all other CPUs out of the execution loop, and wait for them to actually exit; and (3) proceed with the *load-locked* emulation. Step 2 can be implemented easily by waiting for an RCU grace period (see Section 3.1). After Step 2, all regular stores to the newly-added cache line will be protected by the spinlock; this makes Step 3 safe.

Finally, we relax the requirement of keeping an entry for each cache line that an *LL/SC* *ever* operated on, which in the worst-case would degenerate into acquiring a lock on every



```
str x3, [x19, #16]
str x3, [x29, #152]
mov x1, x3
ldxr x2, [x4]
cmp x2, x1
b.ne #+0xc
stlxr w0, x19, [x4]
cbnz w0, #-0x10
```

```
...
xbeginq 0x40062b
...
movq (%eax),%rbp
...
...
movq %rbp,(%rax)
xend
...
```

Figure 8: Example *LL/SC* pair translated with *Pico-HTM*.

emulated store. Thus, we periodically reset both bitmap and hash table, thereby guaranteeing at negligible cost that the bitmap will remain sparsely populated.

Pico-HTM: ***leveraging hardware transactional memory.*** If the host also supports *LL/SC*, one may think of using them for emulating the target's *LL/SC*. This is dangerous, however, because most processors constrain the instructions that can appear between an *LL/SC* pair. If these restrictions are not respected, the store might fail spuriously. The extra overhead of dynamic translation, such as TLB lookups and register spills, may thus cause the store to fail forever.

Fortunately, some of the newest members of the POWER, s390 and x86_64 families feature hardware transactional memory (HTM), which does superficially resemble *LL/SC*. HTM is however more flexible than *LL/SC* and places fewer constraints on the instructions that can appear between the *LL/SC* pair. POWER processors, for example, can write to several hundred cache lines in a single transaction [22].

As depicted in Figure 8, Pico compiles *load-locked* to a "begin transaction" instruction followed by a regular load, and a *store-conditional* to a regular store followed by a "commit transaction" instruction. This works because all commercial HTM implementations provide *strong atomicity* [9]. Under strong atomicity, a store conflicting with a transaction will force the transaction to abort; the emulator can then check for conflicting regular stores just by testing whether the transaction completed successfully.

There is one important difference between HTM and *LL/SC*. Regular stores between the *load-locked* and *store-conditional* instructions persist after a failed conditional store; with HTM, instead, an abort rolls back all stores in the transaction. We cannot therefore map a transaction abort directly to a *store-conditional* failure. Instead, we retry the transaction until it succeeds, so that the conditional store actually never reports a failure. Because the semantics of *store-conditional* is respected, the difference is not visible to the emulated program.

HTM also requires a fallback for repeated aborts. After a few failed attempts, or if one of the blocks between *load-locked* and *store-conditional* is not translated, Pico executes the *LL/SC* sequence with all other CPUs stopped.

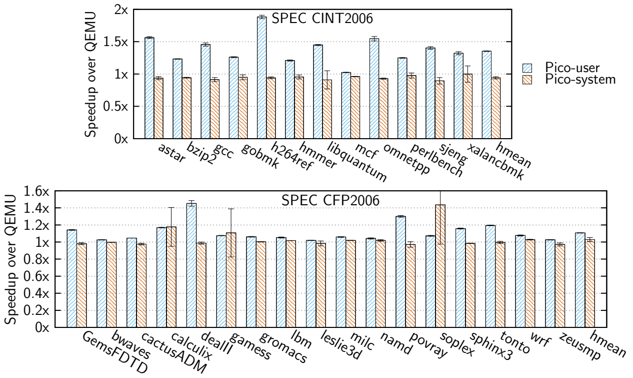


Figure 9: Speedup on *SKL* of Pico over QEMU for single-threaded x86_64 SPEC06 workloads.

5. Evaluation

5.1 Setup

Machines. We perform measurements on the following machines, which all have 64G of RAM and run Linux v4.4:

SKL has a 3.4GHz 4-core Intel Skylake i7-6700 processor with 2-way simultaneous multithreading (SMT), for a total of 8 hardware threads.

