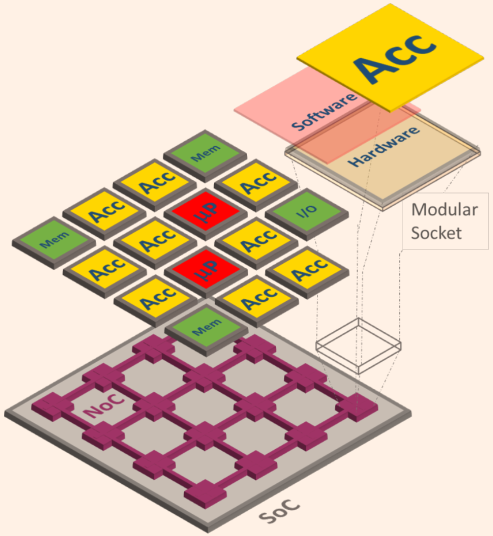
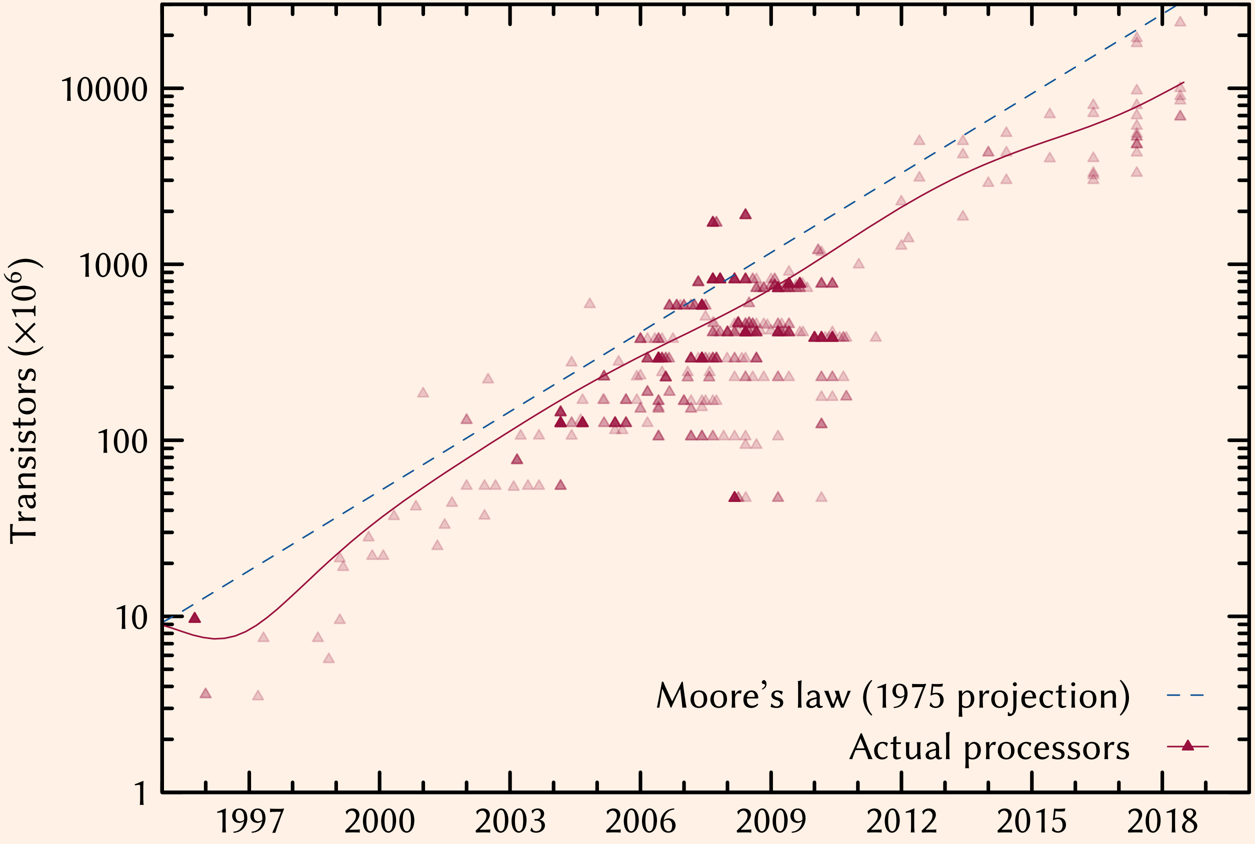
Scalable Emulation of Heterogeneous Systems



Emilio G. Cota
Columbia University

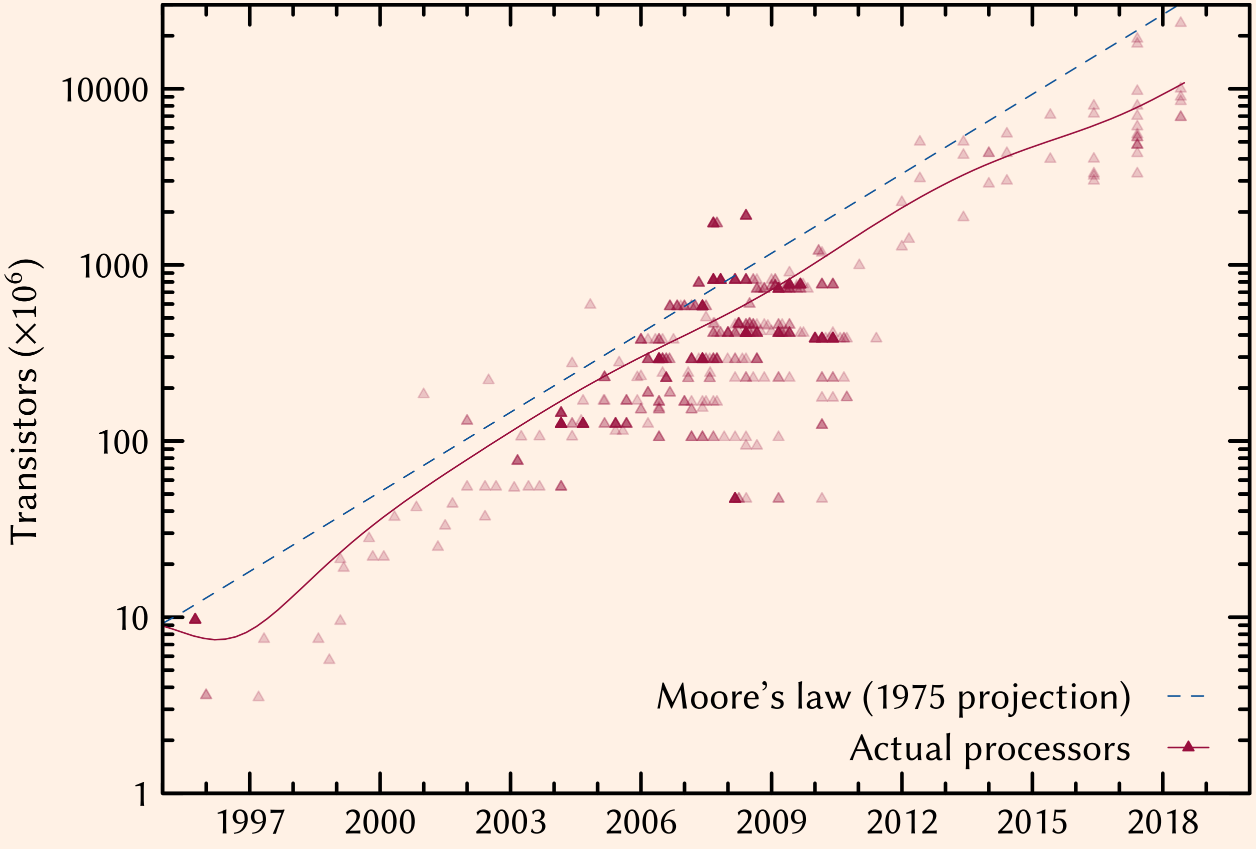
PhD Dissertation Defense
March 14, 2019

Moore's Law



Data: CPUDB, Intel ARK, Wikipedia: https://en.wikipedia.org/wiki/Transistor_count

Moore's Law

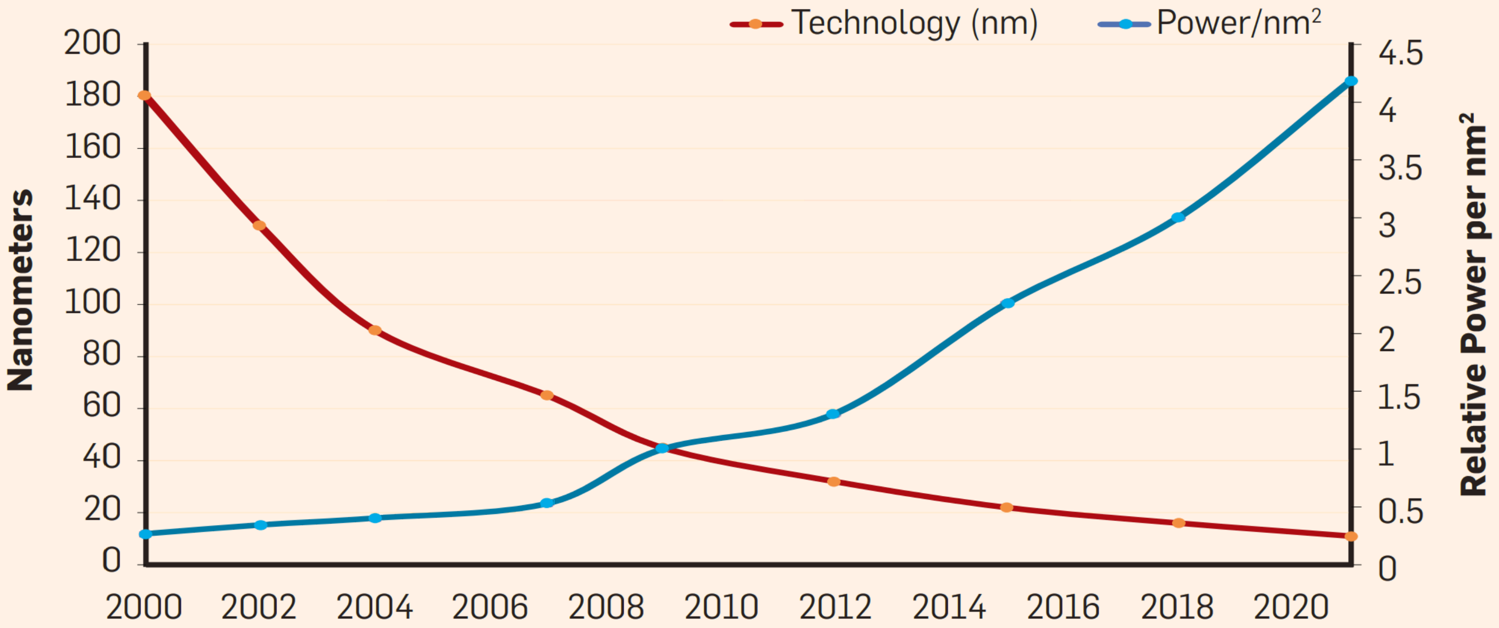


isn't dead

but might be slowing down

Data: CPUDB, Intel ARK, Wikipedia: https://en.wikipedia.org/wiki/Transistor_count

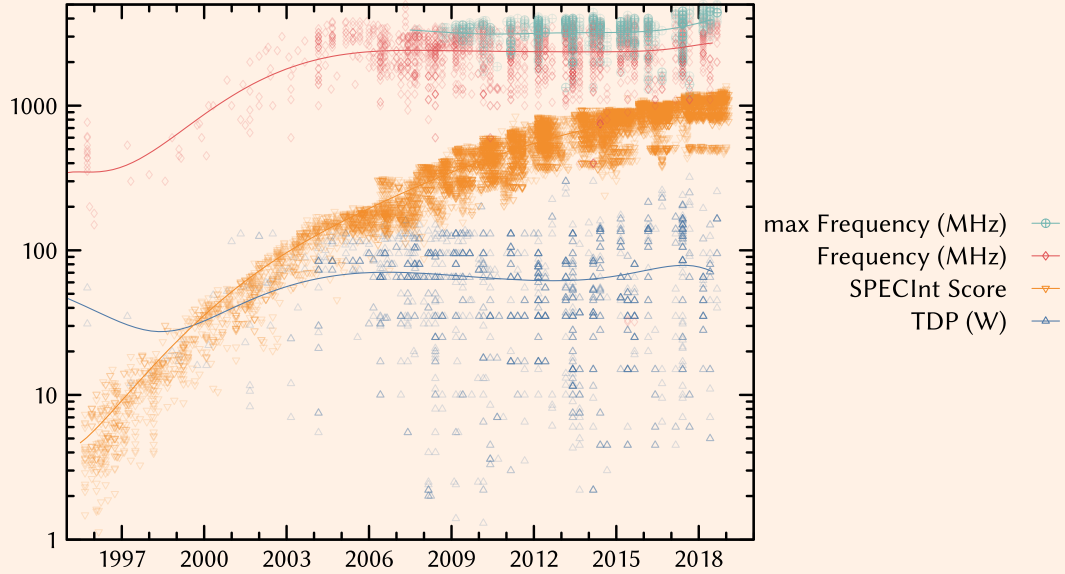
Dennard Scaling *is* dead

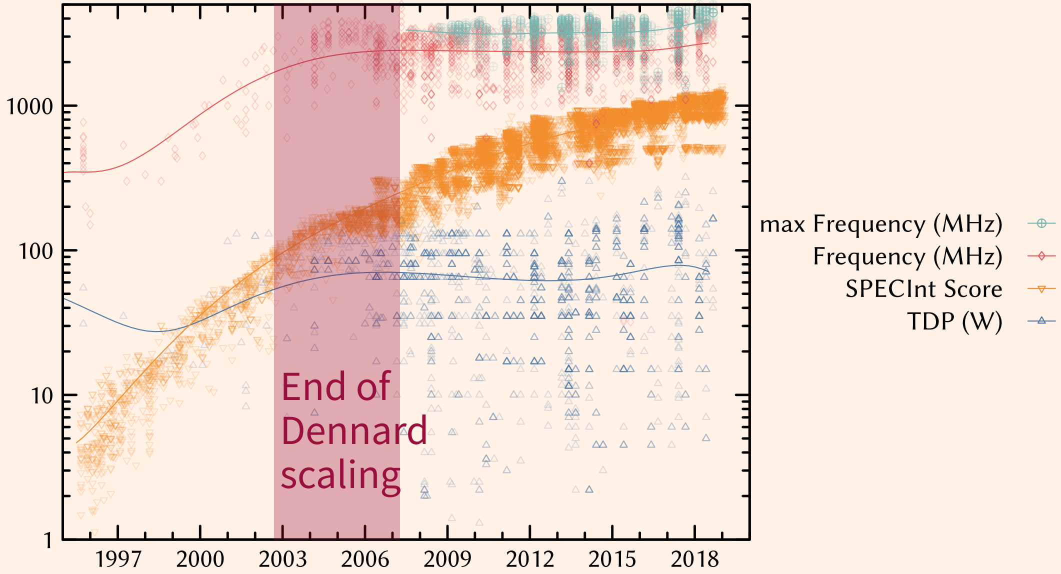


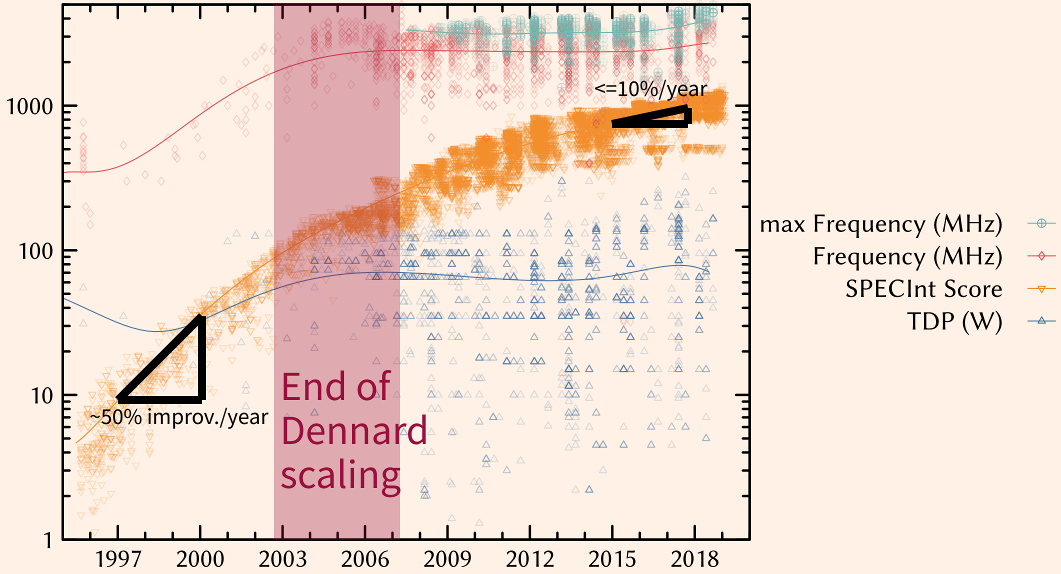
power density increasing since the mid-00's

Plot from 2018 Turing Lecture by Hennessy & Patterson

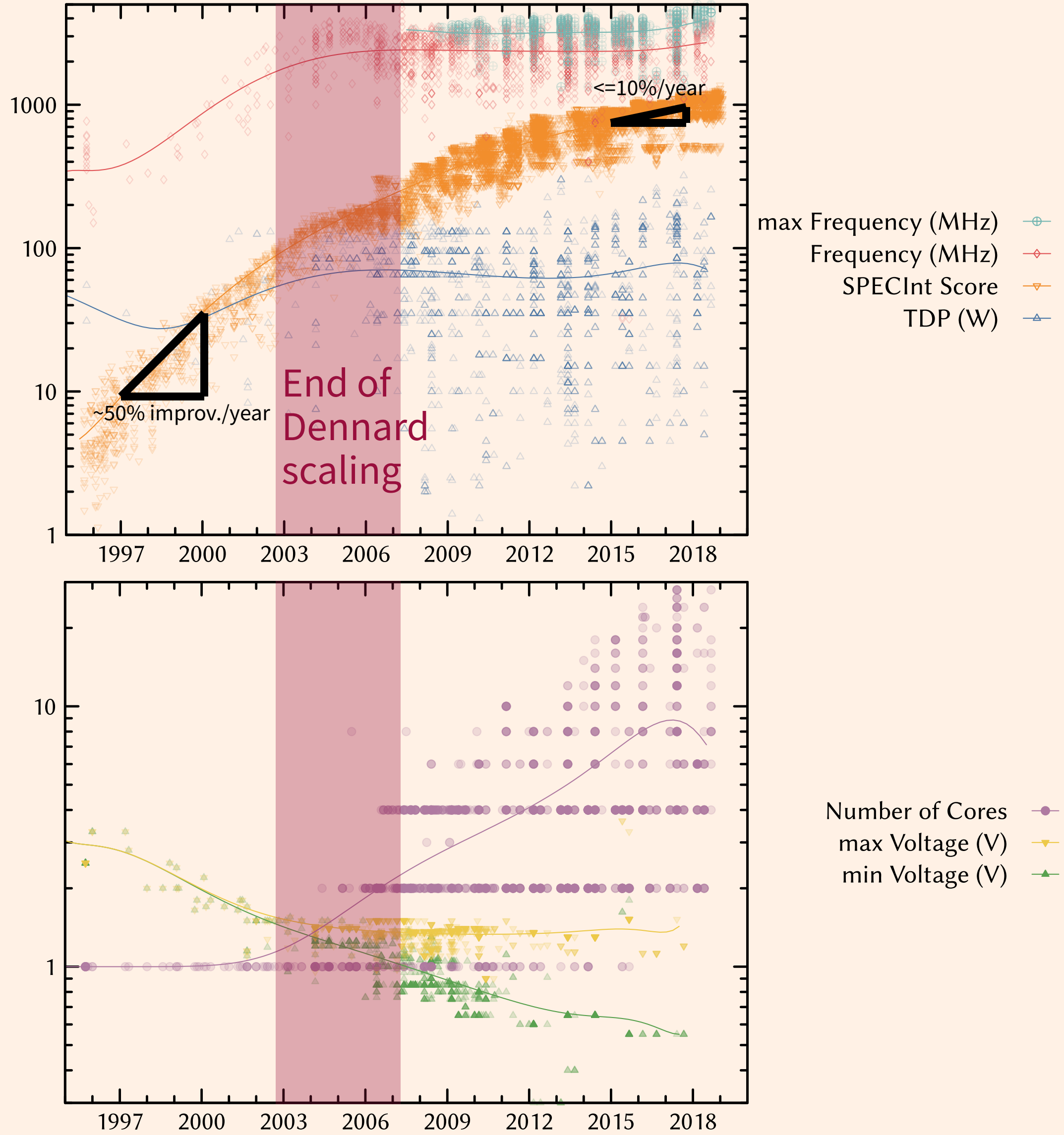
Data based on models in Esmaeilzadeh et al., "Dark Silicon and the End of Multicore Scaling", ISCA'11


1000
100
10
1
1997
2000
2003
2006
2009
2012
2015
2018
max Frequency (MHz)
Frequency (MHz)
SPECInt Score
TDP (W)


1000
100
10
1
1997
2000
2003
2006
2009
2012
2015
2018
End of Dennard scaling
max Frequency (MHz)
Frequency (MHz)
SPECInt Score
TDP (W)



