Cross-ISA Machine Emulation for Multicores

Emilio G. Cota
Paolo Bonzini
Alex Bennée
Luca P. Carloni

Columbia University
Red Hat, Inc.
Linaro, Ltd.
Columbia University
Demand for *Scalable* Cross-ISA Emulation

- Increasing core counts for emulation guests (typically high-perf SoC's)
  - Hosts (servers) are already many-core
- ISA diversity is here to stay
  - e.g. x86, ARM/aarch64, POWER, RISC-V

**our goal:** efficient, correct, multicore-on-multicore cross-ISA emulation
Scalable Cross-ISA Emulation

Challenges

(1) Scalability of the DBT engine
   key data structure: translation code cache

(2) ISA disparities between guest & host:
   (2.1) Memory consistency mismatches
   (2.2) Atomic instruction semantics
      i.e. compare-and-swap vs. load locked-store conditional

Related Work:

- PQEMU [14] and COREMU [33] do not address (2)
- ArMOR [24] solves (2.1)

Our contributions: (1) & (2.2)

Our Proposal: Pico

Makes QEMU [7] a *scalable* emulator

Open source: http://qemu-project.org

Widely used in both industry and academia

Supports many ISAs through TCG, its IR:

Our contributions are not QEMU-specific

They are applicable to Dynamic Binary Translators at large

Emulator Design
Pico's Architecture

- One host thread per guest CPU
  - Instead of emulating guest CPUs one at a time
- Key data structure: Translation Block Cache (or Buffer)
  - See paper for details on Memory Map & CPU state
Translation Block Cache

- Buffers Translation Blocks to minimize retranslation
- Shared by all CPUs to minimize code duplication
  - see [12] for a private vs. shared cache comparison

To scale, we need concurrent code execution

QEMU's Translation Block Cache

Problems in the TB Hash Table:

- Long hash chains: slow lookups
  - Fixed number of buckets
  - hash=h(phys_addr) leads to uneven chain lengths
- No support for **concurrent lookups**
Pico's Translation Block Cache

- $\text{hash} = h(\text{phys_addr}, \text{phys_PC}, \text{cpu_flags})$: uniform chain distribution
  - e.g. longest chain down from 550 to 40 TBs when booting ARM Linux
- **QHT**: A resizable, scalable Hash Table
TB Hash Table

Requirements

- Fast, concurrent lookups
- Low update rate: max 6% booting Linux
TB Hash Table

Requirements
- Fast, concurrent lookups
- Low update rate: max 6% booting Linux

Candidate #1: **ck_hs** [1] (similar to [12])

- **Open addressing**: great scalability under ~0% updates
- **Insertions take a global lock**, limiting update scalability

![Graphs showing performance of ck_hs with different update rates.](attachment:image)

**TB Hash Table**

**Requirements**
- Fast, concurrent lookups
- Low update rate: max 6% booting Linux

**Candidate #1:** ck_hs [1] (similar to [12])

**Candidate #2:** CLHT [13]
- Resizable + scalable lookups & updates
- Wait-free lookups

However, imposes restrictions on the memory allocator

---

[1] [http://concurrencykit.org](http://concurrencykit.org)
TB Hash Table

Requirements

- Fast, concurrent lookups
- Low update rate: max 6% booting Linux

Candidate #1: \texttt{ck hs} [1] (similar to [12])
Candidate #2: \texttt{CLHT} [13]

#3: Our proposal: \texttt{QHT}

- Lock-free lookups, but \textbf{no restrictions on the mem allocator}
  - Per-bucket sequential locks; retries very unlikely

![Graphs showing throughput comparisons for different hash table implementations](image-url)

QEMU emulation modes

User-mode emulation (QEMU-user)

- DBT of user-space code only
- System calls are run natively on the host machine
- QEMU executes all translated code under a global lock
  - Forces serialization to safely emulate multi-threaded code

Full-system emulation (QEMU-system)

- Emulates an entire machine
  - Including guest OS and system devices
- QEMU uses a single thread to emulate guest CPUs using DBT
  - No need for a global lock since no races are possible
Single-threaded perf (x86-on-x86)