P8 has two 3.3 GHz 6-core IBM POWER8 processors with 8-way SMT, for a total of 96 hardware threads.

AMD has four 2.3GHz, 16-core AMD Opteron 6376 processors, for a total of 64 cores.

The guest kernel is always Linux v4.4. We use QEMU v2.4.0-rc3 as the baseline emulator, since newer versions already include some of the improvements in Pico.

Workloads. We use SPEC CPU2006 benchmarks for single-threaded performance measurements. For measuring scalability we use the PARSEC suite [8], as well two server workloads: the *pgbench* tool in PostgreSQL v9.5 [2] and the *Masstree* in-memory key-value store [25]. We compare Pico-user to native execution, and Pico-system to KVM.

For evaluating the overhead of atomic instruction emulation we wrote a simple microbenchmark called *atomic_add*. Each thread in *atomic_add* executes a loop that atomically increments a random element of an array of integers, which is appropriately padded to avoid false cache line sharing. By varying the size of the array, we can observe different levels of contention in the memory hierarchy.

All experiments are run 5 times; we show the resulting mean and corrected sample standard deviation. For each experiment we choose the thread pinning policy that exhibits higher performance from either scattering threads evenly across NUMA nodes, or favoring same-node pinnings.

5.2 Single-Threaded Performance

This experiment (Figure 9) compares the performance of Pico for single-threaded execution over the baseline QEMU implementation on *SKL*. QEMU-user already supports mul-

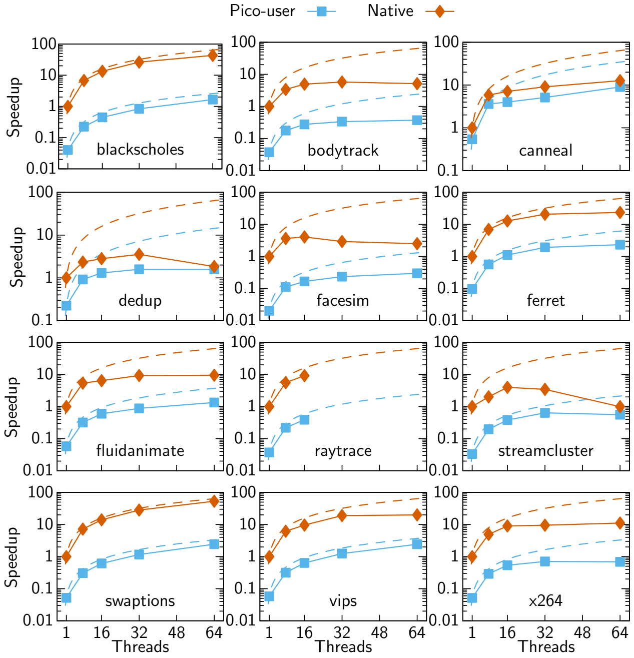


Figure 10: Speedup over Native on *AMD* for PARSEC.

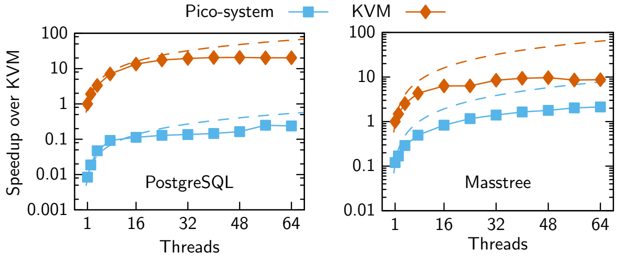


Figure 11: Speedup vs. KVM on *AMD* for server workloads.

tiple threads of execution, but this introduces overhead even for single-threaded programs; for example, translation block lookups (Section 3.3) are serialized with a lock. Pico-user is thus 20–90% faster than QEMU-user, mostly due to its better translation block hashing and to QHT's cache efficiency.

Pico-system introduces locks and fences to cover previously unprotected data structures, and converts some memory accesses to atomic operations. This balances the advantages of improved hashing, making performance virtually identical to QEMU-system—on average slightly worse for integer benchmarks, and slightly better for floating-point.