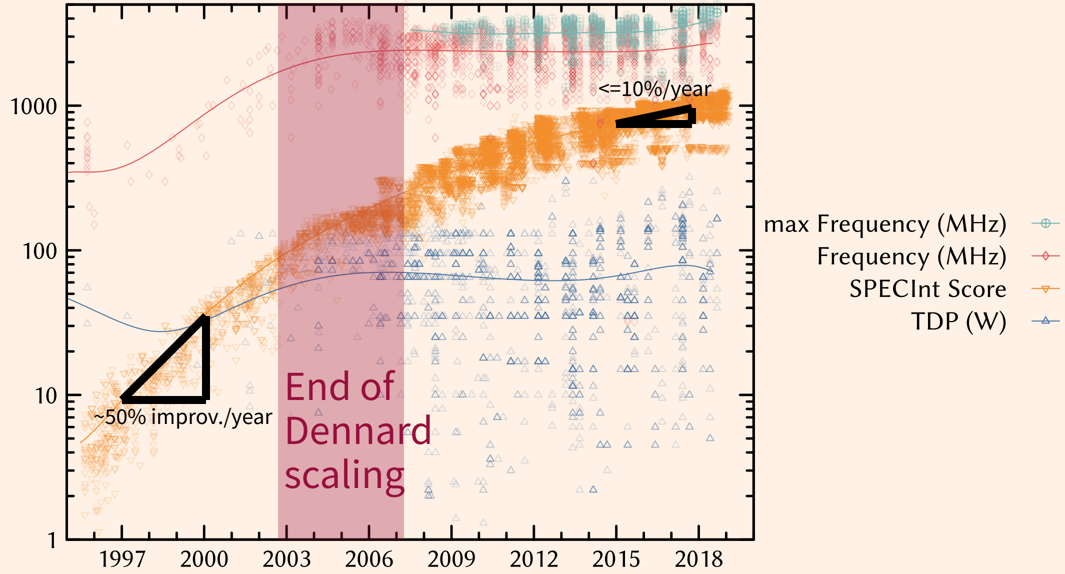
1000
100
10
1
1997
2000
2003
2006
2009
2012
2015
2018
~50% improv./year
<=10%/year
End of Dennard scaling
max Frequency (MHz)
Frequency (MHz)
SPECInt Score
TDP (W)

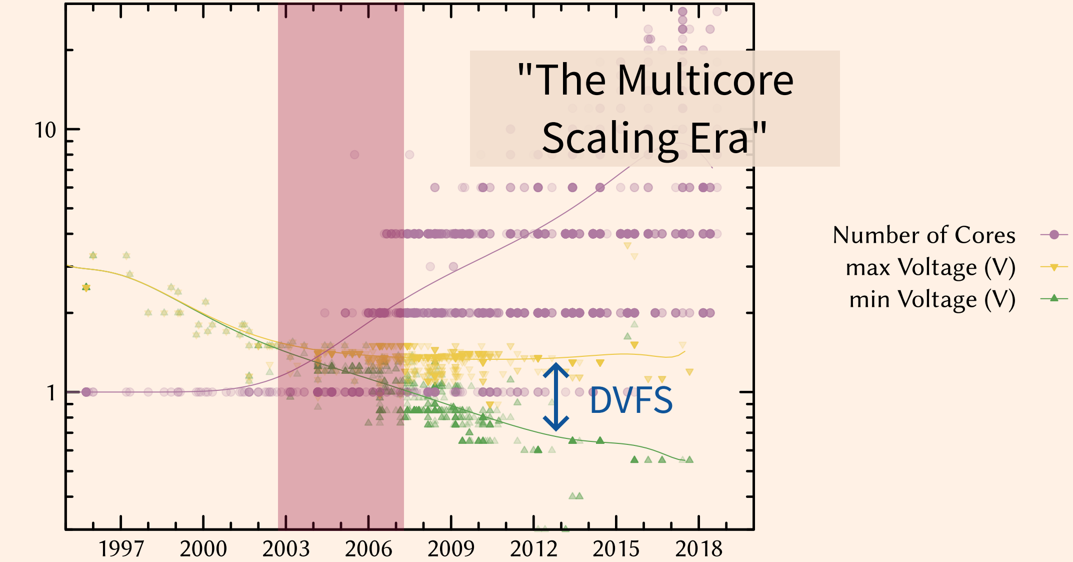


<=10%/year

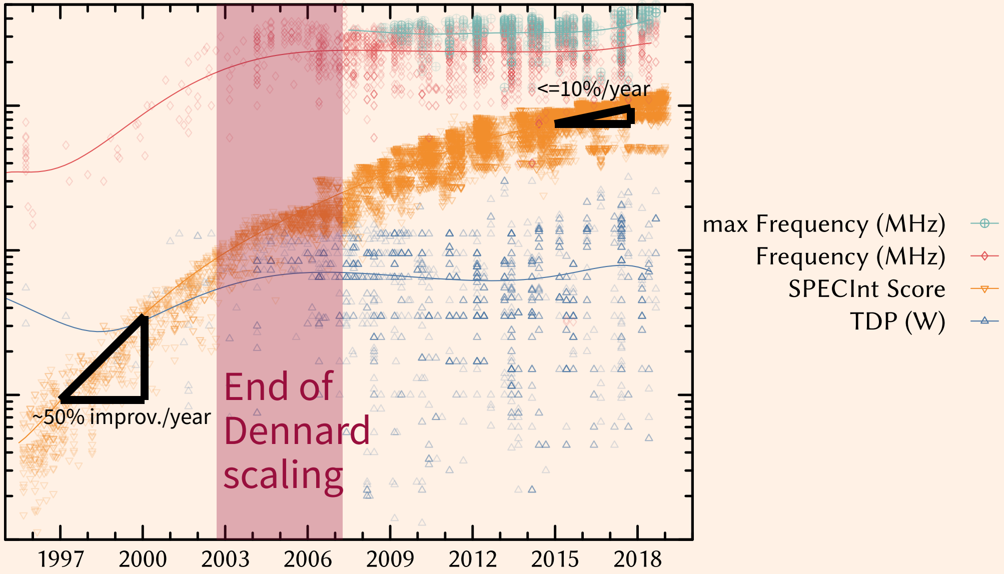
~50% improv./year

End of Dennard scaling





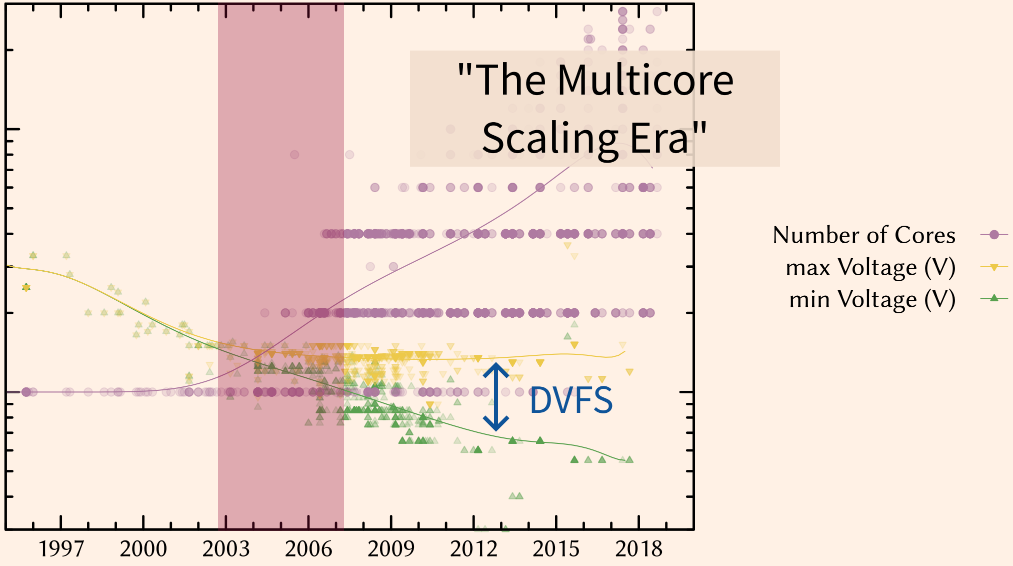
"The Multicore Scaling Era"



~50% improv./year

<=10%/year

End of Dennard scaling

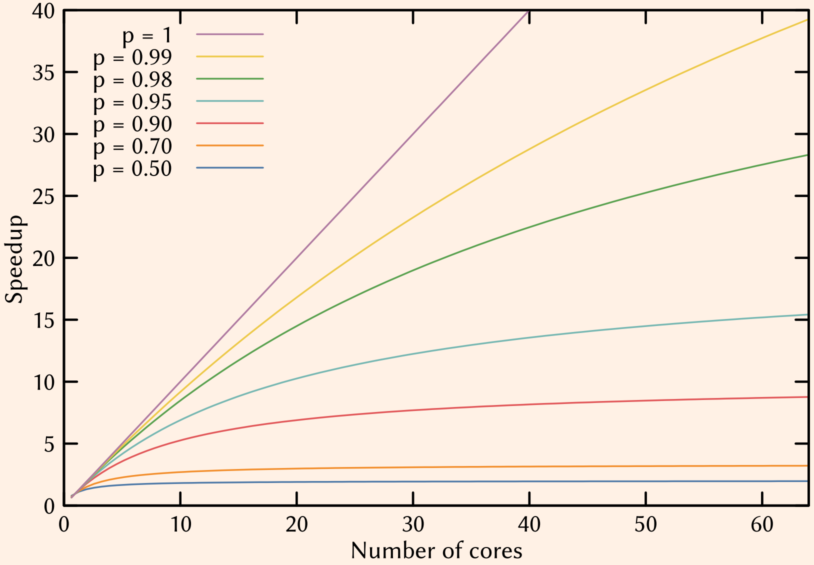


"The Multicore Scaling Era"

DVFS

..but multicores can only take us so far due to Amdahl's law:

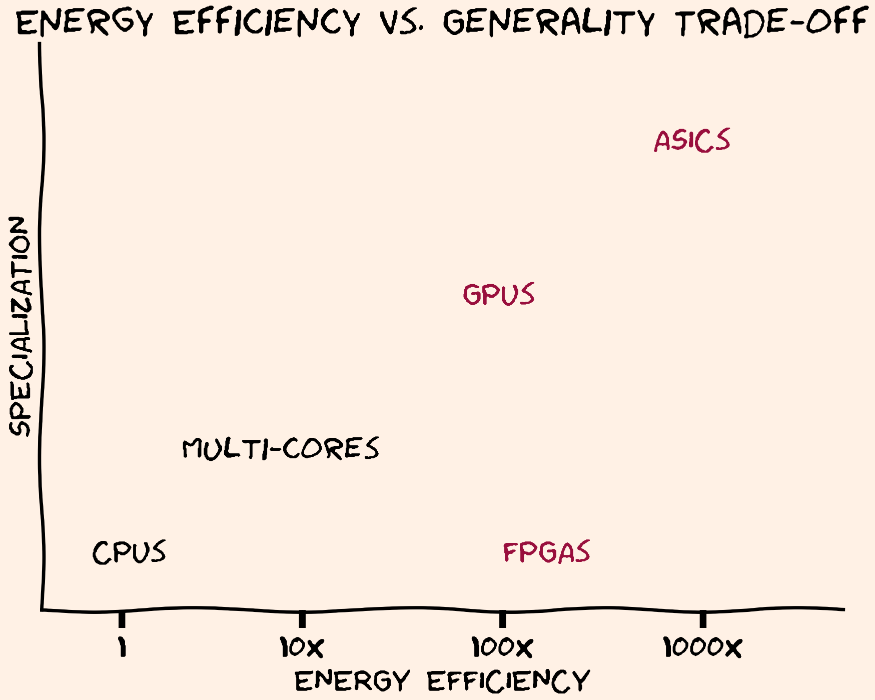
$$Speedup = \frac{1}{(1-p)+\frac{p}{n}}$$



and even if p == 1, multicore scaling will stop due to growing power density: ***growing portions of chips will have to remain powered off (a.k.a. "dark silicon")***

Post-Dennard Scaling Era

Energy efficiency is the key metric



Accelerators

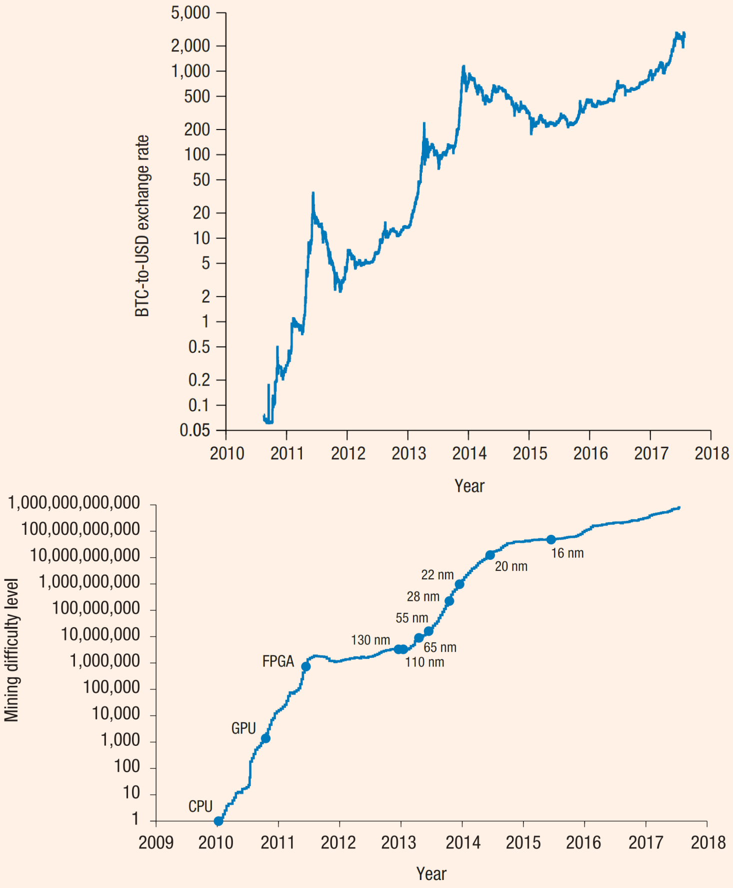
Give up generality for greater efficiency

embrace *dark silicon*: add many accelerators; not all will be on at the same time

Who can afford non-generality?

Accelerators are expensive to develop and deploy, particularly ASICs

Investment can only be amortized for high-demand application domains



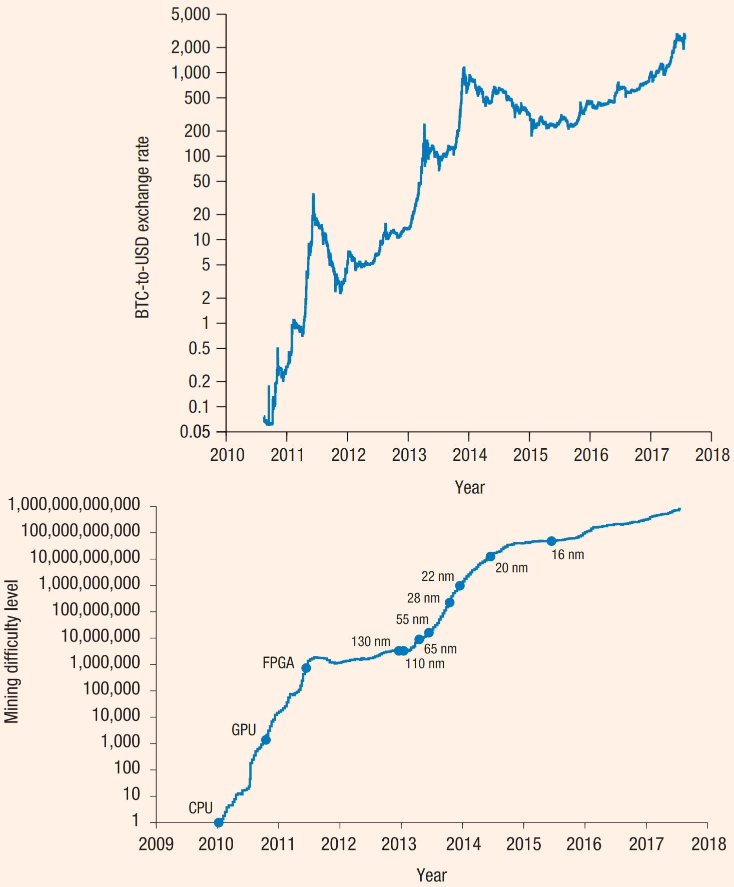
Example: Bitcoin accelerators

Taylor, "The Evolution of Bitcoin Hardware", IEEE Computer 2017

Who can afford non-generality?

Accelerators are expensive to develop and deploy, particularly ASICs

Investment can only be amortized for high-demand application domains

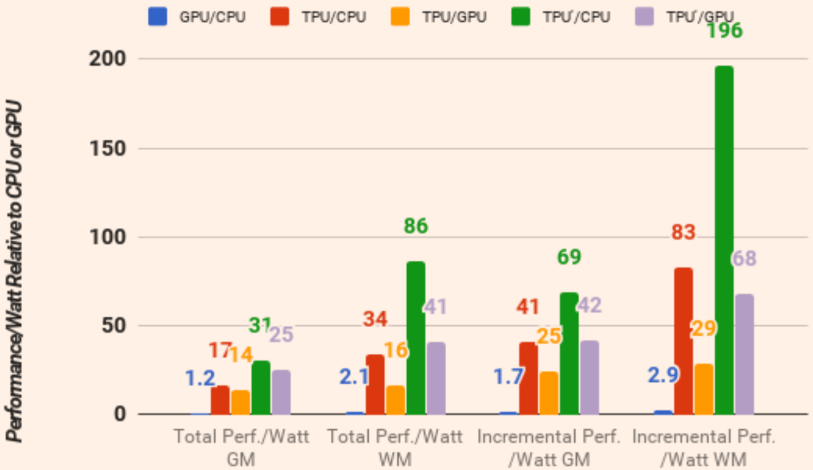


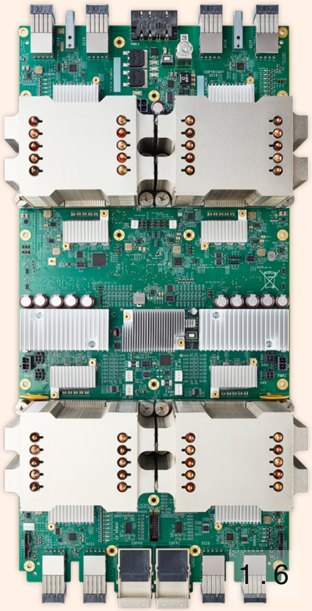
Example: Bitcoin accelerators

Taylor, "The Evolution of Bitcoin Hardware", IEEE Computer 2017

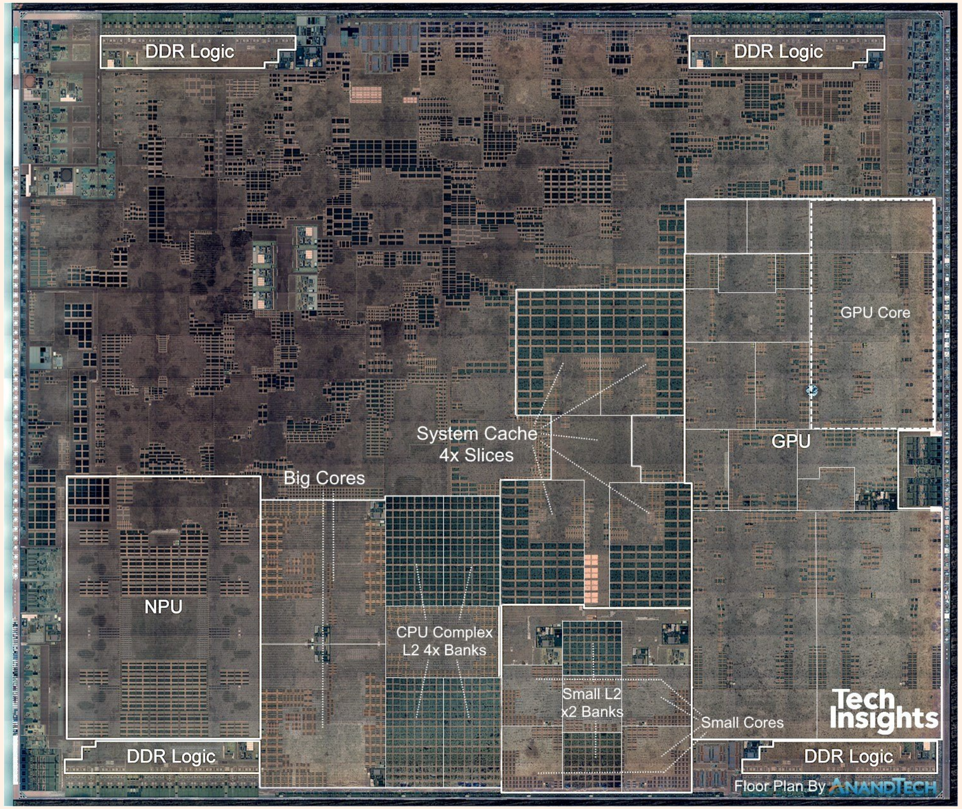
Example: TPU for neural networks

Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA'17





Today's systems are increasingly Heterogeneous



Sources: techinsights.com "Apple iPhone XS Max Teardown", anandtech.com

Heterogeneous systems integrate general-purpose cores with accelerators

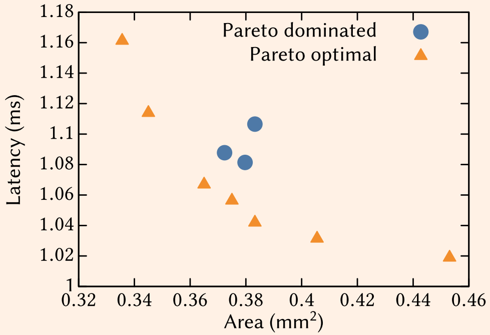
With further transistor scaling, increasing portions of the chip will be devoted to accelerators