- Pico-user is 20-90% faster than QEMU-user due to lock-less TB lookups
- Pico-system's perf is virtually identical to QEMU-system's
  - ARM Linux boot results in the paper; Pico-system ~20% faster
Parallel Performance (x86-on-x86)

- Speedup normalized over Native's single-threaded perf
- Dashed: Ideal scaling
- QEMU-user not shown: does not scale at all
Parallel Performance (x86-on-x86)

- Speedup normalized over Native's single-threaded perf
- Dashed: Ideal scaling
- QEMU-user not shown: does not scale at all
- Pico scales better than Native
  - PARSEC known not to scale to many cores [31]
  - DBT slowdown merely delays scalability collapse
- Similar trends for server workloads (Pico-system vs. KVM): see paper

Guest & Host
ISA Disparities
Atomic Operations

Two families:

**Compare-and-Swap (CAS)**

```c
/* runs as a single atomic instruction */
bool CAS(type *ptr, type old, type new) {
    if (*ptr != old) {
        return false;
    }
    ptr = new;
    return true;
}
```

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

```c
/* store_exclusive() returns 1 if addr has been written to since load_exclusive() */
do {
    val = load_exclusive(addr);
    val += 1;  /* do something */
} while (store_exclusive(addr, val);
```

**x86/IA-64:** `cmpxchg`

**Alpha:**

**POWER:**

**ARM:**

**aarch64:**

**MIPS:**

**RISC-V:**

**x86/IA-64:**

**ldl_l/stl_c**

**lwaxx/stwx**

**ldrex/strex**

**ldaxr/strlxr**

**ll/sc**

**lr/sc**

Challenge: How to **correctly** emulate atomics in a parallel environment, without hurting **scalability**?
Challenge: How to *correctly* emulate atomics in a parallel environment, without hurting *scalability*?

**CAS on CAS host:** Trivial

**CAS on LL/SC:** Trivial

**LL/SC on LL/SC:** Not trivial

Cannot safely leverage the host's LL/SC: operations allowed between LL and SC pairs are limited

**LL/SC on CAS:** Not trivial

LL/SC is stronger than CAS: ABA problem
ABA Problem

Init: *addr = A;

cpu0
do {
    val = load_exclusive(addr); /* reads A */
    ...
    ...
} while (store_exclusive(addr, newval);

<table>
<thead>
<tr>
<th>cpu0</th>
<th>cpu1</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
</tr>
</tbody>
</table>

SC fails, regardless of the contents of *addr

cpu0
do {
    val = atomic_read(addr); /* reads A */
    ...
    ...
} while (CAS(addr, val, newval);

<table>
<thead>
<tr>
<th>cpu0</th>
<th>cpu1</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
</tr>
</tbody>
</table>

CAS succeeds where SC failed!

| time |      |
Pico's Emulation of Atomics

3 proposed options:

1. Pico-CAS: pretend ABA isn't an issue
   - Scalable & fast, yet incorrect due to ABA!
     - However, portable code relies on CAS only, not on LL/SC (e.g. Linux kernel, gcc atomics)

2. Pico-ST: "store tracking"
   - Correct & scalable
   - Perf penalty due to instrumenting regular stores

3. Pico-HTM: Leverages HTM extensions
   - Correct & scalable
   - No need to instrument regular stores
     - But requires hardware support
Pico-ST: Store Tracking

- Each address accessed atomically gets an entry of CPU set + lock
  - LL/SC emulation code operates on the CPU set atomically
- Keep entries in a HT indexed by address of atomic access
- **Problem**: regular stores must abort conflicting LL/SC pairs!
- **Solution**: instrument stores to check whether the address has ever been accessed atomically
  - If so (rare), take the appropriate lock and clear the CPU set
- **Optimization**: *Atomics << regular stores*: filter HT accesses with a sparse bitmap

![Diagram showing the process of Pico-ST](image)
Pico-HTM: Leveraging HTM

- HTM available on recent POWER, s390 and x86_64 machines
- Wrap the emulation of code between LL/SC in a transaction
  - Conflicting regular stores dealt with thanks to the strong atomicity [9] in all commercial HTM implementations: "A regular store forces all conflicting transactions to abort."