5.3 Parallel Workloads

In this experiment we evaluate the scalability of Pico-user when running parallel code from PARSEC with the `native` input set[5]. Figure 10 compares the scalability of Pico against that of a native run on *AMD*. Speedups are shown normalized over the native's single-threaded execution times; ideal, linear scaling is shown with a dashed line for comparison.

Most PARSEC benchmarks do not scale well to dozens of cores [31], showing a performance cliff that is indicative of excessive cache line contention. Nevertheless, on average, the scalability of Pico is better than that of native execution. This is because the slowdown caused by DBT reduces the rate of accesses to shared memory in the workload, thereby delaying the onset of the contention-related performance cliff (e.g., *streamcluster*) and even avoiding it altogether (e.g., *facesim*, *fluidanimate*). This effect is present for most workloads we tested, but it is particularly visible here due to the slow floating point emulation in the DBT engine.

Benchmarks that do not have particular contention show similar scalability under both Native and Pico, as can for instance be seen in *swaptions*, *blackscholes* or *canneal*.

5.4 Server Workloads

We now show the results for full-system emulation of a guest with 32G of RAM running multi-threaded (Masstree) and multi-process (PostgreSQL) server workloads on *AMD*. These workloads are representative of large virtual machines and stress two potential sources of performance degradation: code translation and device emulation.

Code translation is stressed by multi-process server workloads due to their code size [12], and because the virtual address of the program counter is part of the translation block hash key; even if the text of the program is loaded only once in memory by the operating system, techniques such as address space layout randomization (ASLR) can cause the emulator to translate it multiple times. Device emulation is stressed simply because QEMU runs all emulation under a single global mutex, and we did not change this in Pico.

For PostgreSQL, we use *pgbench* to create and populate a database of scale factor 150, running each test for 120s and spawning two connections per thread. Both PostgreSQL server and *pgbench* run in the same guest, with the server configured with a buffer of 8GB. For Masstree, we run 10s get/put tests on a database initialized with 140M "1-to-10-byte decimal" [25] keys.

Figure 11 shows the results of the experiment. Both of the benchmarks scale up to 64 threads. In the case of PostgreSQL, which performs disk and socket I/O as well, device emulation is a bottleneck at 16 to 40 threads. In this range, KVM can take advantage of optimizations to device emulation, such as moving the CPU's interrupt controller (local APIC) inside the hypervisor and triggering the QEMU I/O thread directly from the hypervisor. Pico on the other hand cannot keep all cores pegged at 100% CPU utilization. Nevertheless, the scalability of Pico is in line with KVM's.

Masstree, which is a memory-only workload, scales under Pico up to at least 64 threads. As in the PARSEC benchmarks, emulation scales better than KVM because of the reduced rate of shared memory accesses. In neither case code translation turns out to be a bottleneck.

[5] We had issues running two PARSEC benchmarks on *AMD*: *freqmine* crashed frequently, and so did *raytrace* beyond 16 threads. The crashes happened for both native execution and Pico.

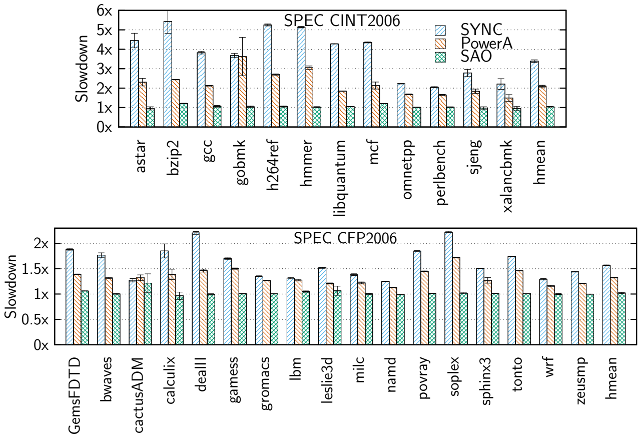


Figure 12: Slowdown on *P8* for Pico-user running x86_64 SPEC06 benchmarks, with two ArMOR state machines and hardware strong-access ordering, relative to omitting all barriers in the translated code.