Example: Apple A12 7nm SoC ("Iphone X/XS")

Heterogeneous Systems' Challenges

Accelerator design

✓ High-level synthesis tools enable productive design space exploration



Accelerator Integration

✗ Unused *accelerators* incur a large *opportunity cost*

System-level evaluation

✗ Simulators for heterogeneous systems are limited by a *lack of fast, scalable emulators*

Heterogeneous systems'

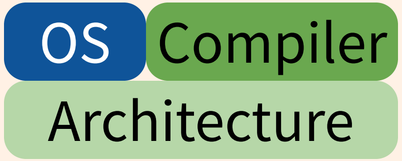
Emulation Requirements

Accelerator modeling

From RTL and/or high-level synthesis descriptions

Full-system

Accelerators might affect the hardware-software interface, e.g. virtual memory or I/O



Portable, cross-ISA

Accelerators might require ISA innovations

Performance

Leverage multi-core hosts while maintaining correctness

Thesis

"Fast, scalable machine emulation is feasible and useful for evaluating the design and integration of heterogeneous systems"

Contributions Thesis

"Fast, scalable machine emulation is feasible and useful for evaluating the design and integration of heterogeneous systems"

Cross-ISA Emulation

Pico [CGO'17]

- Design of a scalable, full-system, cross-ISA emulator
- Handling of guest-host ISA differences in atomic instructions

Qelt [VEE'19]

- Fast, correct cross-ISA FP emulation leveraging the host FPU
- Fast, cross-ISA instrumentation layer
- Scalable emulation also during heavy code generation

Contributions Thesis

"Fast, scalable machine emulation is feasible and useful for evaluating the design and integration of heterogeneous systems"

Cross-ISA Emulation

Pico
[CGO'17]

- Design of a scalable, full-system, cross-ISA emulator
- Handling of guest-host ISA differences in atomic instructions

Qelt
[VEE'19]

- Fast, correct cross-ISA FP emulation leveraging the host FPU
- Fast, cross-ISA instrumentation layer
- Scalable emulation also during heavy code generation

Accelerator Integration

- Quantitative comparison of accelerator couplings **[DAC'15]**
- Technique to lower the opportunity cost of accelerator integration by reusing acc. memories to extend the LLC **ROCA [CAL'14, ICS'16]**

Pico: Cross-ISA Machine Emulation

goal: efficient, correct, multicore-on-multicore cross-ISA emulation

Artifact * AEC * Evaluated * CGO *

Consistent * Complete * Well Documented * Easy to Reuse *

Cota, Bonzini, Bennée, Carloni. "Cross-ISA Machine Emulation for Multicores", CGO, 2017

1-Minute Emulation Tutorial

Main task:

Fetch -> Decode -> Execute

How? Two options:

1-Minute Emulation Tutorial

Main task:

Fetch -> Decode -> Execute

How? Two options:

1. Interpretation

```
uint8_t *ip;
uint8_t  opcode;

while (true)

   // Read the next token from the instruction stream
   opcode = *ip;

   // Advance to the next byte in the stream
   ip++;

   // Decide what to do
   switch (opcode) {
      case PZT_ADD_32:
         ...
         break;
      case PZT_SUB_32:
         ...
         break;
      ...
   }
}
```

1-Minute Emulation Tutorial

Main task:

Fetch -> Decode -> Execute

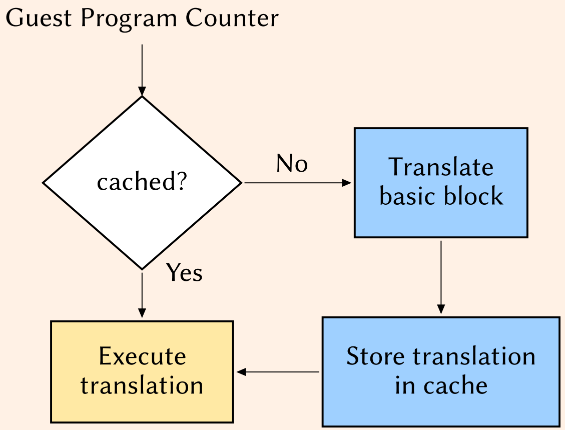
How? Two options:

1. Interpretation

```
uint8_t *ip;
uint8_t  opcode;

while (true)

   // Read the next token from the instruction stream
   opcode = *ip;

   // Advance to the next byte in the stream
   ip++;

   // Decide what to do
   switch (opcode) {
      case PZT_ADD_32:
         ...
         break;
      case PZT_SUB_32:
         ...
         break;
      ...
   }
}
```

2. Dynamic Binary Translation (DBT)

"DBT dispatch loop"



✓ Faster than interpretation

✗ More complex

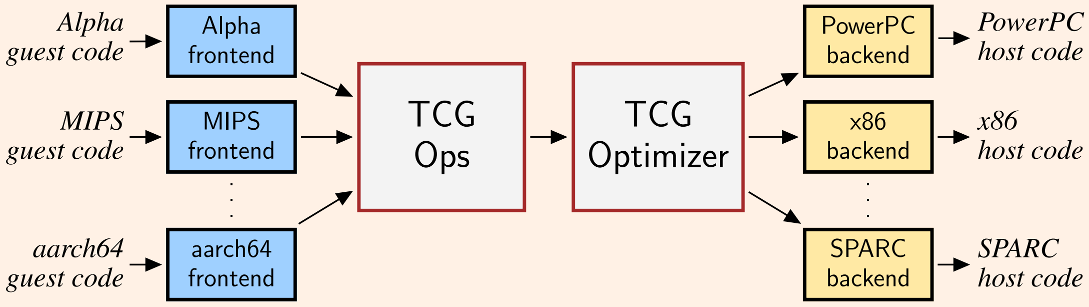
e.g., external "helpers" are needed to deal with complex emulation

Pico makes QEMU* a scalable emulator

Open source: https://www.qemu.org

Widely used in both industry and academia

Supports many ISAs through **DBT** via TCG, its Intermediate Representation (IR):



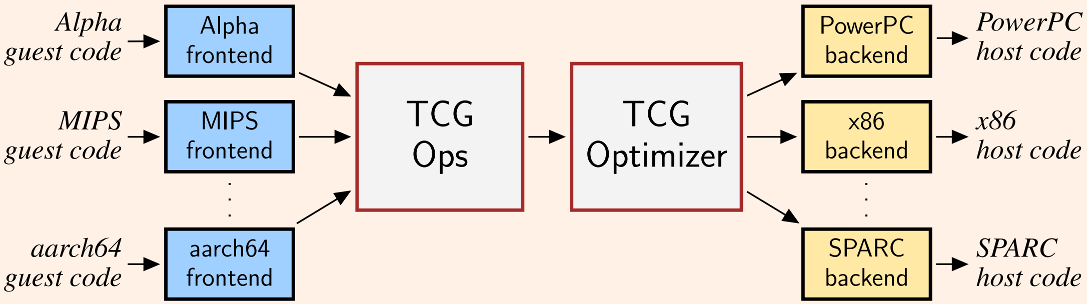
[*] Bellard. "QEMU, a fast and portable dynamic translator", ATC, 2005

Pico makes QEMU* a scalable emulator

Open source: https://www.qemu.org

Widely used in both industry and academia

Supports many ISAs through **DBT** via TCG, its Intermediate Representation (IR):



Our contributions are not QEMU-specific

They are applicable to dynamic binary translators at large

[*] Bellard. "QEMU, a fast and portable dynamic translator", ATC, 2005

Challenges in scalable cross-ISA emulation

(1) Scalability of the DBT engine

(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

Challenges in scalable cross-ISA emulation

(1) Scalability of the DBT engine

(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 [A] and COREMU [B] do not address (2)
- ArMOR [C] solves (2.1)

[A] J. H. Ding et al. PQEMU: A parallel system emulator based on QEMU. ICPADS, 2011
[B] Z. Wang et al. COREMU: A scalable and portable parallel full-system emulator. PPoPP, 2011
[C] D. Lustig et al. ArMOR: defending against memory consistency model mismatches in heterogeneous architectures. ISCA, 2015

Challenges in scalable cross-ISA emulation

(1) Scalability of the DBT engine

(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

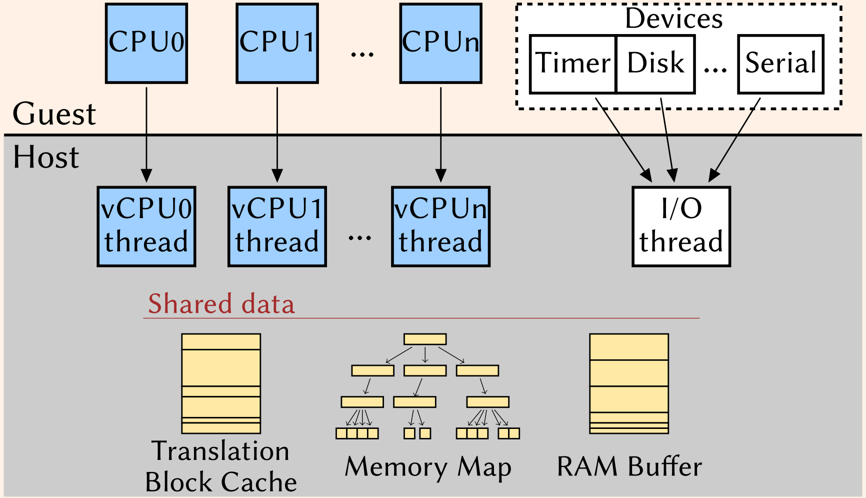
Related Work:

- PQEMU [A] and COREMU [B] do not address (2)
- ArMOR [C] solves (2.1)

Pico's contributions: (1) & (2.2)

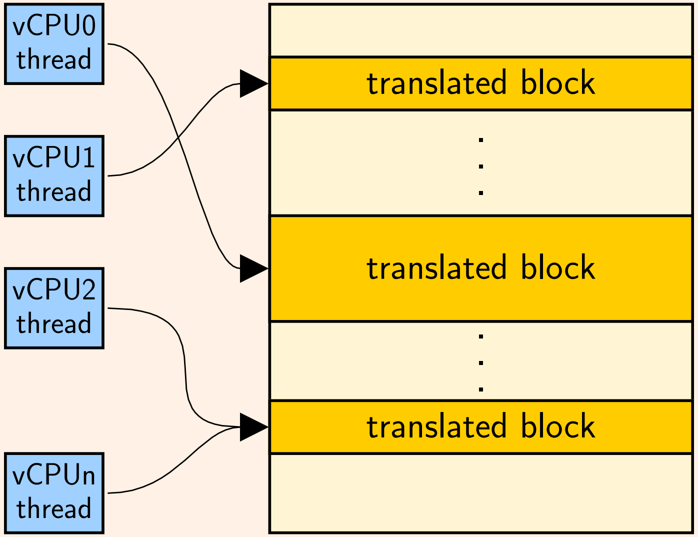
[A] J. H. Ding et al. PQEMU: A parallel system emulator based on QEMU. ICPADS, 2011
[B] Z. Wang et al. COREMU: A scalable and portable parallel full-system emulator. PPoPP, 2011
[C] D. Lustig et al. ArMOR: defending against memory consistency model mismatches in heterogeneous architectures. ISCA, 2015

Pico's Architecture



- One host thread per guest CPU
  - Instead of emulating guest CPUs one at a time
- Key data structure: translation block cache

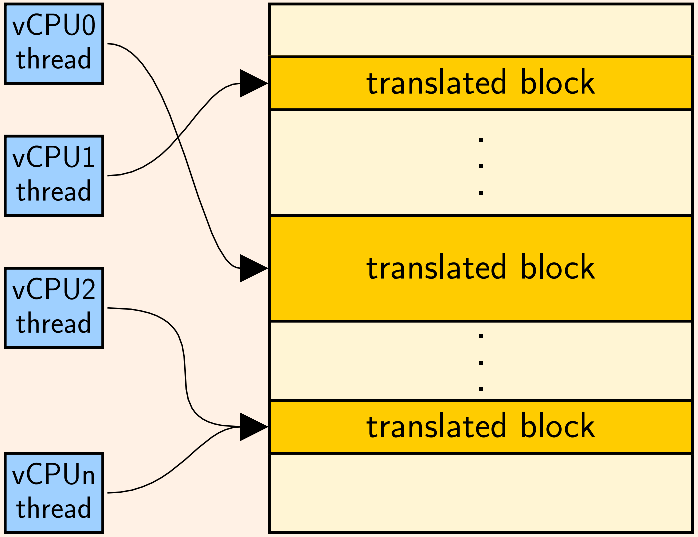
Translation Block (TB) Cache



- Buffers TBs to amortize translation cost
- Shared by all vCPUs to minimize code duplication
    - see [*] for a private vs. shared cache comparison

[*] Bruening, Kiriansky, Garnett, Banerji. "Thread-shared software code caches", CGO, 2006

Translation Block (TB) Cache

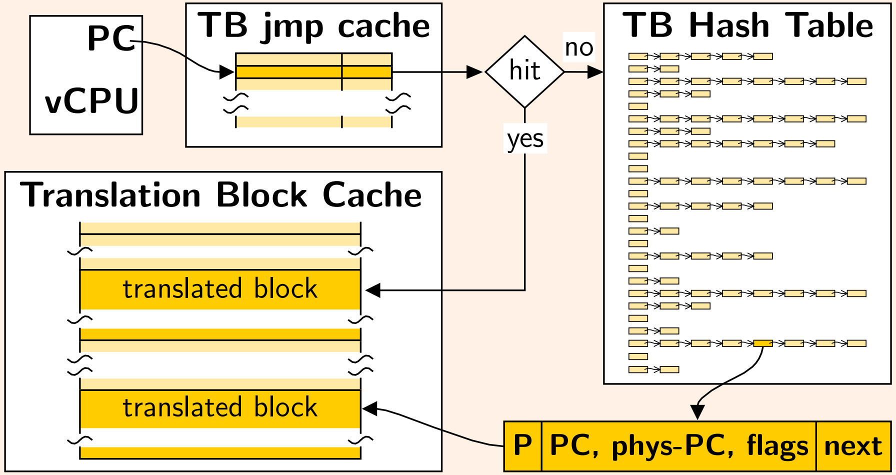


- Buffers TBs to amortize translation cost
- Shared by all vCPUs to minimize code duplication
    - see [*] for a private vs. shared cache comparison

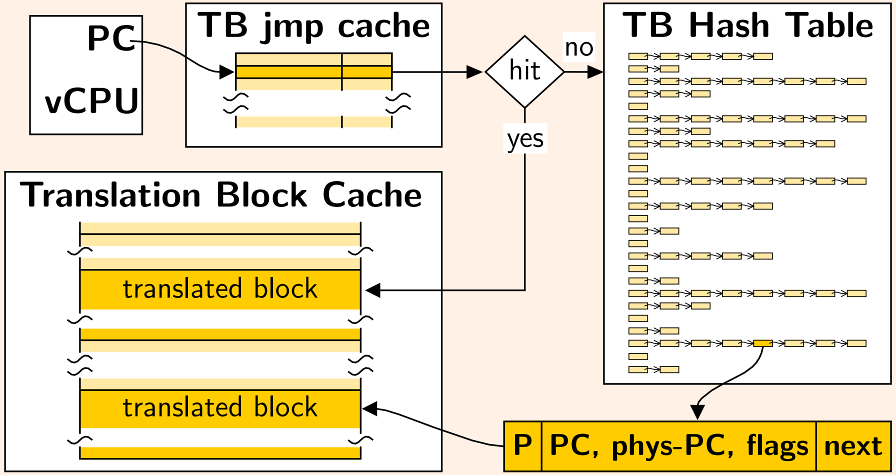
To scale for most workloads, we need concurrent code *execution*

[*] Bruening, Kiriansky, Garnett, Banerji. "Thread-shared software code caches", CGO, 2006

QEMU's Translation Block Cache

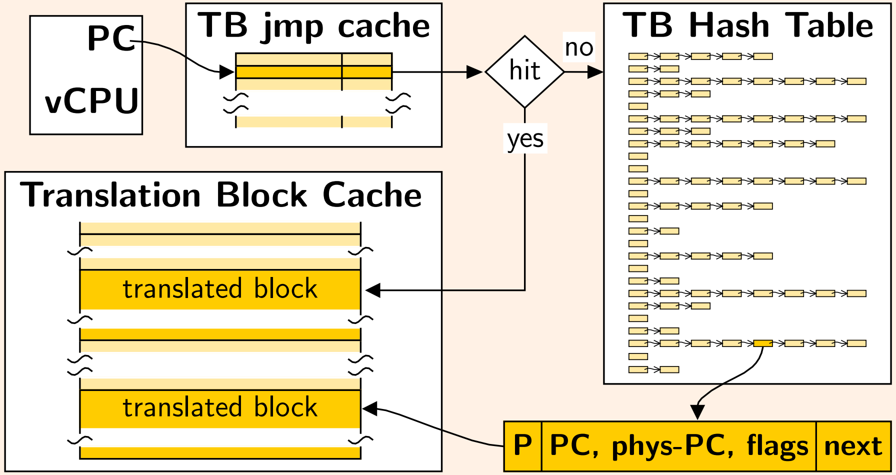


QEMU's Translation Block Cache



Problems in QEMU's TB hash table

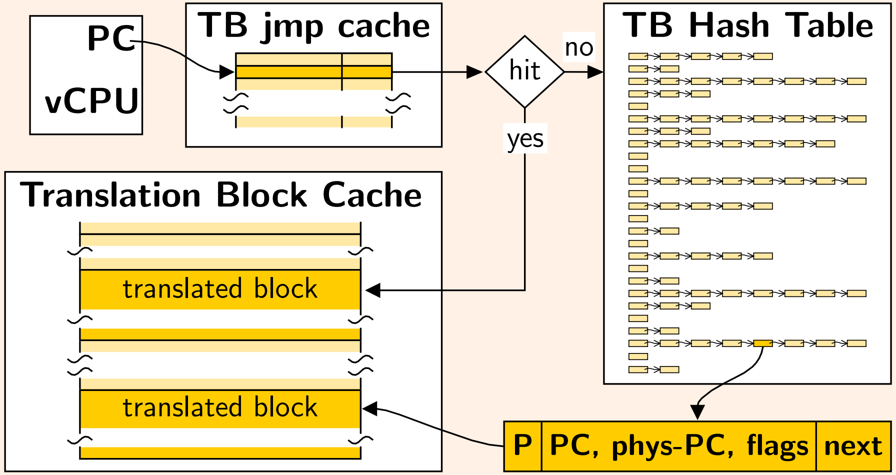
QEMU's Translation Block Cache



Problems in QEMU's TB hash table

- Long hash chains: slow lookups
  - Fixed number of buckets
  - hash=h(phys_addr) leads to uneven chain lengths

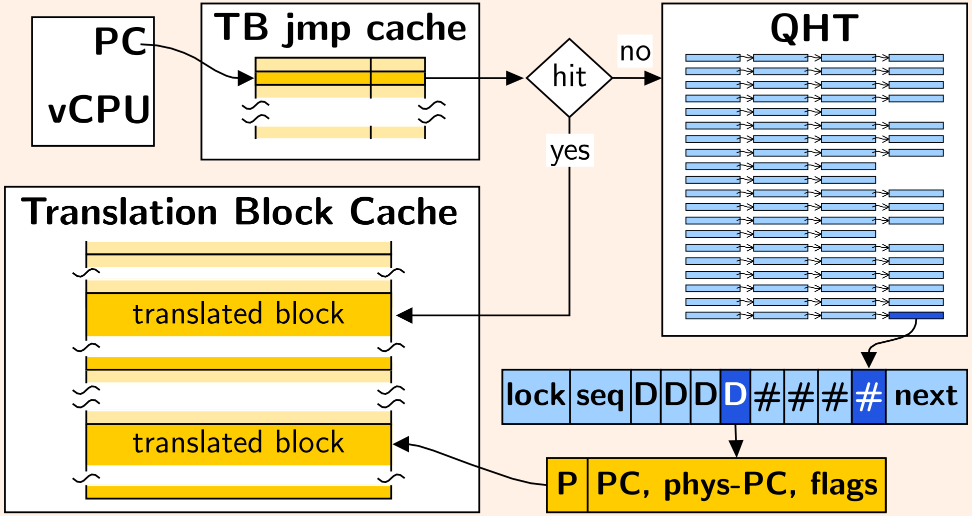
QEMU's Translation Block Cache



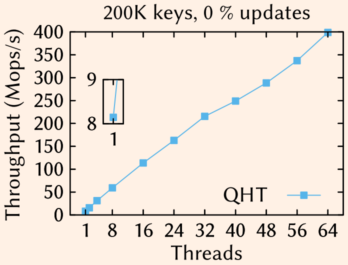
Problems in QEMU's TB hash table

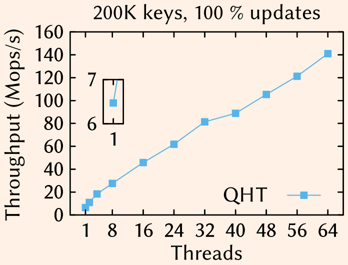
- Long hash chains: slow lookups
  - Fixed number of buckets
  - hash=h(phys_addr) leads to uneven chain lengths
- No support for **concurrent lookups**