- Fallback: Emulate the LL/SC sequence with all other CPUs stopped
- Fun fact: no emulated SC ever reports failure!
Atomic emulation perf

Pico-user, single thread, aarch64-on-x86

- Pico-CAS & HTM: no overhead (but only HTM is correct)
- Pico-ST: Virtually all overhead comes from instrumenting stores
- Pico-ST-nobm: highlights the benefits of the bitmap
Atomic emulation perf

Pico-user *atomic_add*, multi-threaded, aarch64-on-POWER

```c
struct count {
    u64 val;
} __aligned(64); /* avoid false sharing */

struct count *counts;

while (!test_stop) {
    int index = rand() % n_elements;
    atomic_increment(&counts[index].val);
}
```

*atomic_add* microbenchmark

- All threads perform atomic increments in a loop
- No false sharing: each count resides in a separate cache line
- Contention set by the *n_elements* parameter
  - i.e. if *n_elements* = 1, all threads contend for the same line
- Scheduler policy: evenly scatter threads across cores
Atomic emulation perf

Pico-user *atomic_add*, multi-threaded, aarch64-on-POWER

Trade-off: correctness vs. scalability vs. portability

- All Pico options scale as contention is reduced
  - QEMU cannot scale: it stops all other CPUs on every atomic
- Pico-CAS is the fastest, yet is not correct
- Pico-HTM performs well, but requires hardware support
- Pico-ST scales, but it is slowed down by store instrumentation
- HTM noise: probably due to optimized same-core SMT transactions
Wrap-Up

Contributions:

- Scalable DBT design with a shared code cache
- Scalable, correct cross-ISA emulation of atomics
QEMU Integration

- QEMU v2.7 includes our improved hashing + QHT
- QEMU v2.8 includes:
  - atomic instruction emulation
  - Support for parallel user-space emulation
  - Under review for v2.9: parallel full-system emulation

Contributions:
- Scalable DBT design with a shared code cache
- Scalable, correct cross-ISA emulation of atomics

Wrap-Up
Thank you
Backup Slides
Linux boot
single thread

- **QHT** & **ck_hs** resize to always achieve the best perf
  - but **ck_hs** does not scale w/ ~6% update rates
Server Workloads (x86-on-x86)

- Server workloads have higher code footprint [12] and therefore stress the TB cache
- PostgreSQL: Pico's scalability is inline with KVM's
- Masstree [25], an in-memory Key-Value store, scales better in Pico
  - Again, the DBT slowdown delays cache contention

We applied ArMOR's [24] FSMs:

- **SYNC**: Insert a full barrier before every load or store
- **PowerA**: Separate loads with `lwsync`, pretending that POWER is multi-copy atomic, and also leveraged
- **SAO**: Strong Access Ordering

RCU is a way of waiting for things to finish, without tracking every one of them.
Sequence Locks

```c
void *qht_lookup__slowpath(struct qht_bucket *b, qht_lookup_func_t func, const void *userp, uint32_t hash)
{
    unsigned int version;
    void *ret;

    do {
        version = seqlock_read_begin(&b->sequence);
        ret = qht_do_lookup(b, func, userp, hash);
    } while (seqlock_read_retry(&b->sequence, version));
    return ret;
}
```

Reader: Sequence number must be even, and must remain unaltered. Otherwise, retry

---

Reader

```
seq=0
seq=3
```

Retry

```
seq=3
seq=4
```

Retry

---

Writer

```
seq=1
seq=2
seq=3
seq=4
```
CLHT malloc requirement

```c
val_t val = atomic_read(&bucket->val[i]);
smp_rmb();
if (atomic_read(&bucket->key[i]) == key && atomic_read(&bucket->val[i]) == val) {
  /* found */
}
```

“the memory allocator of the values must guarantee that the same address cannot appear twice during the lifespan of an operation.


ASPLOS, pages 631–644, 2015
Multi-copy Atomicity

iriw litmus test

<table>
<thead>
<tr>
<th>cpu0</th>
<th>cpu1</th>
<th>cpu2</th>
<th>cpu3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td>y=1</td>
<td>r1=x</td>
<td>r3=y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2=y</td>
<td>r4=x</td>
</tr>
</tbody>
</table>

- Forbidden outcome: \( r_1 = r_3 = 1, \ r_2 = r_4 = 0 \)
- The outcome is forbidden on x86
- It is observable on POWER unless the loads are separated by a sync instruction