5.5 Mismatches in the Memory Consistency Model

In order to quantify the cost of emulating parallel code from strongly-ordered ISAs on weakly-ordered hosts, we implemented the two state machines provided by ArMOR for executing x86_64 guest code on a PowerPC host. We then ran x86_64 SPEC06 code on *P8*. The two state machines, which we call SYNC and PowerA, are summarized as follows.

- *SYNC:* Insert a full memory barrier (`sync` in PowerPC assembly language) before every load or store. This is always correct for all possible legal code.
- *PowerA:* Separate loads with `lwsync` barriers, pretending that PowerPC is multi-copy atomic even though it is not[6]. While this does allow illegal behavior for the `iriw` litmus test [10], it allows for greater performance if the user is sure that the system does not include code similar to `iriw`—which indeed is practically never seen.

The results are shown in Figure 12. SYNC provides full correctness but incurs significant slowdown, on average surpassing 3X for integer workloads. PowerA has a more adequate average slowdown of approximately 2X for CINT2006, at the expense of not handling correctly the `iriw` pattern. The slowdown of both approaches is significantly lower for CFP2006, since floating point emulation dominates execution time.

P8 has hardware support for strong-access ordering (*SAO*) [5], which in our tests shows negligible overhead. Unfortunately, SAO is only available on recent IBM hardware, which makes its use highly non-portable. Nevertheless, ArMOR is a viable strategy to correctly emulate parallel code from strongly-ordered ISAs on weakly-ordered hosts.

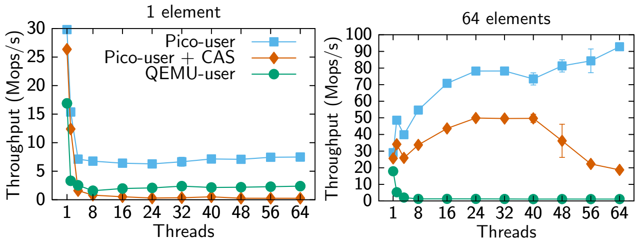


Figure 13: Performance on *AMD* for x86_64 *atomic_add*.

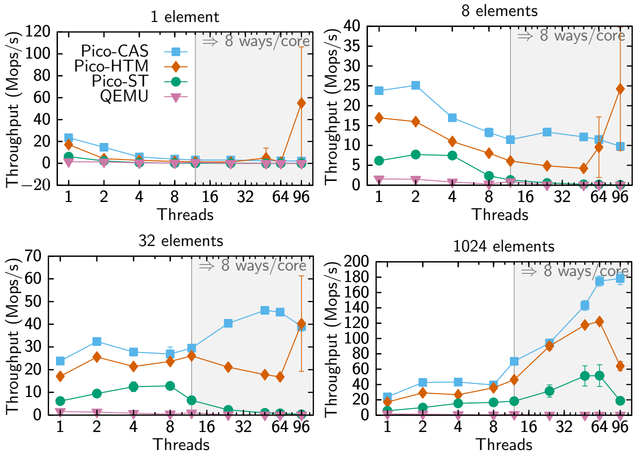


Figure 14: Performance on *P8* for Aarch64 *atomic_add*.

5.6 Bus-Locked Atomics

Figure 13 compares the performance of emulating bus-locked atomics with the equivalent bus-locked atomics on the host (Pico) and emulating them with a *CAS* loop (Pico + CAS). In hardware, bus-locked atomics can be implemented more efficiently than *CAS*. This explains why Pico's performance is superior with just 1 array element, which is a worst-case scenario from a scalability viewpoint. Furthermore, when contention is slightly lower (64 elements), Pico shows greater scalability, scaling to 64 cores whereas the *CAS* loop's performance collapses around 40 cores.

QEMU shows no scalability in either scenario, due to its serial emulation of guest atomics.

5.7 Load-Locked/Store-Conditional (*LL/SC*)

Figure 14 compares *atomic_add* performance on *P8* for QEMU-user and the three Pico approaches presented in Section 4.3. We ran these experiments on *P8* because of its HTM support and large number of hardware threads.