Pico's Translation Block Cache

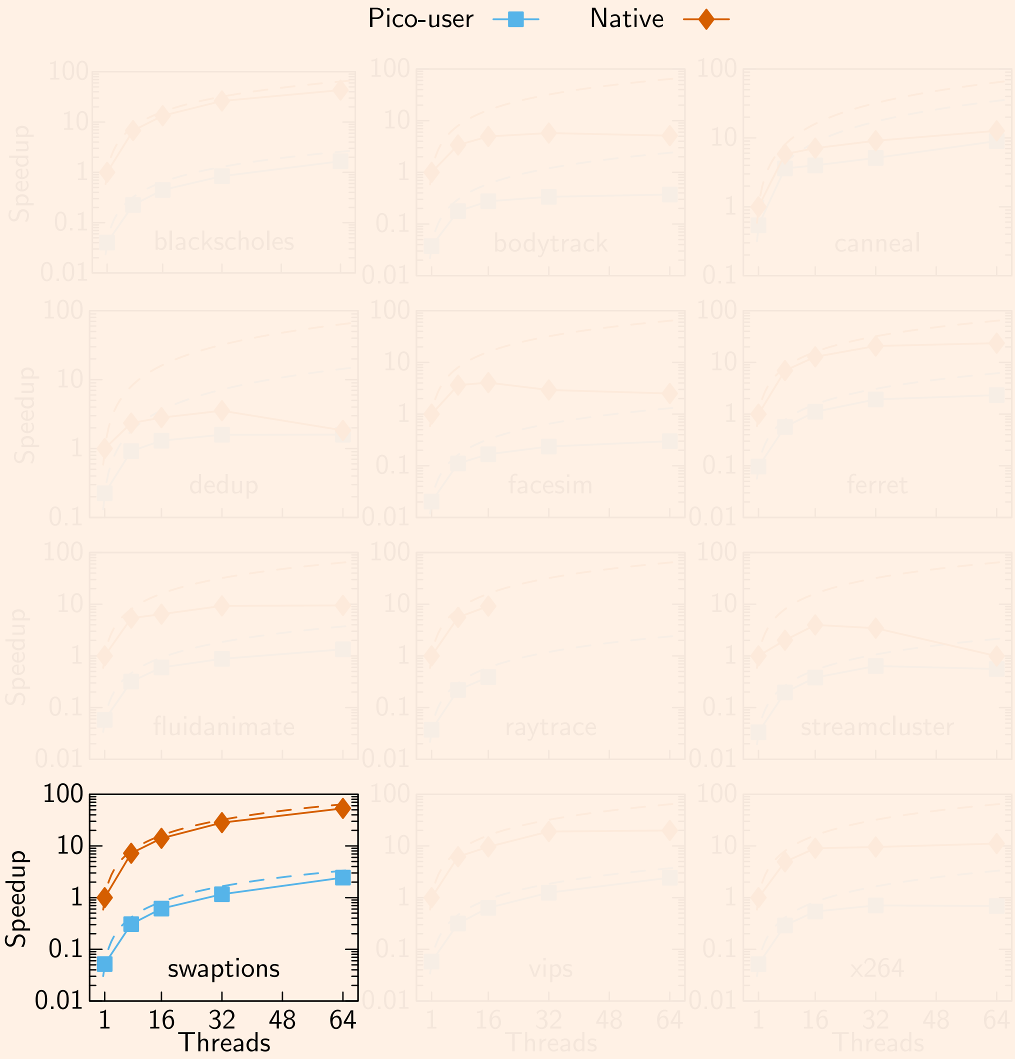


- *hash=h(phys_addr,* ***phys_PC, cpu_flags****)*: uniform chain distribution
  - e.g. longest chain down from 550 to 40 TBs when booting ARM Linux
- **QHT**: A resizable, scalable hash table
  - scales for both reads & writes
- Keeps QEMU's global lock for code translation
  - Translation is rare, but more on this later!



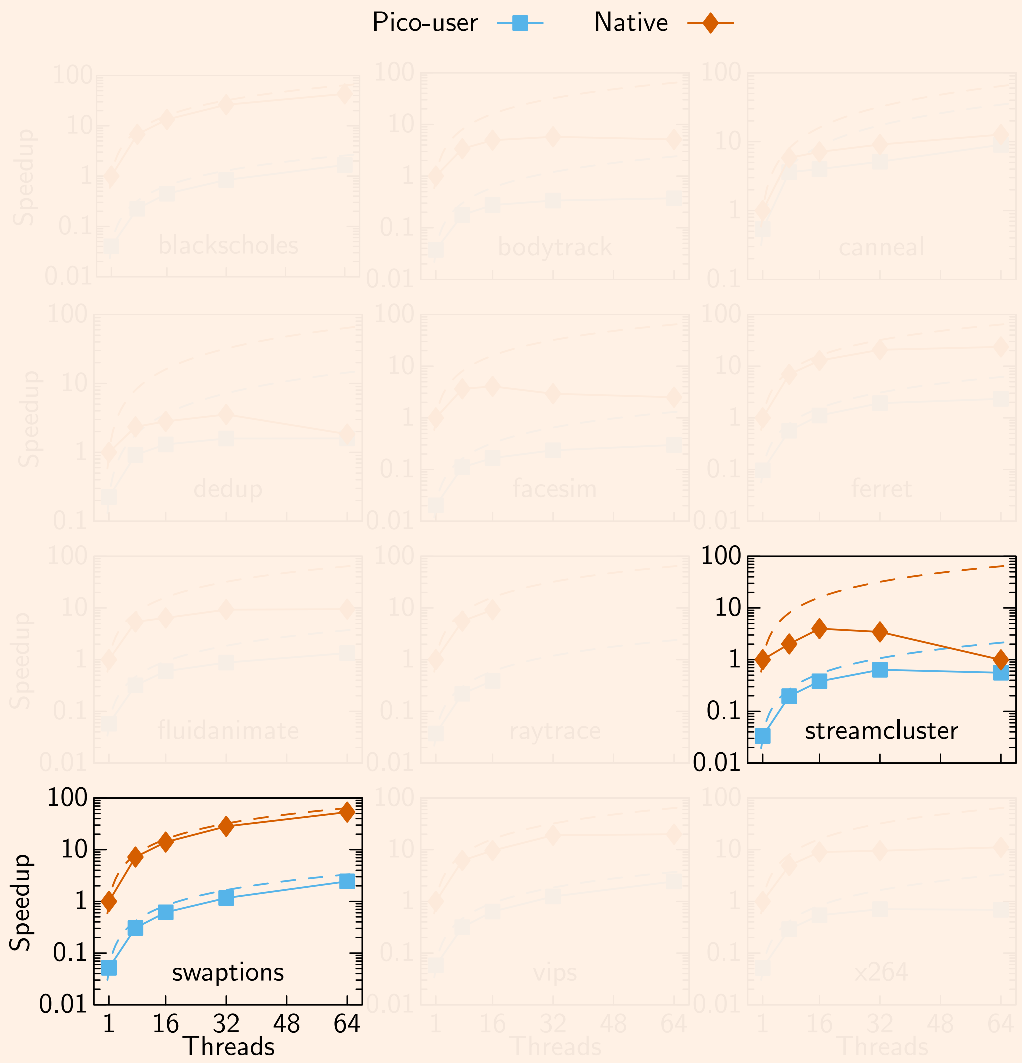


Parallel Performance (x86-on-x86)



- Speedup normalized over native's single-threaded perf
- Dashed: Ideal scaling
- QEMU-user not shown: does not scale at all

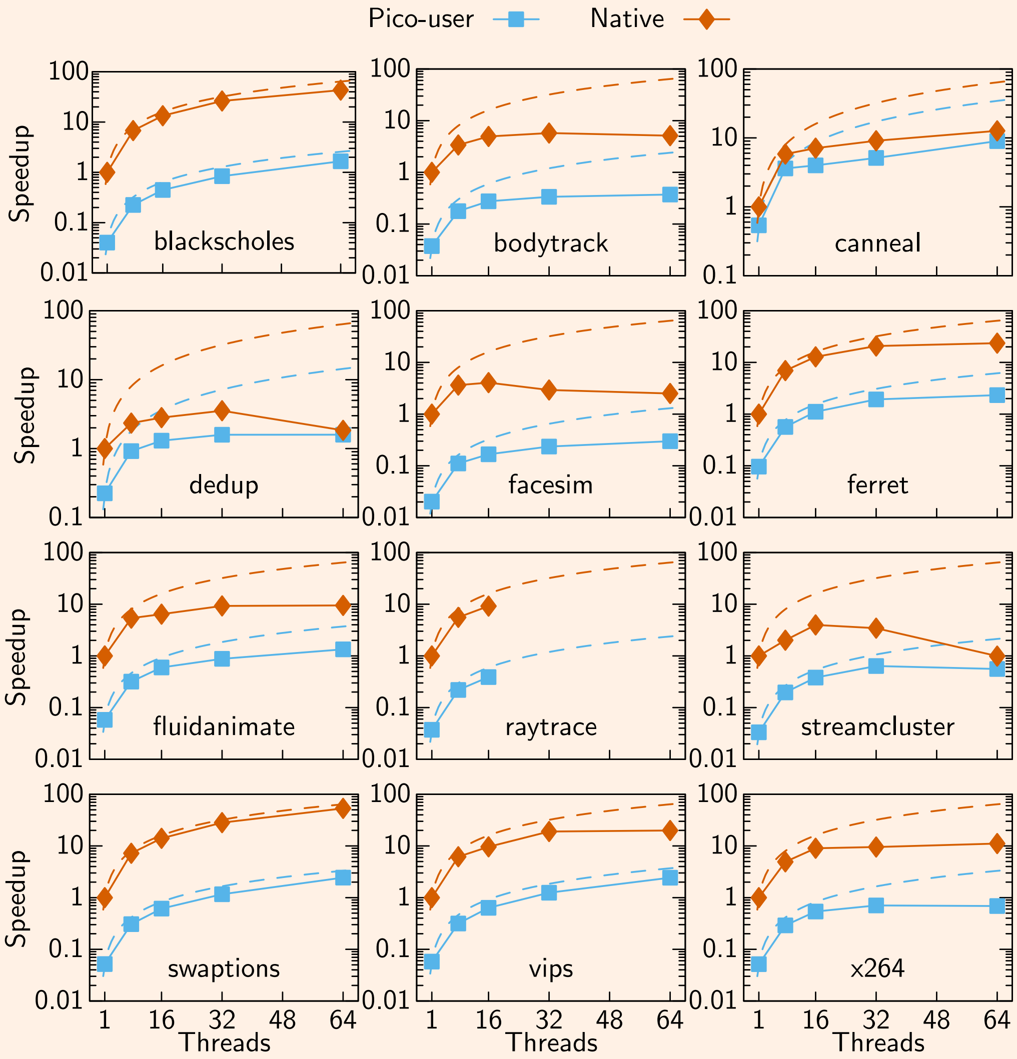
Parallel Performance (x86-on-x86)



- 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*
  - DBT slowdown delays scalability collapse

[*] Southern, Renau. "Deconstructing PARSEC scalability", WDDD, 2015

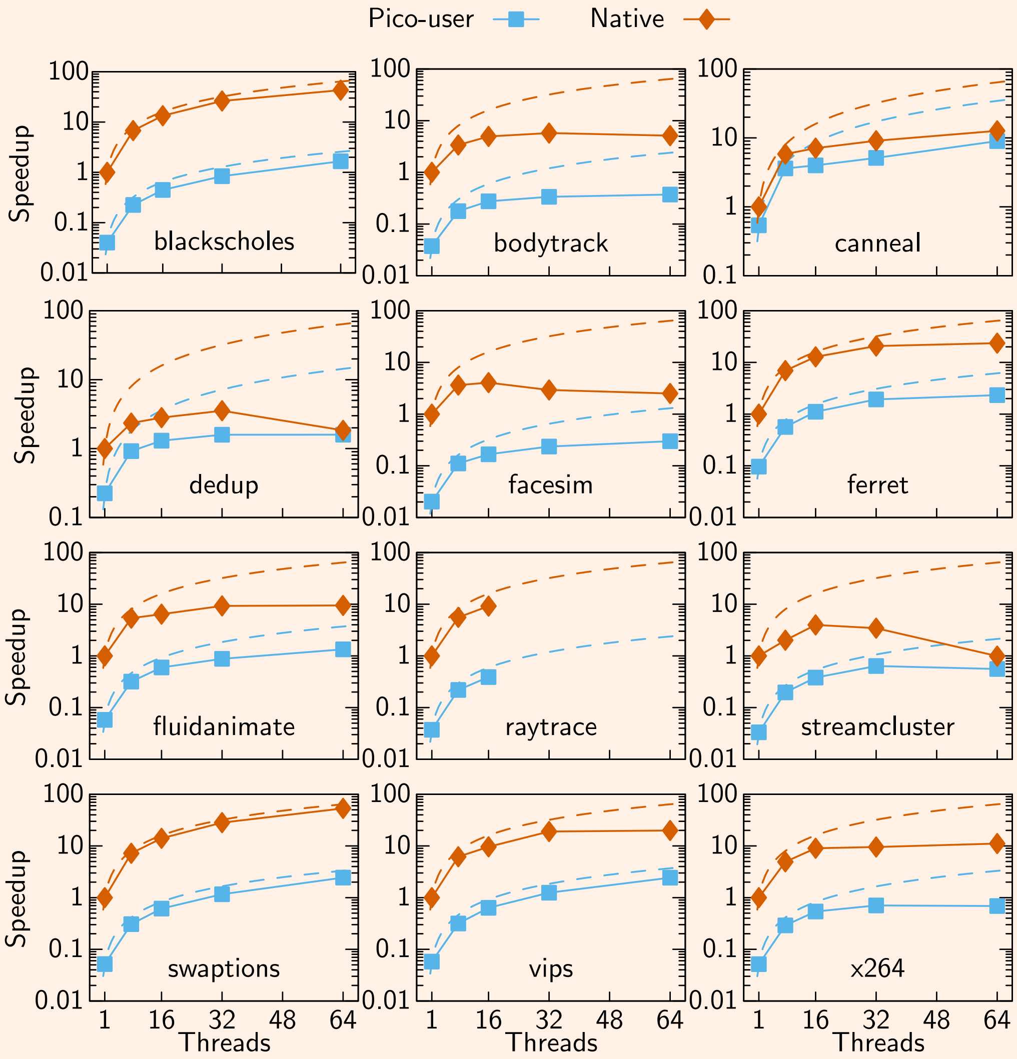
Parallel Performance (x86-on-x86)



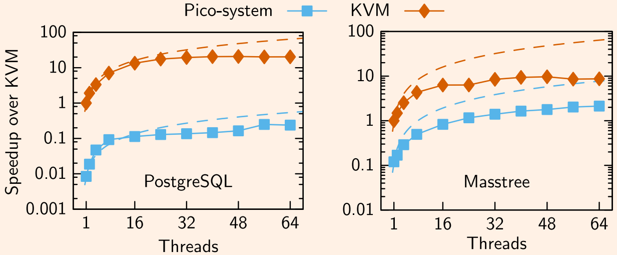
- Speedup normalized over native's single-threaded perf
- Dashed: Ideal scaling
- QEMU-user not shown: does not scale at all
- Pico scales better than Native
  - PARSEC known not to scale to many cores*
  - DBT slowdown delays scalability collapse

[*] Southern, Renau. "Deconstructing PARSEC scalability", WDDD, 2015

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*
  - DBT slowdown delays scalability collapse
- Similar trends for server workloads



[*] Southern, Renau. "Deconstructing PARSEC scalability", WDDD, 2015

Atomic Operations

Two families:

Compare-and-Swap (CAS)

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

x86/IA-64:	`cmpxchg`

Load Locked-Store Conditional (LL/SC)

```
/*
 * 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);
```

Alpha:	`ldl_l/stl_c`
POWER:	`lwarx/stwcx`
ARM:	`ldrex/strex`
aarch64:	`ldaxr/strlxr`
MIPS:	`ll/sc`
RISC-V:	`lr/sc`

Atomic Operations

Two families:

Compare-and-Swap (CAS)

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

x86/IA-64:	`cmpxchg`

Load Locked-Store Conditional (LL/SC)

```
/*
 * 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);
```

Alpha:	`ldl_l/stl_c`
POWER:	`lwarx/stwcx`
ARM:	`ldrex/strex`
aarch64:	`ldaxr/strlxr`
MIPS:	`ll/sc`
RISC-V:	`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

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

Only a few simple instructions are allowed between LL and SC

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

Only a few simple instructions are allowed between LL and SC

LL/SC on CAS: Not trivial

Solving this solves LL/SC on LL/SC, because LL/SC is stronger than CAS

However, there's the **ABA problem**

ABA Problem

Init: *addr = A;

time ↓

cpu0	**cpu1**

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

SC fails, regardless of the contents of *addr

ABA Problem

Init: *addr = A;

time

cpu0	cpu1
do {	
val = **load_exclusive**(addr); /* reads A */	
…	atomic_set(addr, B);
…	atomic_set(addr, A);
} while (**store_exclusive**(addr, newval);	

SC fails, regardless of the contents of *addr

time

cpu0	cpu1
do {	
val = atomic_read(addr); /* reads A */	
…	atomic_set(addr, B);
…	atomic_set(addr, A);
} while (**CAS**(addr, val, newval);	

CAS succeeds where SC failed!

Pico's Emulation of Atomics

3 proposed options that scale:

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"

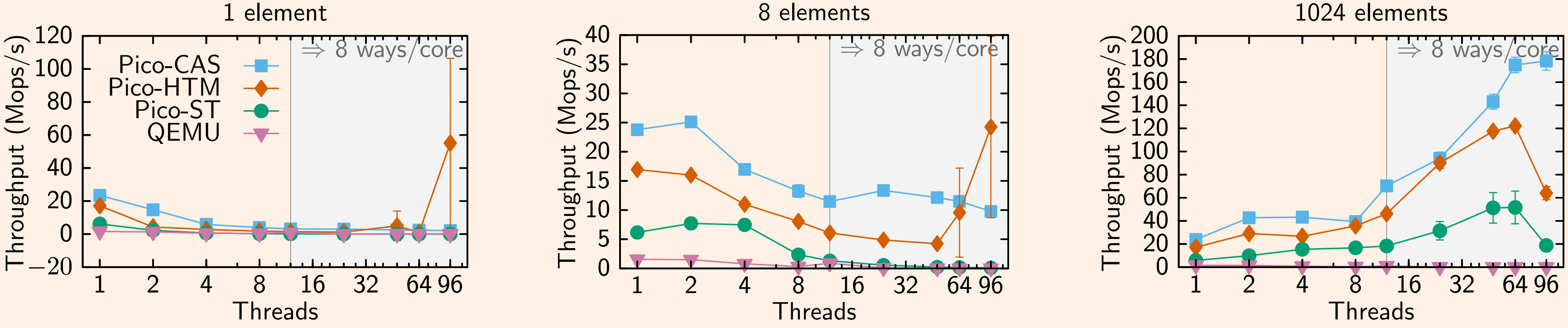
- Correct, scalable & portable
- Perf penalty due to instrumenting regular stores

3. Pico-HTM: Leverages hardware transactional memory (HTM) extensions

- Correct & scalable
- No need to instrument regular stores
    - But requires HTM support on the host

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

Qelt: Cross-ISA Machine Instrumentation

goal: fast, scalable instrumentation of a machine emulator

Cota, Carloni. "Cross-ISA Machine Instrumentation using Fast and Scalable Dynamic Binary Translation", VEE, 2019

Recall our motivation:

Fast, cross-ISA, full-system *instrumentation*

How fast?

- Goal: match Pin's speed when using it for simulation
  - Note that Pin is same-ISA, user-only

Recall our motivation:
Fast, cross-ISA, full-system *instrumentation*

How fast?