The overall performance of the Pico approaches depends on the overhead of the emulation mechanism. Thus, Pico-CAS is the fastest, followed by Pico-HTM and Pico-ST. All three approaches show scalability that grows as contention (i.e., number of array elements) is reduced.

Measurements are stable in all benchmarks when SMT is not used (i.e., less or equal than 12 threads), and progressively become more noisy. For 96 threads, Pico-HTM is occasionally able to beat Pico-CAS and scale almost linearly.

[6] See the ArMOR paper for details on this state machine.

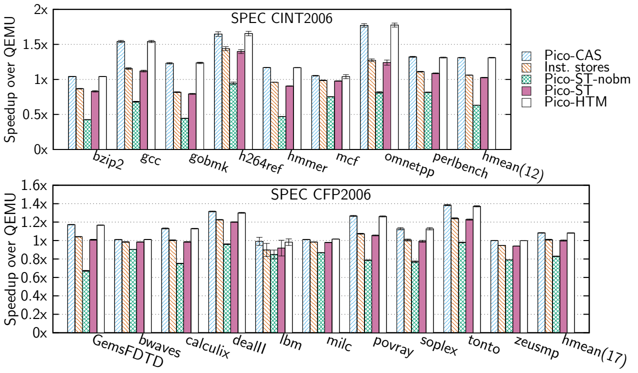


Figure 15: Speedup over QEMU on *SKL* of different implementations of *LL/SC* in Pico-user runs of Aarch64 SPEC06. Some benchmark results not shown due to space constraints.

Our hypothesis is that the POWER8 microcode optimizes the case where all hardware threads in the same core are contending for the same cache lines, and effectively ensures that the transactions do not conflict. In fact, by forcing the benchmark to run on fewer cores we were able to see the same behavior also for 32, 48 and 64 threads (corresponding to 4, 6 and 8 fully-utilized cores).

The plots show the trade-off between correctness, scalability and portability. If fully correct *LL/SC* emulation is not necessary, Pico-CAS has the highest performance while maintaining portability. Failing that, Pico-HTM provides good performance if processor support is available or, for hosts without HTM extensions, Pico-ST is both fully correct and portable.

Of the three, Pico-ST is also the only one to have non-trivial single-threaded overhead. To characterize this, we emulate SPEC06 benchmarks (compiled for Aarch64) on *SKL* in user-mode. Figure 15 shows a subset of the results (some omitted due to space constraints), although the mean is computed from the full set of results.

Pico-HTM and Pico-CAS show the highest performance, which is not surprising given that they only affect the emulation of atomic operations; these are rare in SPEC06. The speedup is similar to that in Figure 9; as discussed in Section 5.2, it is due to improvements in Pico, and not to atomic instruction emulation.

In order to analyze the performance of Pico-ST, we include two additional sets of results in Figure 15. The *Inst. stores* set is obtained from instrumenting stores in Pico with empty helper functions, thereby measuring the cost of calling C code around every store; *Pico-ST-nobm* is Pico-ST without the bitmap acting as a filter for QHT lookups. Two observations can be made. First, the limitations in QEMU's register allocator introduce a high overhead in Pico's store instrumentation; the C helpers slow down emulation on average by around 20%. Second, the bitmap plays a key role in filtering accesses to the hash table; QHT, despite its high performance, requires a non-trivial amount of instructions that are easily outperformed by a bitmap lookup. Thanks to the bitmap, Pico-ST only has 4% overhead above that of store instrumentation. Since the bitmap check is so effective, a natural improvement to Pico-ST would be for the backends to inline it in the generated assembly code. This change would be more invasive than the other changes we made in Pico, for which portable C code was enough.

6. Additional Related Work

Concurrent hash tables. The idea of using cache-friendly buckets, as done by QHT and CLHT, was introduced earlier in MemC3 [15], albeit partial hashes were used to minimize cache references due to Cuckoo hashing. Leveraging RCU for concurrency in hash tables was proposed by Triplett et al. [32]. We use RCU for deferring QHT bucket's deletion, and handle dynamic resizes (which in our case are rare) by acquiring all bucket locks.