- Goal: match Pin's speed when using it for simulation
  - Note that Pin is same-ISA, user-only

How to get there? Need to:

- **Increase emulation speed**
  - Pico is slower than Pin, particularly for full-system and FP workloads
  - Pico does not scale for workloads that translate a lot of code in parallel, e.g. parallel compilation
- **Support cross-ISA instrumentation of the guest**

Qelt's contributions

Emulation Speed

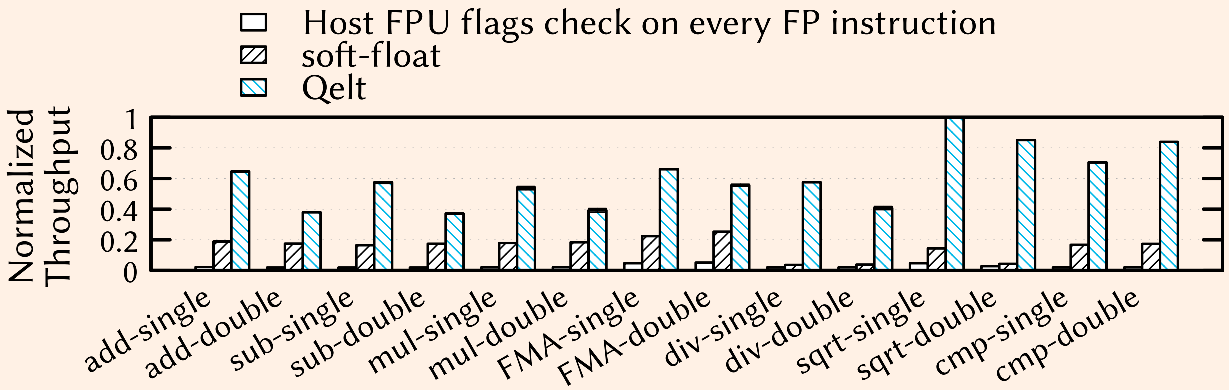
1. Correct cross-ISA **FP emulation** using the host FPU
2. Integration of two state-of-the-art optimizations:
   - indirect branch handling
   - dynamic sizing of the **software TLB**
3. Make the DBT engine **scale** under heavy **code *translation***
   - Not just during ***execution***, like Pico

Instrumentation

4. Fast, ISA-agnostic instrumentation layer for QEMU

1. Cross-ISA FP Emulation

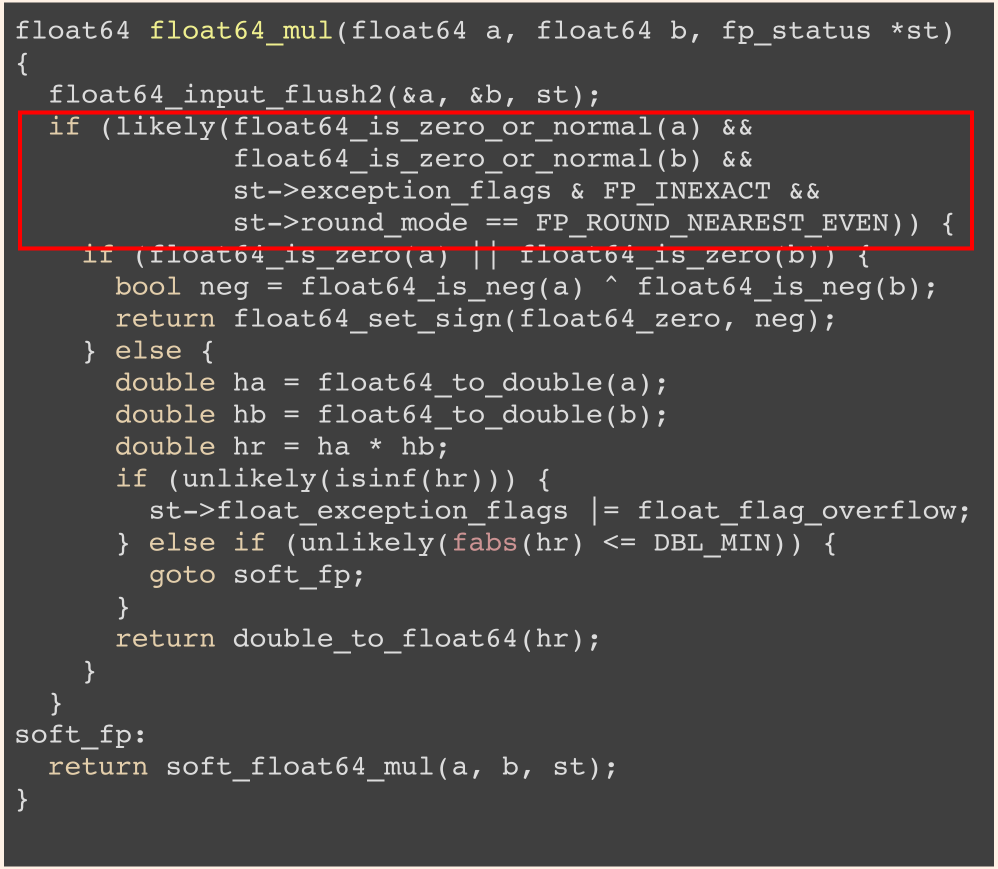
- Rounding, NaN propagation, exceptions, etc. have to be emulated correctly
- Reading the host FPU flags is *very* expensive
  - soft-float is faster, which is why QEMU uses it



baseline (incorrect): always uses the host FPU and never reads excp. flags

- Qelt uses the host FPU for a **subset of FP operations**, *without ever reading the host FPU flags*
  - Fortunately, this subset is **very common**
  - defers to soft-float otherwise

1. Cross-ISA FP Emulation



```
float64 float64_mul(float64 a, float64 b, fp_status *st)
{
  float64_input_flush2(&a, &b, st);
  if (likely(float64_is_zero_or_normal(a) &&
             float64_is_zero_or_normal(b) &&
             st->exception_flags & FP_INEXACT &&
             st->round_mode == FP_ROUND_NEAREST_EVEN)) {
    if (float64_is_zero(a) || float64_is_zero(b)) {
      bool neg = float64_is_neg(a) ^ float64_is_neg(b);
      return float64_set_sign(float64_zero, neg);
    } else {
      double ha = float64_to_double(a);
      double hb = float64_to_double(b);
      double hr = ha * hb;
      if (unlikely(isinf(hr))) {
        st->float_exception_flags |= float_flag_overflow;
      } else if (unlikely(fabs(hr) <= DBL_MIN)) {
        goto soft_fp;
      }
      return double_to_float64(hr);
    }
  }
soft_fp:
  return soft_float64_mul(a, b, st);
}
```

.. and similarly for 32/64b + , - , × , ÷, √, ==

Common case:

- A, B are normal or zero
- Inexact already set
- Default rounding

How common?

99.18%

of FP instructions in SPECfp06

2. Other Optimizations

derived from state-of-the-art DBT engines

A. Indirect branch handling

- We implement Hong et al.'s [A] technique to speed up indirect branches
  - We add a new TCG operation so that all ISA targets can benefit

[A] Hong, Hsu, Chou, Hsu, Liu, Wu. "Optimizing Control Transfer and Memory Virtualization in Full System Emulators", ACM TACO, 2015

[B] Tong, Koju, Kawahito, Moshovos. "Optimizing memory translation emulation in full system emulators", ACM TACO, 2015

2. Other Optimizations

derived from state-of-the-art DBT engines

A. Indirect branch handling

- We implement Hong et al.'s [A] technique to speed up indirect branches
  - We add a new TCG operation so that all ISA targets can benefit

B. TLB Emulation (full-system)

- Virtual memory is emulated with a *software TLB*
  - Guest memory accesses first check a TLB array on the host

[A] Hong, Hsu, Chou, Hsu, Liu, Wu. "Optimizing Control Transfer and Memory Virtualization in Full System Emulators", ACM TACO, 2015

[B] Tong, Koju, Kawahito, Moshovos. "Optimizing memory translation emulation in full system emulators", ACM TACO, 2015

2. Other Optimizations

derived from state-of-the-art DBT engines

A. Indirect branch handling

- We implement Hong et al.'s [A] technique to speed up indirect branches
  - We add a new TCG operation so that all ISA targets can benefit

B. TLB Emulation (full-system)

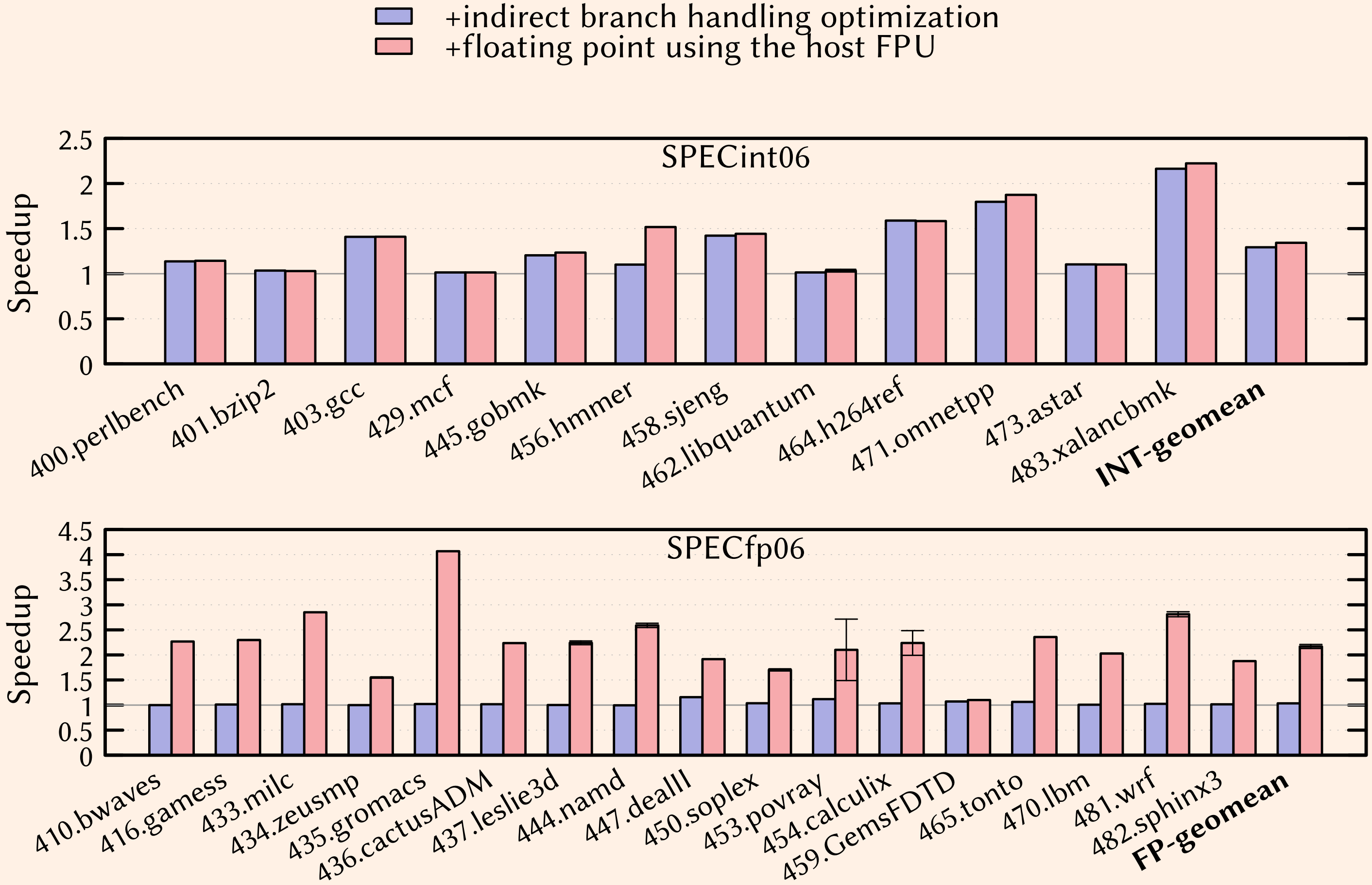
- Virtual memory is emulated with a *software TLB*
  - Guest memory accesses first check a TLB array on the host
- Tong et al. [B] present TLB resizing based on TLB use rate at flush time
  - We improve on it by incorporating **history to shrink less aggressively**
    - Rationale: if a memory-hungry process was just scheduled out, it is likely that it will be scheduled in in the near future

[A] Hong, Hsu, Chou, Hsu, Liu, Wu. "Optimizing Control Transfer and Memory Virtualization in Full System Emulators", ACM TACO, 2015

[B] Tong, Koju, Kawahito, Moshovos. "Optimizing memory translation emulation in full system emulators", ACM TACO, 2015

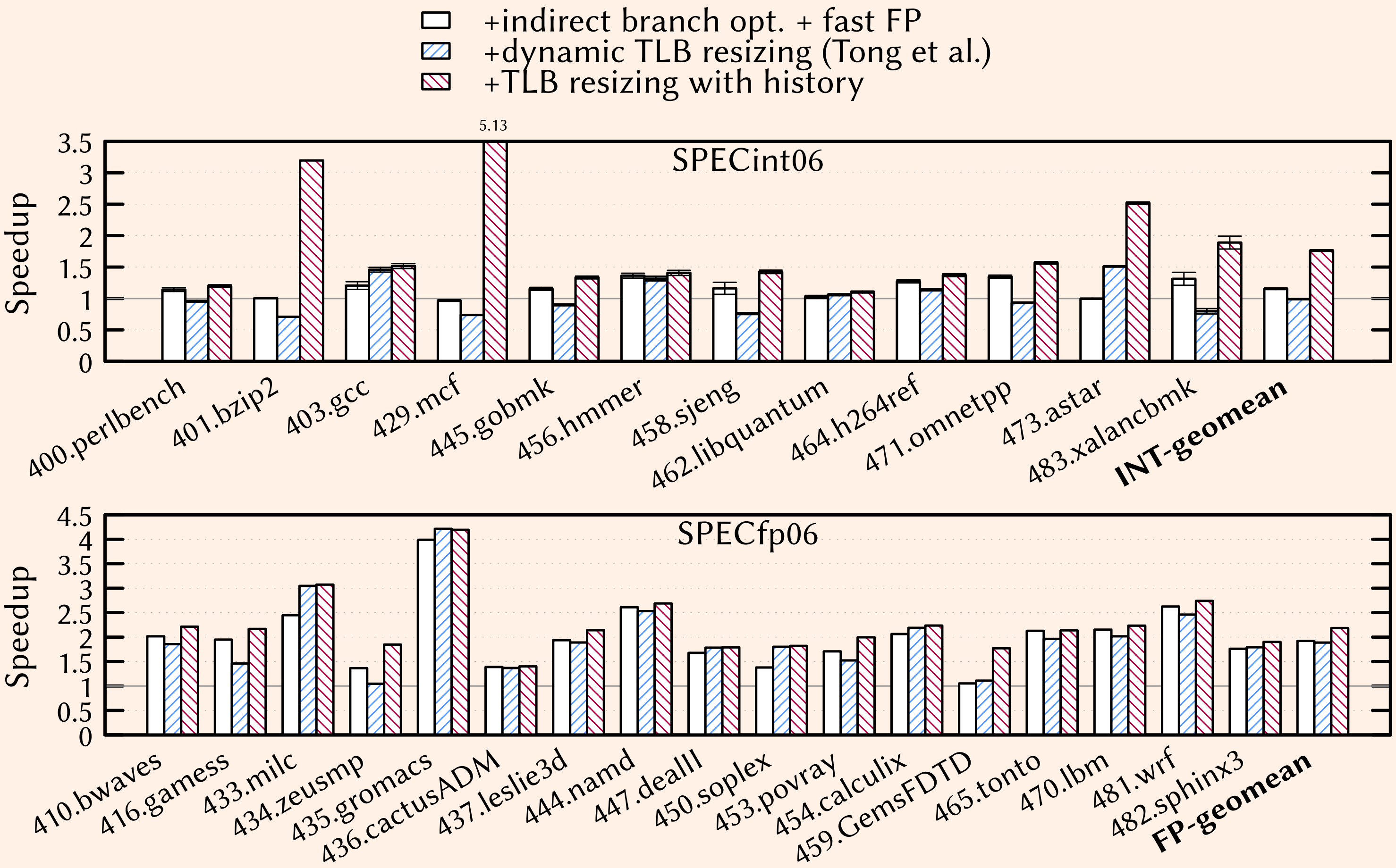
Ind. branch + FP improv.

user-mode x86_64-on-x86_64. Baseline: Pico (i.e. QEMU v3.1.0)



TLB resizing

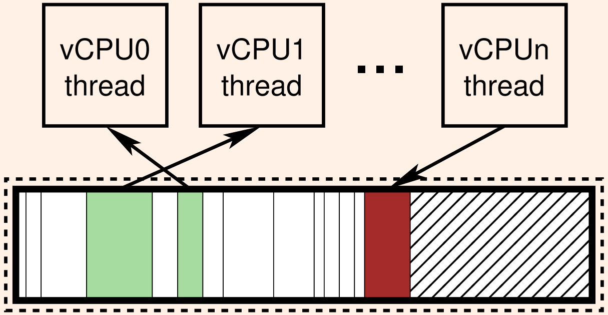
full-system x86_64-on-x86_64. Baseline: Pico (i.e. QEMU v3.1.0)



- +TLB **history**: takes into account recent usage of the TLB to shrink less aggressively, improving performance

3. Parallel code translation

with a shared translation block (TB) cache

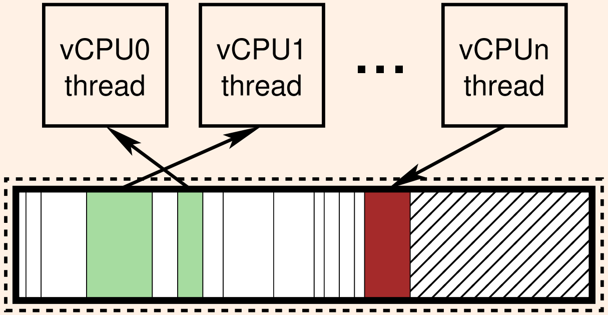


Monolithic TB cache (Pico)

- ✓ Parallel TB execution (*green* blocks)
- ✗ Serialized TB generation (*red* blocks) with a **global lock**

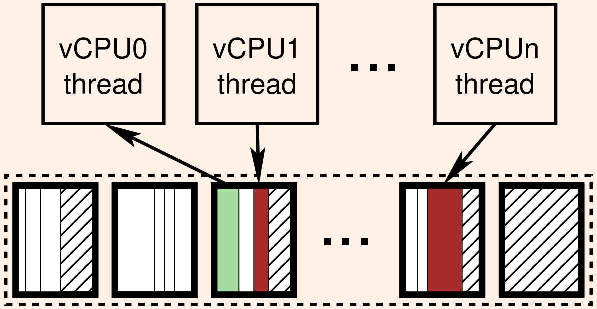
3. Parallel code translation

with a shared translation block (TB) cache



Monolithic TB cache (Pico)

- ✓ Parallel TB execution (*green* blocks)
- ✗ Serialized TB generation (*red* blocks) with a **global lock**



Partitioned TB cache (Qelt)

- ✓ Parallel TB execution
- ✓ Parallel TB generation (one region per vCPU)

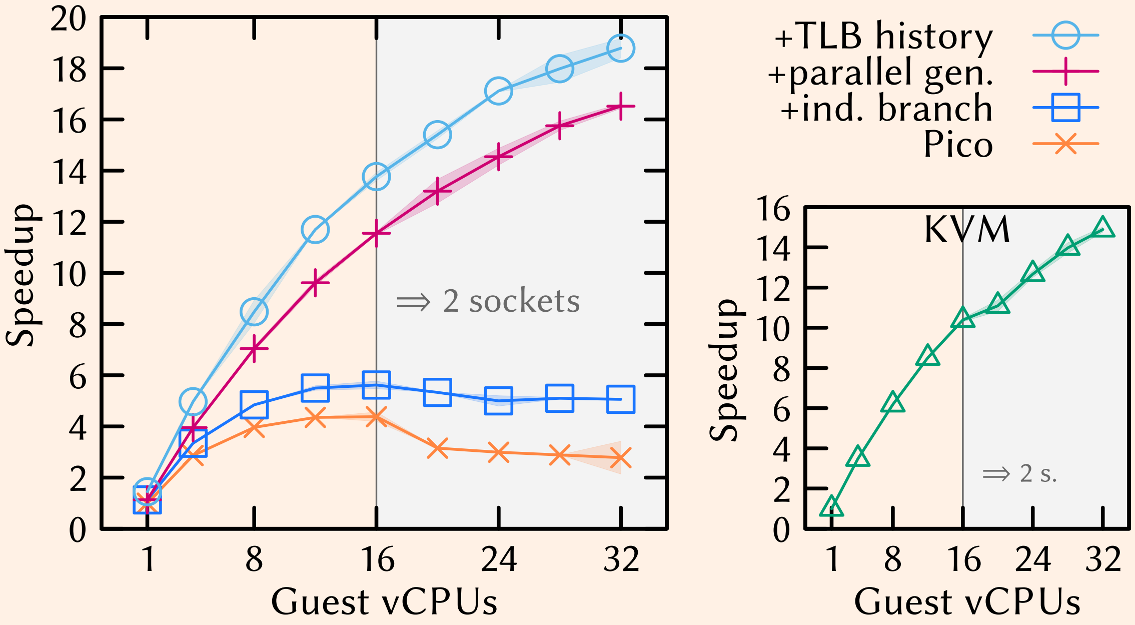
- vCPUs generate code at different rates
  - Appropriate region sizing ensures low code cache waste

Parallel code translation

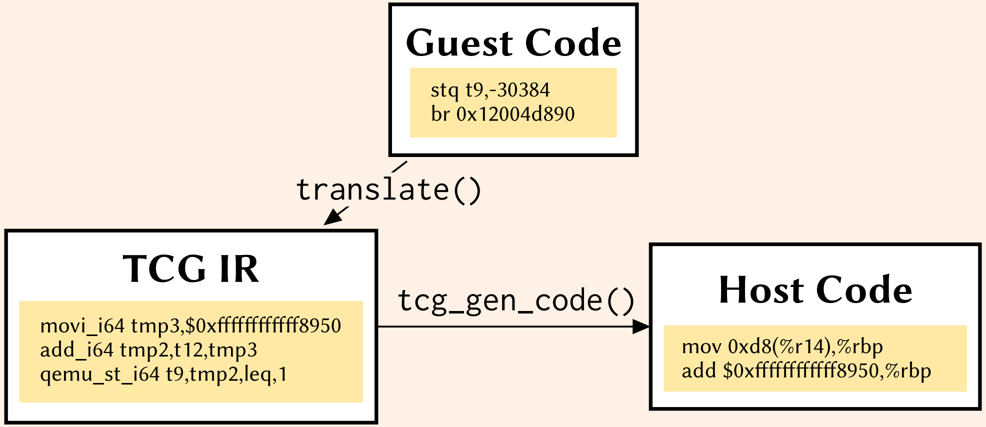
Guest VM performing parallel compilation of Linux kernel modules, x86_64-on-x86_64

- Pico does not scale for this workload due to contention on the **lock serializing code generation**

- +parallel generation **removes the scalability bottleneck**
    - Scalability is similar (or better) to KVM's



4. Cross-ISA Instrumentation



Guest Code

```
stq t9,-30384
br 0x12004d890
```

translate()

TCG IR

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
```

tcg_gen_code()

Host Code