Cross-ISA emulation. HQEMU [18] speeds up DBT by translating frequently-executed guest basic blocks into high-quality host code. This translation is concurrent with code execution, which allows multi-core hosts to speed up DBT for single-threaded code. HERMES [35] shares the same goal, but instead of exploiting concurrency, it optimizes the quality of QEMU's emitted code.

Cross-ISA dynamic binary instrumentation. PEMU [34] is an instrumentation tool based on QEMU, with an interface compatible with that of Pin [23] yet supporting both user and full-system modes. PEMU could be combined with Pico to build parallel, cross-ISA binary instrumentation tools.

7. Conclusion

We have presented Pico, a novel design for multi-threaded cross-ISA emulators. Pico leverages multi-core hosts by using a shared code cache from a highly parallel fast path. Furthermore, Pico adopts recent research for handling memory consistency model mismatches between guest and host, and proposes different strategies for emulation of atomic instructions and strongly-ordered memory accesses.

We have implemented our design in the QEMU open source emulator, with an explicit focus on contributing our code to the project. Our experimental evaluation covers both user-mode and full-system emulation, comparing Pico with both native execution and the state of the art. Our results show that Pico's design scales to 64 cores without forgoing simplicity or memory efficiency.

Acknowledgments

We thank Richard Henderson, Sergey Fedorov and the rest of the QEMU developer community for their invaluable help. We also thank Jason Nieh, David W. King and the anonymous reviewers for their feedback. This work is supported in part by DARPA PERFECT (C#: R0011-13-C-0003) and C-FAR (C#: 2013-MA-2384), an SRC STARnet center.

References

[1] http://concurrencykit.org.

[2] PostgreSQL. https://www.postgresql.org/.

[3] Cavium ThunderX Processors. http://www.cavium.com.

[4] https://github.com/cyan4973/xxhash.

[5] IBM Power ISA, version 2.06 revision b. *Book I: Power ISA User Instruction Set Architecture*, 2010.

[6] E. K. Ardestani and J. Renau. ESESC: A fast multicore simulator using time-based sampling. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 448–459, 2013.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conference (ATC)*, pages 41–46, 2005.

[8] C. Bienia et al. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[9] C. Blundell, E. C. Lewis, and M. M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.

[10] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78, 2008.

[11] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 265–275, 2003.

[12] D. Bruening, V. Kiriansky, T. Garnett, and S. Banerji. Thread-shared software code caches. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 28–38, 2006.

[13] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 631–644, 2015.

[14] J. H. Ding et al. PQEMU: A parallel system emulator based on QEMU. In *Proc. of the Intl. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 276–283, 2011.

[15] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 371–384, 2013.

[16] Ferdman et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, 2012.

[17] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, 1991.

[18] D.-Y. Hong et al. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 104–113, 2012.

[19] X.-W. Jiang et al. A parallel full-system emulator for RISC architure host. In *Advances in Computer Science and its Applications*, pages 1045–1052. 2014.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Proc. of the Ottawa Linux Symposium*, volume 1, pages 225–230, 2007.

[21] C. Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, 2005.

[22] H. Le et al. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8–1, 2015.

[23] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[24] D. Lustig et al. ArMOR: defending against memory consistency model mismatches in heterogeneous architectures. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 388–400, 2015.

[25] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. of the European Conf. on Computer Systems (EuroSys)*, pages 183–196, 2012.

[26] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[27] M. M. Michael. ABA prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

[28] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[29] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[30] S. Snapdragon. Processors: System on chip solutions for a new mobile age. *White paper, ARM*, 2011.

[31] G. Southern and J. Renau. Deconstructing PARSEC scalability. In *Proc. of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2015.

[32] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proc. of the USENIX Annual Technical Conference (ATC)*, 2011.

[33] Z. Wang et al. COREMU: A scalable and portable parallel full-system emulator. In *Proc. of Principles and Practice of Parallel Programming (PPoPP)*, pages 213–222, 2011.

[34] J. Zeng, Y. Fu, and Z. Lin. PEMU: A Pin highly compatible out-of-VM dynamic binary instrumentation framework. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*, pages 147–160, 2015.

[35] X. Zhang et al. HERMES: A fast cross-ISA binary translator with post-optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 246–256, 2015.