```
mov 0xd8(%r14),%rbp
add $0xffffffffffff8950,%rbp
```

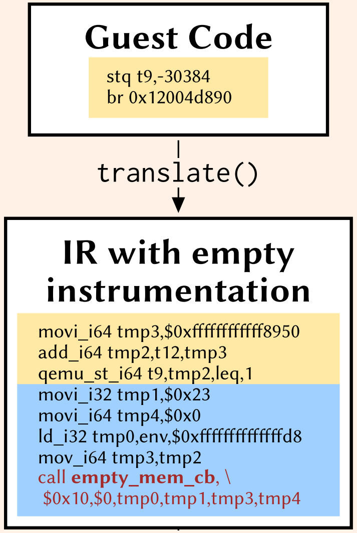
QEMU/Pico cannot instrument the guest

- Would like **plugin** code to receive ***callbacks*** on ***instruction-grained events***
  - e.g. memory accesses performed by a particular instruction in a translated block (TB), as in Pin

4. Cross-ISA Instrumentation

Instrumentation with Qelt

- Qelt first adds "empty" instrumentation in TCG, QEMU's IR



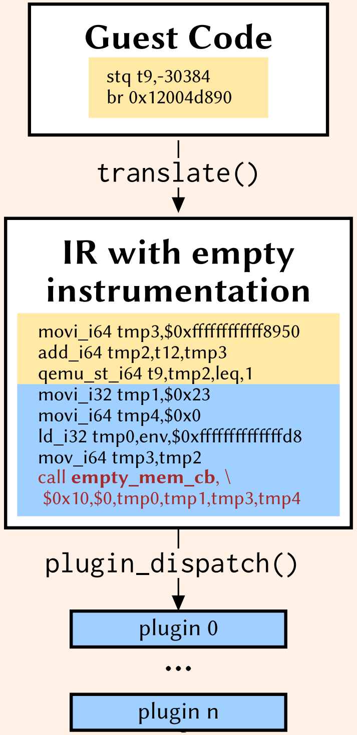
```
stq t9,-30384
br 0x12004d890
```

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
movi_i32 tmp1,$0x23
movi_i64 tmp4,$0x0
ld_i32 tmp0,env,$0xffffffffffffffd8
mov_i64 tmp3,tmp2
call empty_mem_cb, \
 $0x10,$0,tmp0,tmp1,tmp3,tmp4
```

4. Cross-ISA Instrumentation

Instrumentation with Qelt

- Qelt first adds "empty" instrumentation in TCG, QEMU's IR
- Plugins subscribe to events in a TB
    - They can use a decoder; Qelt only sees opaque insns/accesses



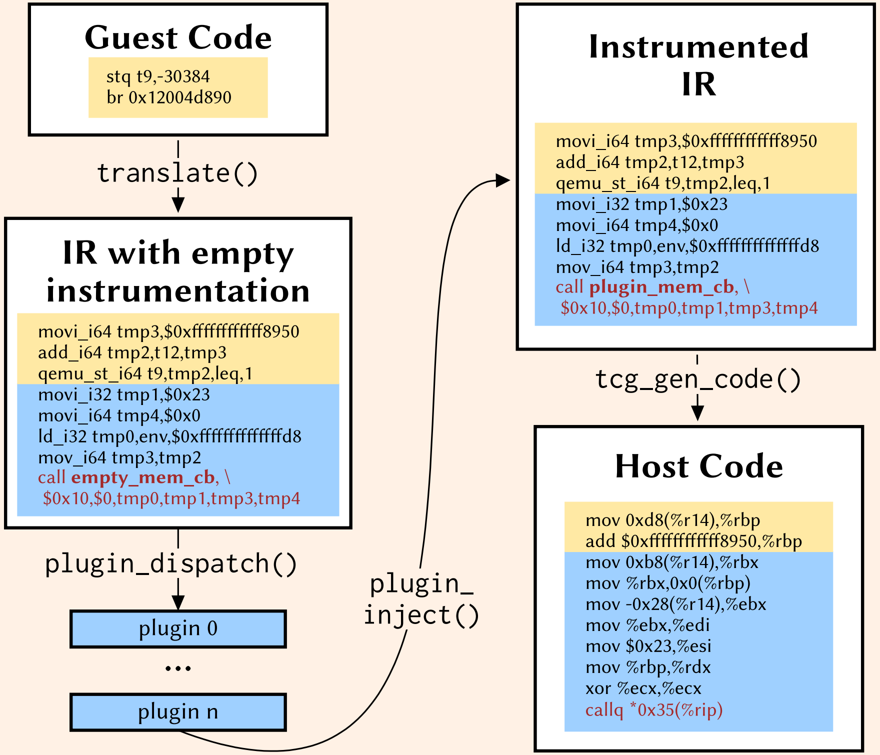
```
stq t9,-30384
br 0x12004d890
```

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
movi_i32 tmp1,$0x23
movi_i64 tmp4,$0x0
ld_i32 tmp0,env,$0xffffffffffffffd8
mov_i64 tmp3,tmp2
call empty_mem_cb, \
 $0x10,$0,tmp0,tmp1,tmp3,tmp4
```

4. Cross-ISA Instrumentation

Instrumentation with Qelt

- Qelt first adds "empty" instrumentation in TCG, QEMU's IR
- Plugins subscribe to events in a TB
  - They can use a decoder; Qelt only sees opaque insns/accesses
- Qelt then substitutes "empty" instrumentation with the actual calls to plugin *callbacks* (or removes it if not needed)



Guest Code

```
stq t9,-30384
br 0x12004d890
```

IR with empty instrumentation

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
movi_i32 tmp1,$0x23
movi_i64 tmp4,$0x0
ld_i32 tmp0,env,$0xffffffffffffffd8
mov_i64 tmp3,tmp2
call empty_mem_cb, \
 $0x10,$0,tmp0,tmp1,tmp3,tmp4
```

Instrumented IR

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
movi_i32 tmp1,$0x23
movi_i64 tmp4,$0x0
ld_i32 tmp0,env,$0xffffffffffffffd8
mov_i64 tmp3,tmp2
call plugin_mem_cb, \
 $0x10,$0,tmp0,tmp1,tmp3,tmp4
```

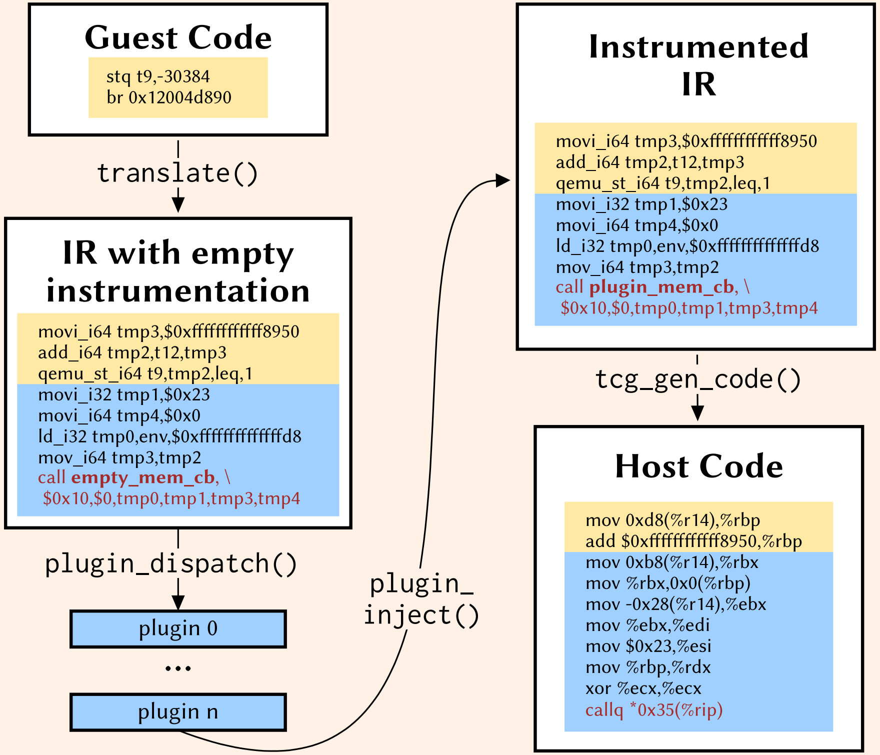
Host Code

```
mov 0xd8(%r14),%rbp
add $0xffffffffffff8950,%rbp
mov 0xb8(%r14),%rbx
mov %rbx,0x0(%rbp)
mov -0x28(%r14),%ebx
mov %ebx,%edi
mov $0x23,%esi
mov %rbp,%rdx
xor %ecx,%ecx
callq *0x35(%rip)
```

4. Cross-ISA Instrumentation

Instrumentation with Qelt

- Qelt first adds "empty" instrumentation in TCG, QEMU's IR
- Plugins subscribe to events in a TB
  - They can use a decoder; Qelt only sees opaque insns/accesses
- Qelt then substitutes "empty" instrumentation with the actual calls to plugin ***callbacks*** (or removes it if not needed)



Guest Code

```
stq t9,-30384
br 0x12004d890
```

IR with empty instrumentation

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
movi_i32 tmp1,$0x23
movi_i64 tmp4,$0x0
ld_i32 tmp0,env,$0xffffffffffffffd8
mov_i64 tmp3,tmp2
call empty_mem_cb, \
 $0x10,$0,tmp0,tmp1,tmp3,tmp4
```

Instrumented IR

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
movi_i32 tmp1,$0x23
movi_i64 tmp4,$0x0
ld_i32 tmp0,env,$0xffffffffffffffd8
mov_i64 tmp3,tmp2
call plugin_mem_cb, \
 $0x10,$0,tmp0,tmp1,tmp3,tmp4
```

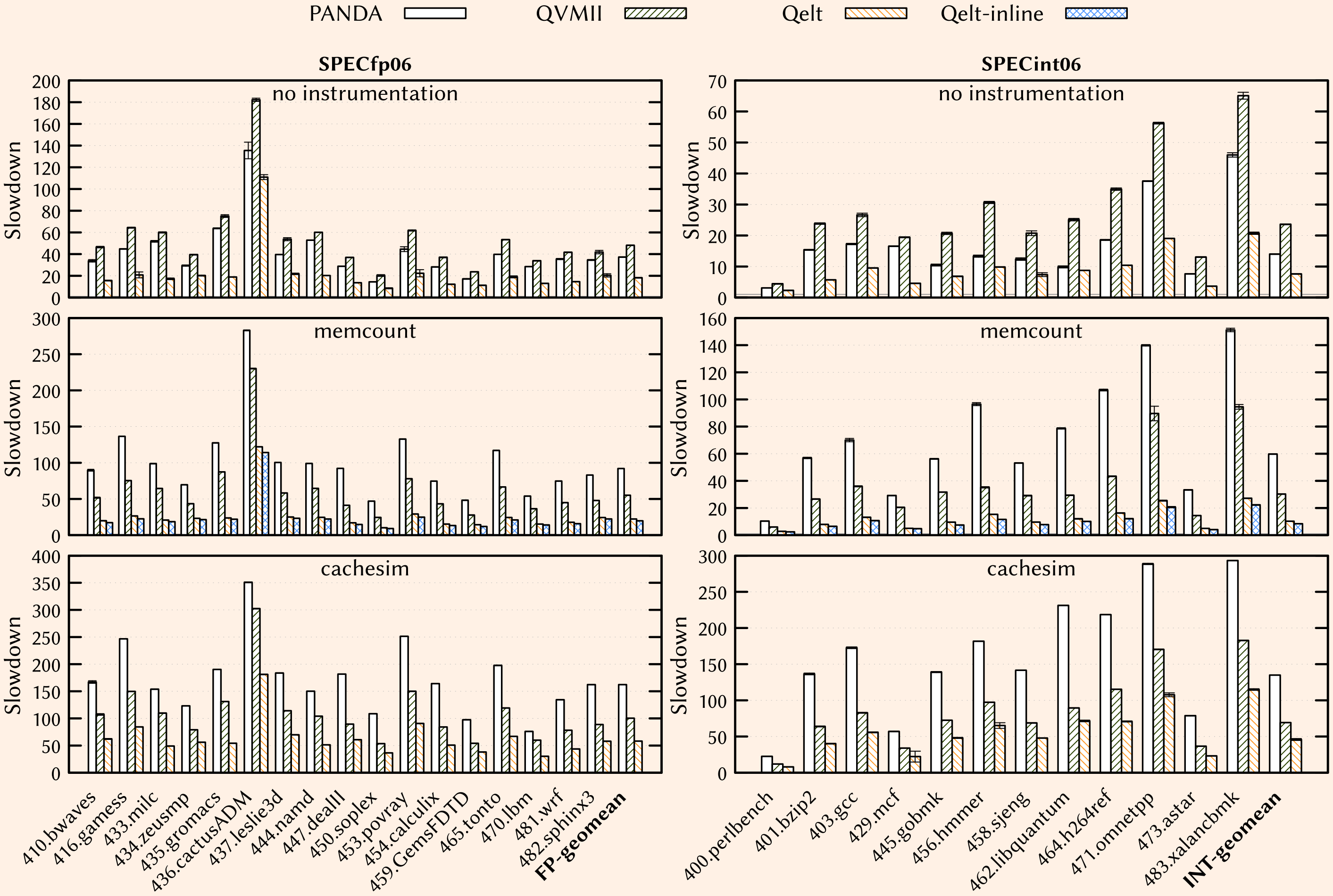
Host Code

```
mov 0xd8(%r14),%rbp
add $0xffffffffffff8950,%rbp
mov 0xb8(%r14),%rbx
mov %rbx,0x0(%rbp)
mov -0x28(%r14),%ebx
mov %ebx,%edi
mov $0x23,%esi
mov %rbp,%rdx
xor %ecx,%ecx
callq *0x35(%rip)
```

- Other features: inlining, helper instr., accelerator support (DMA, interrupts, I/O)...

Full-system instrumentation

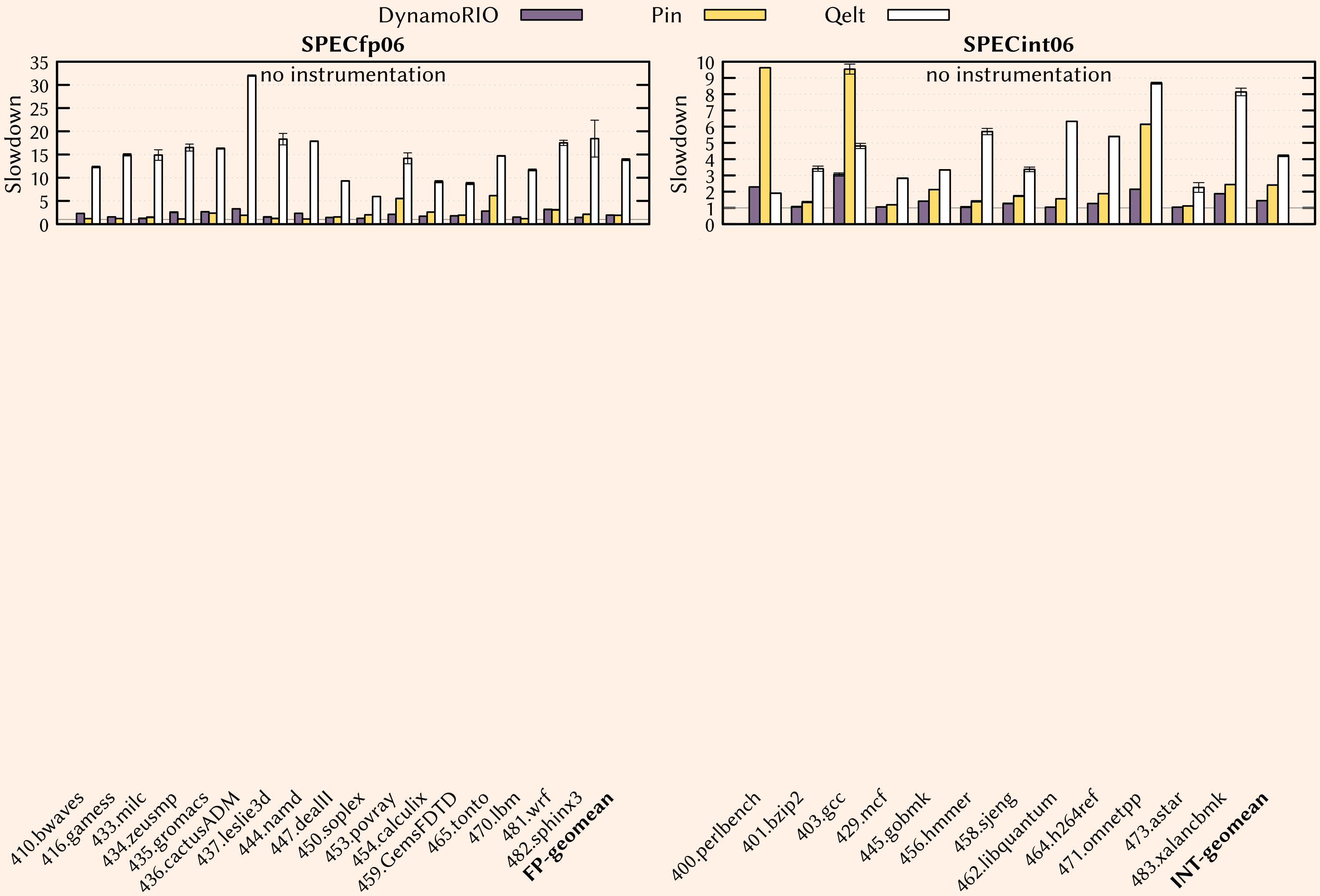
x86_64-on-x86_64 (lower is better). Baseline: KVM



Qelt faster than the state-of-the-art, even for heavy instrumentation (cachesim)

User-mode instrumentation

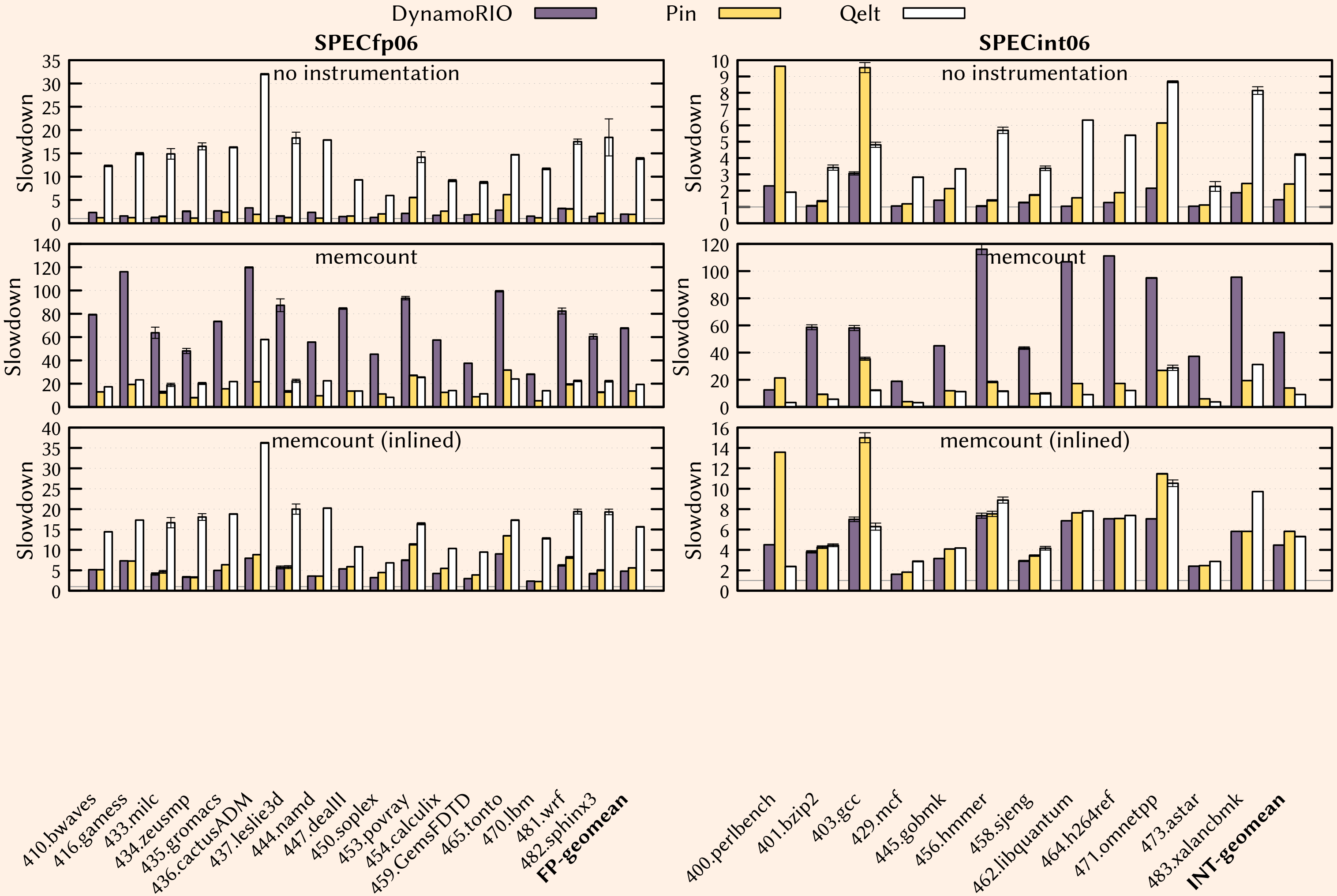
x86_64-on-x86_64 (lower is better). Baseline: native



- Qelt has narrowed the gap with Pin/DRIO for no instr., although for FP the gap is still significant

User-mode instrumentation

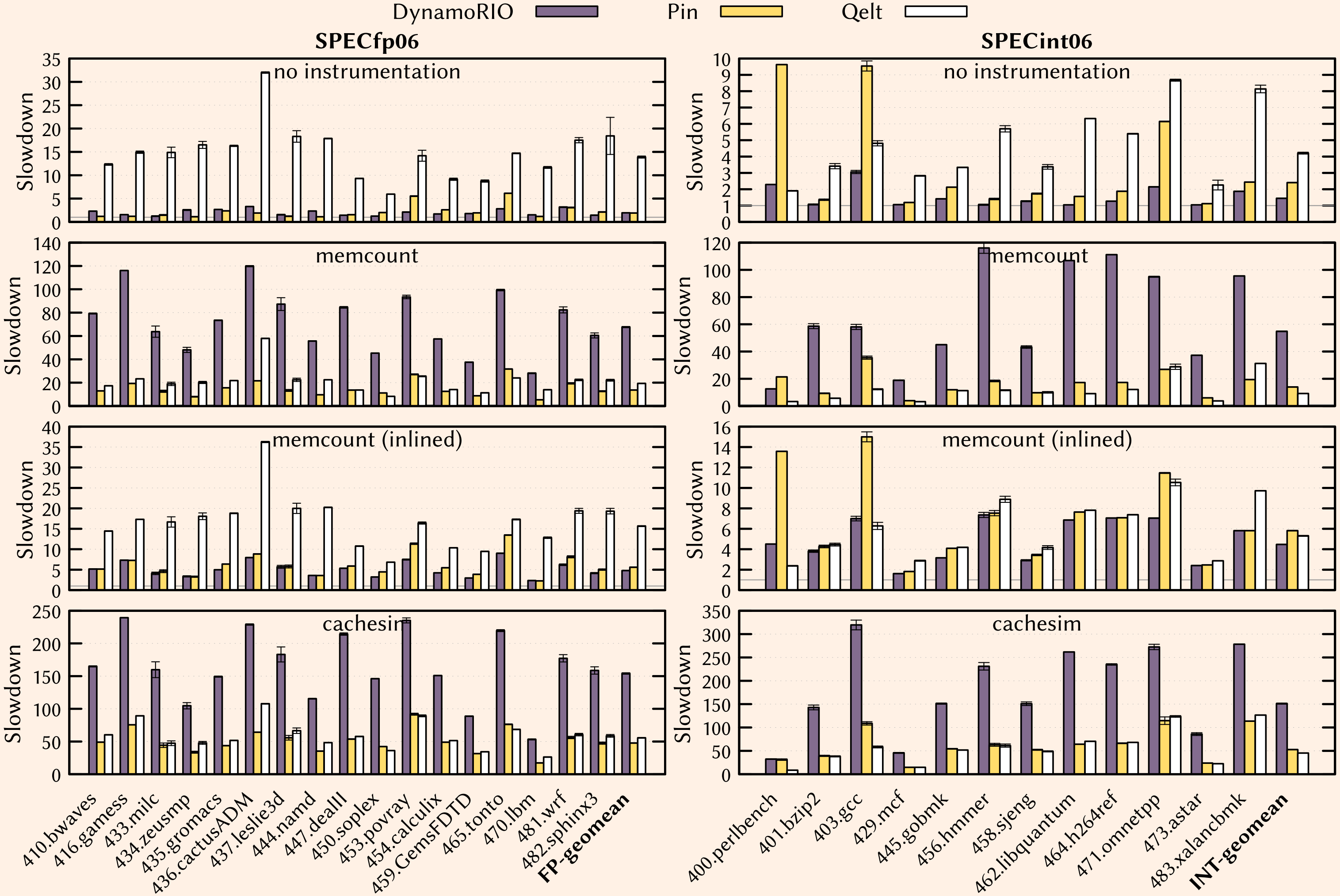
x86_64-on-x86_64 (lower is better). Baseline: native



- Qelt has narrowed the gap with Pin/DRIO for no instr., although for FP the gap is still significant
- DRIO is not designed for non-inline instr.

User-mode instrumentation

x86_64-on-x86_64 (lower is better). Baseline: native



- Qelt has narrowed the gap with Pin/DRIO for no instr., although for FP the gap is still significant

- DRIO is not designed for non-inline instr.

- **Qelt is competitive with Pin for heavy instrumentation (cachesim), while being cross-ISA**

Thesis

"Fast, scalable machine emulation is feasible and useful for evaluating the design and integration of heterogeneous systems"

Contributions

Cross-ISA Emulation

Pico **[CGO'17]**	• Design of a scalable, full-system, cross-ISA emulator • Handling of guest-host ISA differences in atomic instructions
Qelt **[VEE'19]**	• Fast, correct cross-ISA FP emulation leveraging the host FPU • Fast, cross-ISA instrumentation layer • Scalable emulation also during heavy code generation

Accelerator Integration

• Quantitative comparison of accelerator couplings	**[DAC'15]**
• Technique to lower the opportunity cost of accelerator integration by reusing acc. memories to extend the LLC	**ROCA** **[CAL'14, ICS'16]**

Thesis

"Fast, scalable machine emulation is feasible and useful for evaluating the design and integration of heterogeneous systems"

Contributions

Cross-ISA Emulation

Pico
[CGO'17]

- Design of a scalable, full-system, cross-ISA emulator
- Handling of guest-host ISA differences in atomic instructions

Qelt
[VEE'19]

- Fast, correct cross-ISA FP emulation leveraging the host FPU
- Fast, cross-ISA instrumentation layer
- Scalable emulation also during heavy code generation

Accelerator Integration

- Quantitative comparison of accelerator couplings [DAC'15]
- Technique to lower the opportunity cost of accelerator integration by reusing acc. memories to extend the LLC ROCA [CAL'14, ICS'16]

Accelerator Coupling in Heterogeneous Architectures

goal: to draw observations about performance, efficiency and programmability of accelerators with different couplings

Cota, Mantovani, Di Guglielmo, Carloni. "An Analysis of Accelerator Coupling in Heterogeneous Architectures", DAC, 2015

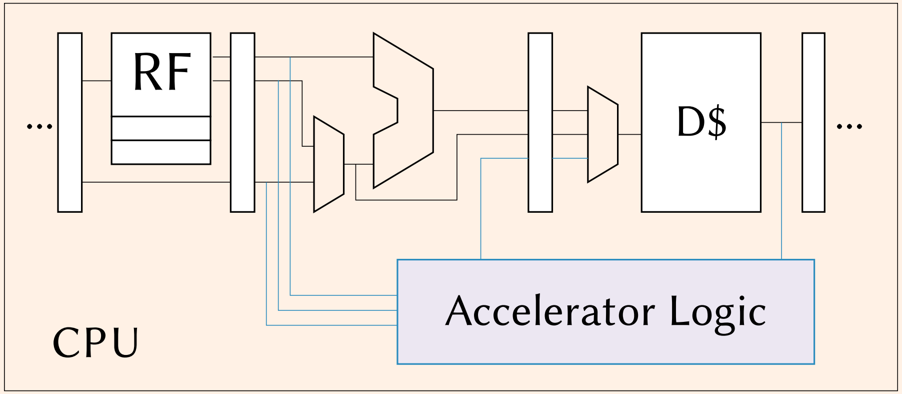
Tightly-Coupled Accelerators

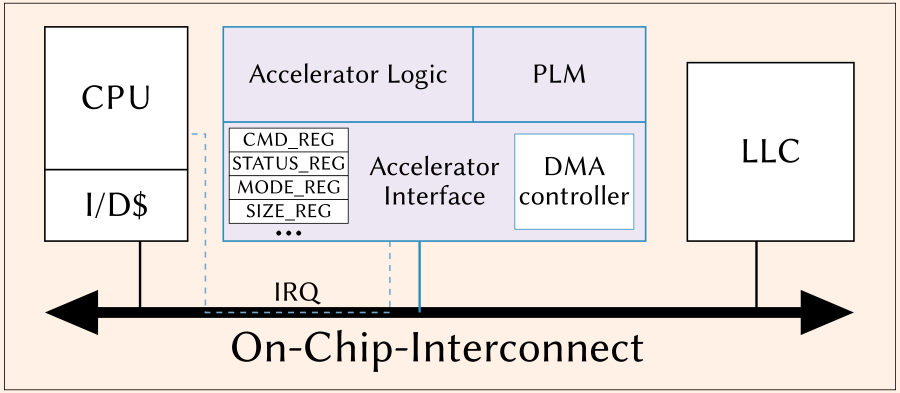
TCA

VS.

Loosely-Coupled Accelerators

LCA, two flavors: DRAM-DMA, LLC-DMA





✔Nil invocation overhead (via ISA extensions)

✔ No internal storage: direct access to L1 cache

✗ Limited portability: design heavily tied to CPU

✔Good design reuse: no CPU-specific knowledge

✗ High set-up costs: driver invocation and DMA

✔ Freedom to tailor private memories (PLMs), e.g. providing different banks, ports, and bit widths

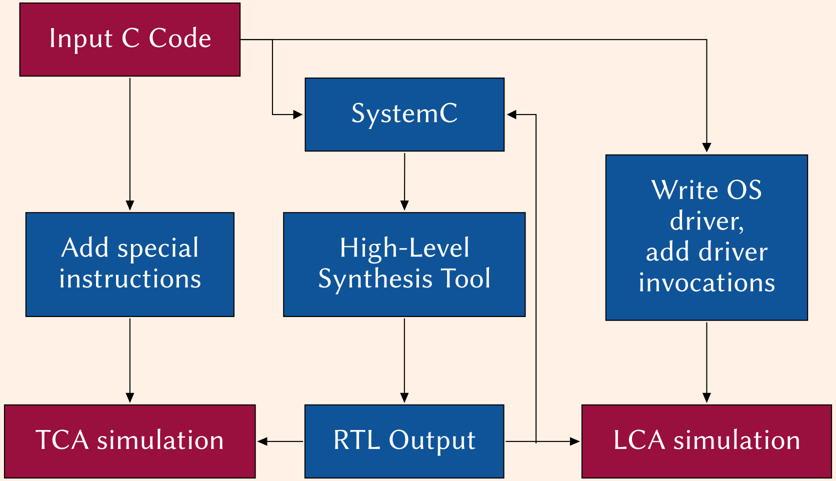
✗ PLMs require large area expenses

- Applications: Seven high-throughput kernels from the PERFECT Benchmark Suite[*]
  - Used High-Level Synthesis for productivity

[*] http://hpc.pnl.gov/PERFECT/

Cargo: Heterogeneous System Simulation

Accelerators



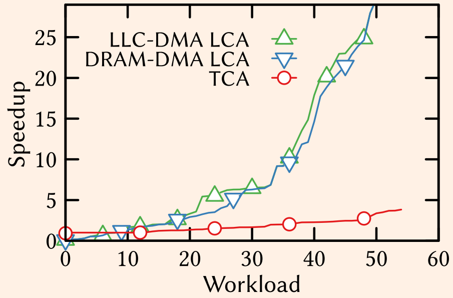
CPU & memory

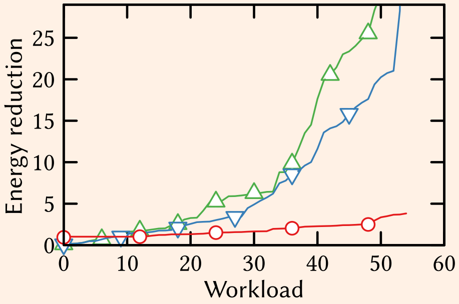
Cores	2 cores, i386 ISA, 3-stage pipeline, 2 GHz
Execute Latency	1 cycle except IMUL=4, IDIV=15, FPADD=5, FPMUL=5, FPDIV=25 [5]
L1 caches	32KB I, 64KB D, 4 ways, 2+2 I/O ports, 1-cycle latency, LRU replacement
L2 cache	4MB, 16 ways, 16 banks, 4 MSHRs, 1+1 I/O ports, 11-cycle latency, LRU
DRAM	1 Controller, 3.5GB, Micron DDR3 400MHz
Operating System	Linux v2.6.34

- Full-system running Linux
- Detailed event-driven L1 and L2 cache models + DRAMSim2 for DRAM
- LCAs: Unmodified SystemC/RTL/Chisel/C/C++ accelerators are simulated in parallel with the CPU simulation, synchronizing every n cycles (e.g. 100)

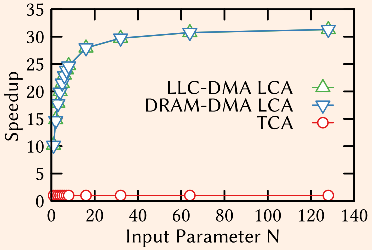
Results

Perf. & Efficiency



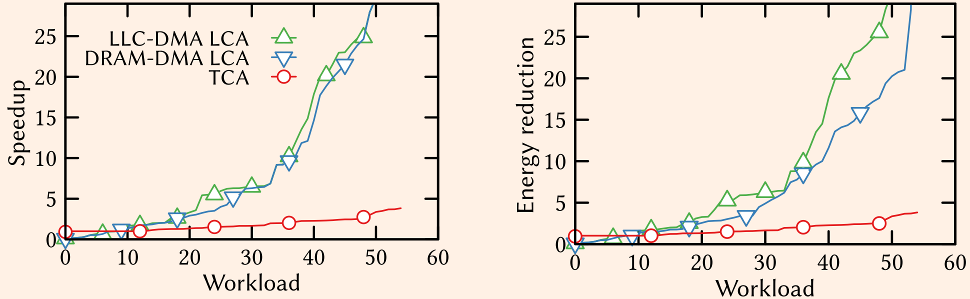


- **LCAs** best positioned to deliver high throughput given inputs of non-trivial size
  - Efficiency gap between LCAs due to difference in off-chip accesses
  - LCAs can saturate DRAM bandwidth, e.g. sort:

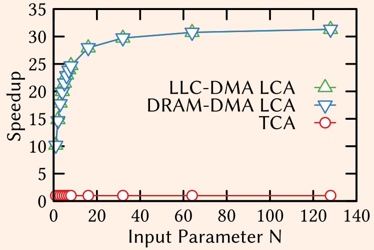


Results

Perf. & Efficiency

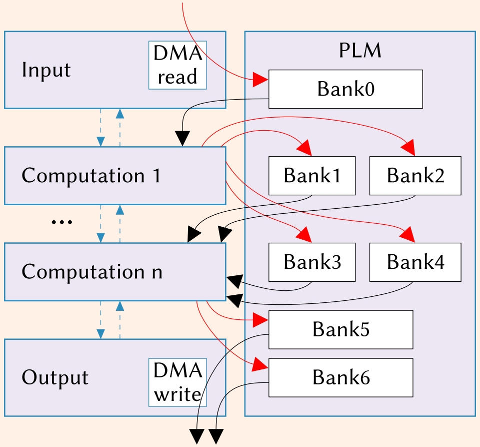


- **LCAs** best positioned to deliver high throughput given inputs of non-trivial size
  - Efficiency gap between LCAs due to difference in off-chip accesses
  - LCAs can saturate DRAM bandwidth, e.g. sort:



Why **LCAs > TCAs:**
Tailored, many-ported PLMs are key to performance

- L1s cannot provide this parallelism (at most 2 ports!)



ROCA: Reducing the Opportunity Cost of Accelerator Integration

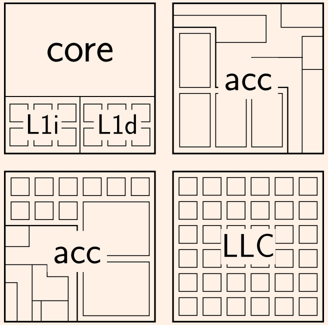
goal: to expose on-chip accelerator PLMs to the LLC, thereby extracting utility from accelerators when otherwise unused

Cota, Mantovani, Petracca, Casu, Carloni. "Accelerator Memory Reuse in the Dark Silicon Era", Computer Architecture Letters (CAL), 2014

Cota, Mantovani, Carloni. "Exploiting Private Local Memories to Reduce the Opportunity Cost of Accelerator Integration", Intl. Conf. on Supercomputing (ICS), 2016

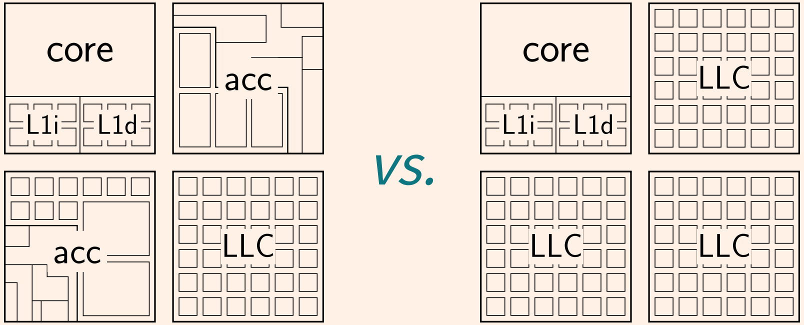
Accelerators' Opportunity Cost

An accelerator is only of ***utility*** if it applies to the system's workload



Accelerators' Opportunity Cost

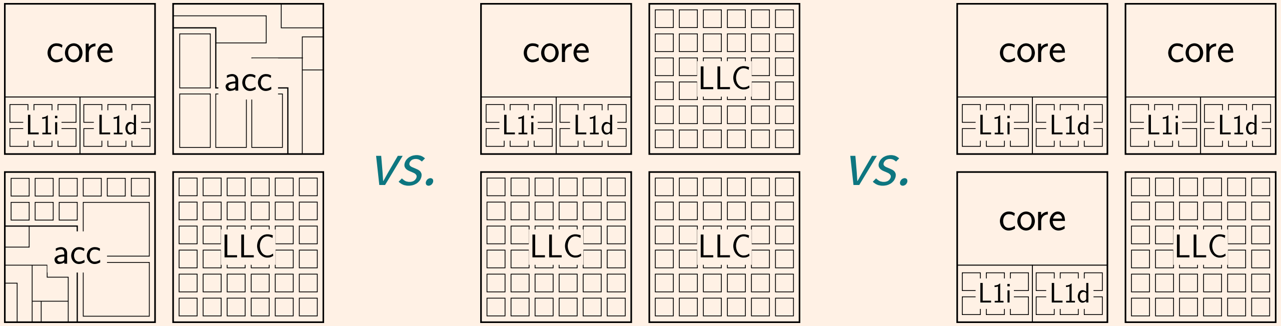
An accelerator is only of ***utility*** if it applies to the system's workload



If it doesn't, more generally-applicable alternatives are more productive

Accelerators' Opportunity Cost

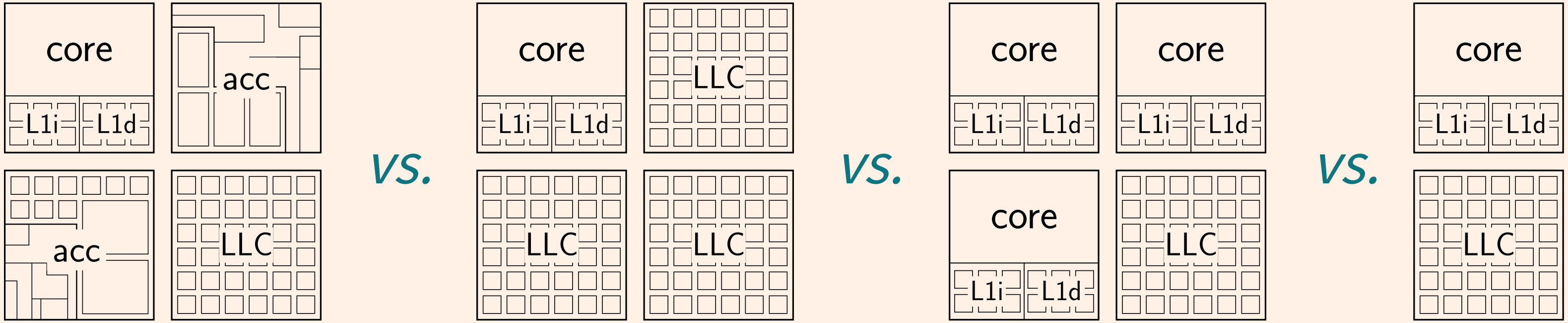
An accelerator is only of ***utility*** if it applies to the system's workload



If it doesn't, more generally-applicable alternatives are more productive

Accelerators' Opportunity Cost

An accelerator is only of ***utility*** if it applies to the system's workload



If it doesn't, more generally-applicable alternatives are more productive

Observation #1: Accelerators are mostly memory

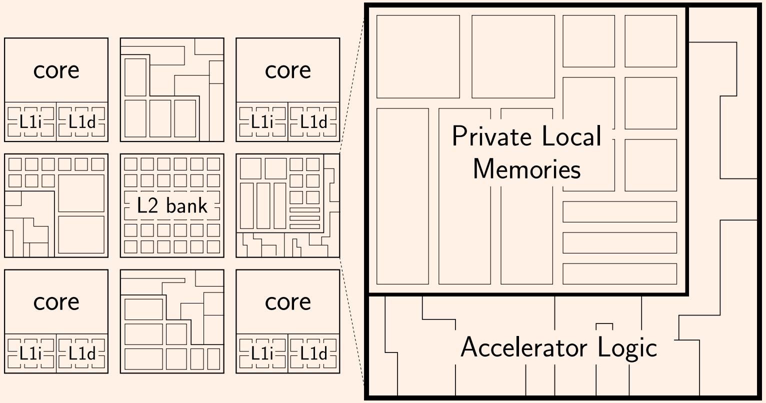
> *“An average of 69% of accelerator area is consumed by memory*
>
> *Lyons, Hempstead, Wei, Brooks. "The Accelerator Store", TACO, 2012*

Accelerator examples: AES, JPEG encoder, FFT, USB, CAN, TFT controller, UMTS decoder..

#2: Average accelerator memory utilization is low

Not all accelerators on a chip are likely to run at the same time

#3: Acc. PLMs provide a de facto NUCA substrate



Observation #1: Accelerators are mostly memory

> *“An average of 69% of accelerator area is consumed by memory*
>
> *Lyons, Hempstead, Wei, Brooks. "The Accelerator Store", TACO, 2012*

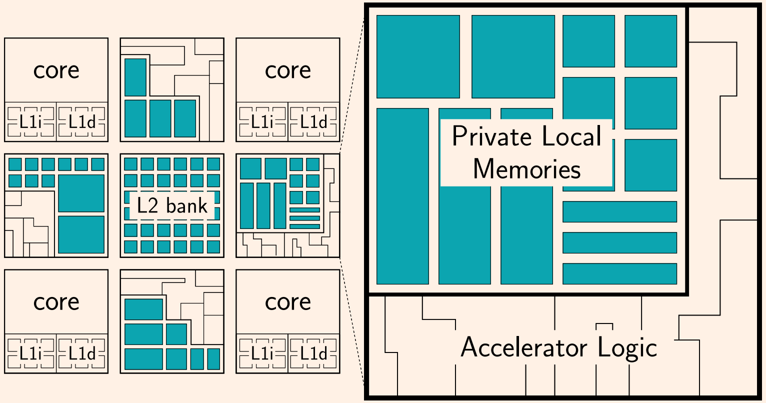
Accelerator examples: AES, JPEG encoder, FFT, USB, CAN, TFT controller, UMTS decoder..

#2: Average accelerator memory utilization is low

Not all accelerators on a chip are likely to run at the same time

#3: Acc. PLMs provide a de facto NUCA substrate

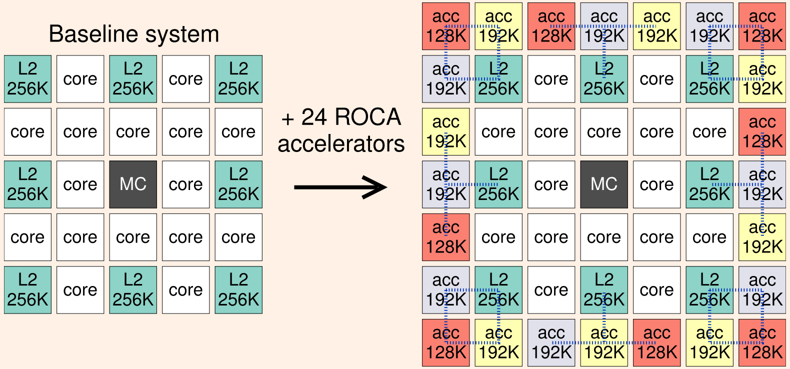
ROCA's Goal: **To extend the LLC with acc. PLMs when otherwise not in use**



✓ **No changes** to coherence protocol

✓ **Minimal modifications** to accelerators

Parallel Simulation with Cargo



Baseline system

L2 256K	core	L2 256K	core	L2 256K
core	core	core	core	core
L2 256K	core	MC	core	L2 256K
core	core	core	core	core
L2 256K	core	L2 256K	core	L2 256K

+ 24 ROCA accelerators

acc 128K	acc 192K	acc 128K	acc 192K	acc 192K	acc 192K	acc 128K
acc 192K	L2 256K	core	L2 256K	core	L2 256K	acc 192K
acc 192K	core	core	core	core	core	acc 128K
acc 192K	L2 256K	core	MC	core	L2 256K	acc 192K
acc 128K	core	core	core	core	core	acc 192K
acc 192K	L2 256K	core	L2 256K	core	L2 256K	acc 192K
acc 128K	acc 192K	acc 192K	acc 192K	acc 128K	acc 192K	acc 128K

Configurations:

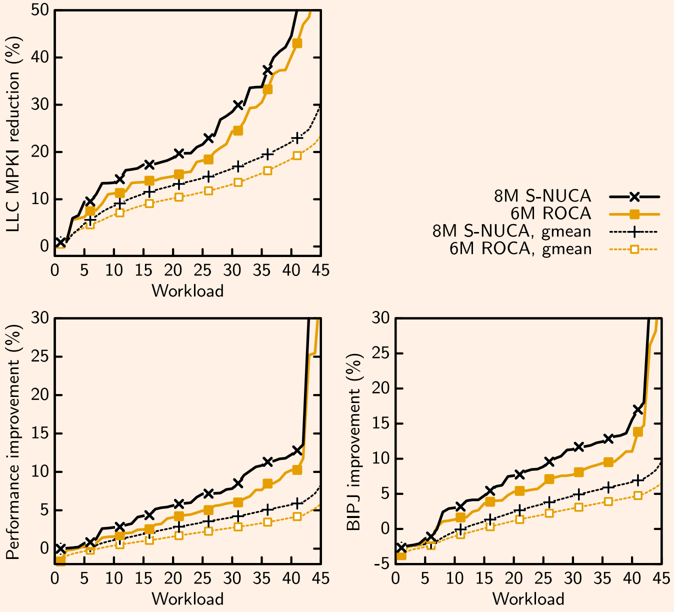
- 2M S-NUCA **baseline**
- **8MB S-NUCA** (not pictured)
- **same-area 6M ROCA**, assuming accelerators are 66% memory (below the typical 69%)

Workloads:

- Multi-programmed SPEC06 runs, not amenable to acceleration

cores	16 cores, i386 ISA, in-order IPC=1 except on memory accesses, 1GHz
L1 caches	Split I/D 32KB, 4-way set-associative, 1-cycle latency, LRU
L2 caches	8-cycle latency, LRU S-NUCA: 16ways, 8 banks ROCA: 12 ways
Coherence	MESI protocol, 64-byte blocks, standalone directory cache
DRAM	1 controller, 200-cycle latency, 3.5GB physical
NoC	5x5 or 7x7 mesh, 128b flits, 2-cycle router traversal, 1-cycle links, XY router
OS	Linux v2.6.34

Results



Assuming no accelerator activity,

- **6M ROCA** can realize **70%/68%** of the performance/energy efficiency benefits of a **same-area 8M S-NUCA**
  - while retaining accelerators' potential orders-of-magnitude gains

- **Sensitivity studies** sweeping accelerator activity over
  - **space** (which accelerators are reclaimed)
  - **time** (how frequently they are reclaimed)
- **Key result:** Accelerators with **idle windows >10ms** are prime candidates for ROCA
  - perf/eff. within 10/20% of that with 0% activity

Conclusions

Contributions

Cross-ISA Emulation

- **[CGO'17, VEE'19] Fast, scalable, cross-ISA machine emulation and instrumentation**
  - Performance for simulator-like instrumentation is competitive with state-of-the-art same-ISA emulators such as Pin

Accelerator Integration

- **[DAC'15]** Quantitative comparison of **accelerator couplings**
- **[CAL'14, ICS'16] ROCA**: Lower the **opportunity cost** of **accelerator** integration by reusing acc. memories to extend the LLC

[CAL'14] Cota, Mantovani, Petracca, Casu, Carloni. "Accelerator Memory Reuse in the Dark Silicon Era", Computer Architecture Letters (CAL), 2014
[DAC'15] Cota, Mantovani, Di Guglielmo, Carloni. "An Analysis of Accelerator Coupling in Heterogeneous Architectures", DAC, 2015
[ICS'16] Cota, Mantovani, Carloni. "Exploiting Private Local Memories to Reduce the Opportunity Cost of Accelerator Integration", Intl. Conf. on Supercomputing (ICS), 2016
[CGO'17] Cota, Bonzini, Bennée, Carloni. "Cross-ISA Machine Emulation for Multicores", CGO, 2017
[VEE'19] Cota, Carloni. "Cross-ISA Machine Instrumentation using Fast and Scalable Dynamic Binary Translation", VEE, 2019

Pico + Qelt in QEMU

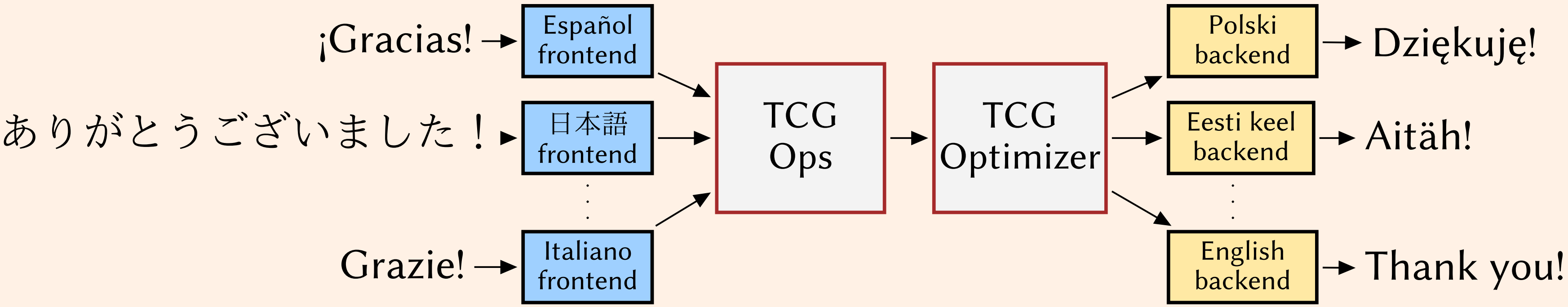
- Instrumentation layer: under review by the QEMU community
- Everything else: **merged upstream** QEMU v2.7 (Sept'16)↔QEMU v4.0 (April'19)
  - Code contributions well-received (and improved!) by the QEMU community
  - 302 commits to date

Pico + Qelt in QEMU

- Instrumentation layer: under review by the QEMU community
- Everything else: **merged upstream** QEMU v2.7 (Sept'16)↔QEMU v4.0 (April'19)
  - Code contributions well-received (and improved!) by the QEMU community
  - 302 commits to date

Future Impact

- We hope other researchers and educators will adopt QEMU to drive **simulators** of **heterogeneous systems**
- Cargo is a good example
  - Orders of magnitude faster than existing tools such as gem5-aladdin (~200 KIPS vs. ~10s of MIPS)
  - Has been used for both **research** and **teaching** at Columbia


¡Gracias!
ありがとうございました！
Grazie!
Español frontend
日本語 frontend
Italiano frontend
TCG Ops
TCG Optimizer
Polski backend
Eesti keel backend
English backend
Dziękuję!
Aitäh!
Thank you!

Backup slides

Hash table Requirements

{ Fast, concurrent lookups

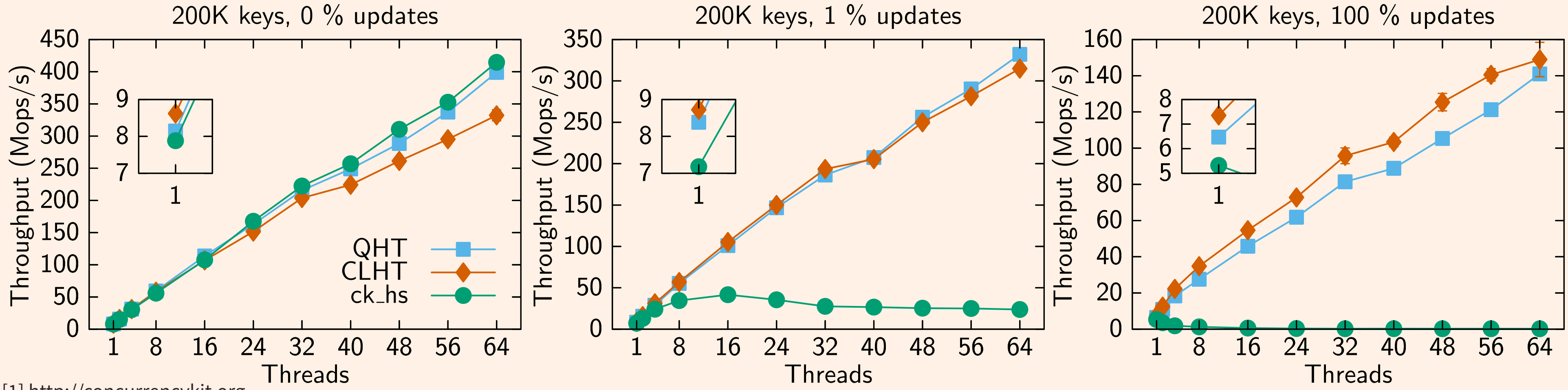
Low update rate: max 6% booting Linux

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

~~Candidate #2: **CLHT** [13]~~

#3: Our proposal: **QHT**

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



[1] http://concurrencykit.org

[12] D. Bruening, V. Kiriansky, T. Garnett, and S. Banerji. Thread-shared software code caches. CGO, pages 28–38, 2006

[13] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. ASPLOS, p. 631–644, 2015

QEMU emulation modes

User-mode (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

QEMU emulation modes

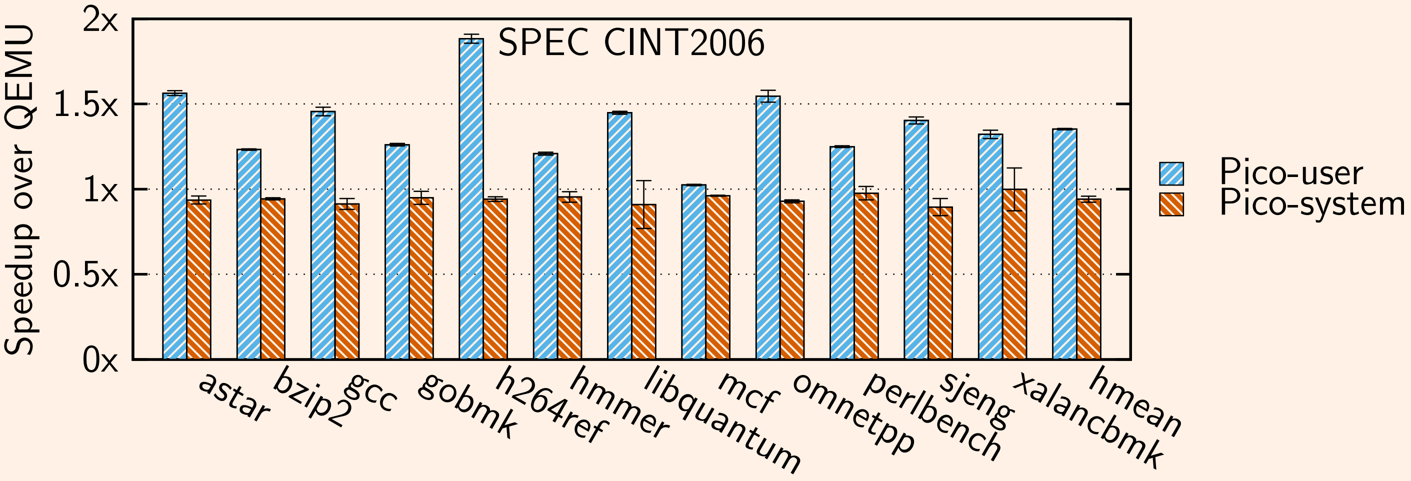
User-mode (QEMU-user)

- 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

System-mode (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 lockless TB lookups
- Pico-system's perf is virtually identical to QEMU-system's

Pico-ST: Store Tracking

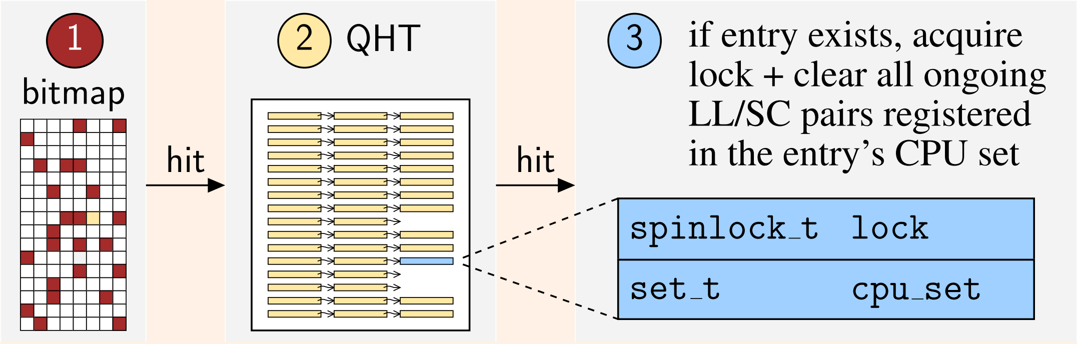
- Each address accessed atomically gets an entry of CPU set + lock
    - LL/SC emulation code operates on the CPU set atomically
- Keep entries in a HT indexed by address of atomic access

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!

Pico-ST: Store Tracking

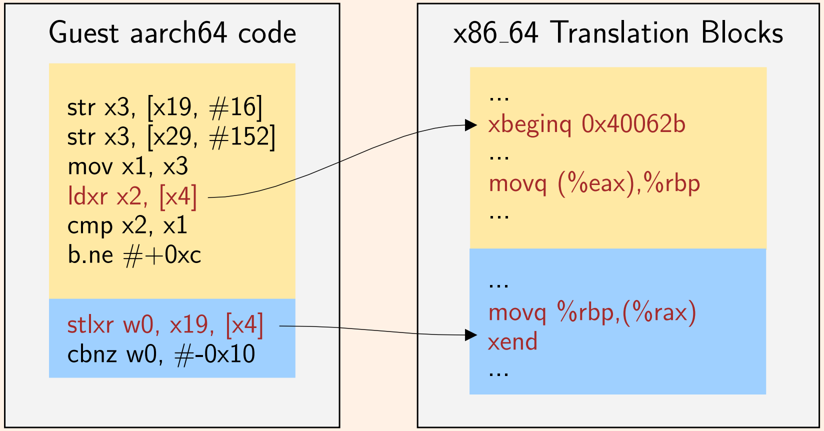
- Each address accessed atomically gets an entry of CPU set + lock
  - LL/SC emulation code operates on the CPU set atomically
- Keep entries in a HT indexed by address of atomic access
- **Problem:** regular stores must abort conflicting LL/SC pairs!
- Solution: instrument stores to check whether the 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



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

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* property*: "*A regular store forces all conflicting transactions to abort.*"



Guest aarch64 code

```
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
```

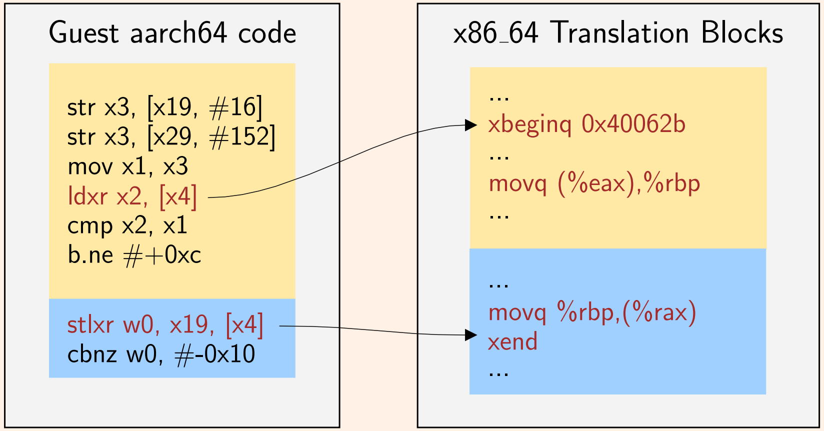
x86_64 Translation Blocks

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

[*] Blundell, Lewis, Martin. "Subtleties of transactional memory atomicity semantics", Computer Architecture Letters, 2006

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* property*: "*A regular store forces all conflicting transactions to abort.*"



Guest aarch64 code

```
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
```

x86_64 Translation Blocks

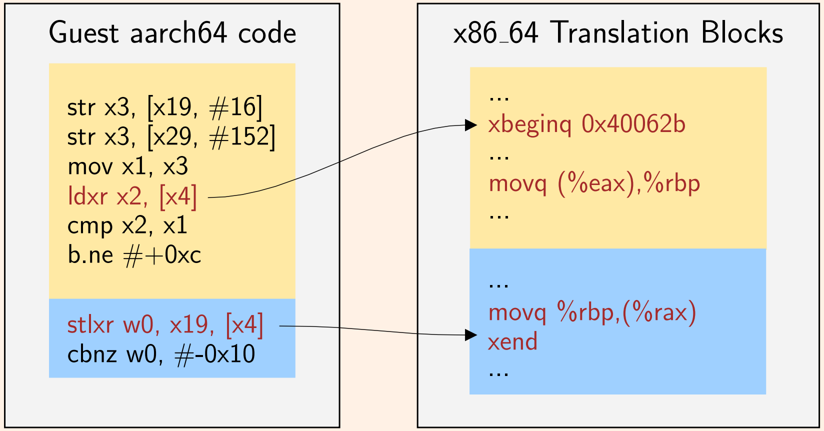
```
...
xbeginq 0x40062b
...
movq (%eax),%rbp
...

...
movq %rbp,(%rax)
xend
...
```

- Fallback: Emulate the LL/SC sequence with all other CPUs stopped

[*] Blundell, Lewis, Martin. "Subtleties of transactional memory atomicity semantics", Computer Architecture Letters, 2006

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* property*: "*A regular store forces all conflicting transactions to abort.*"



```
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
```

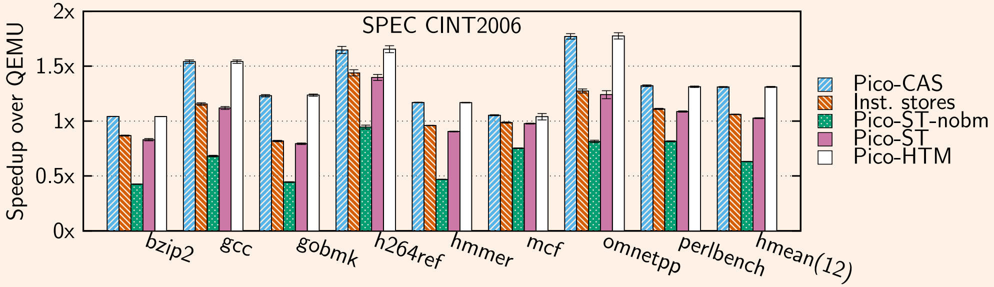
```
...
xbeginq 0x40062b
...
movq (%eax),%rbp
...

...
movq %rbp,(%rax)
xend
...
```

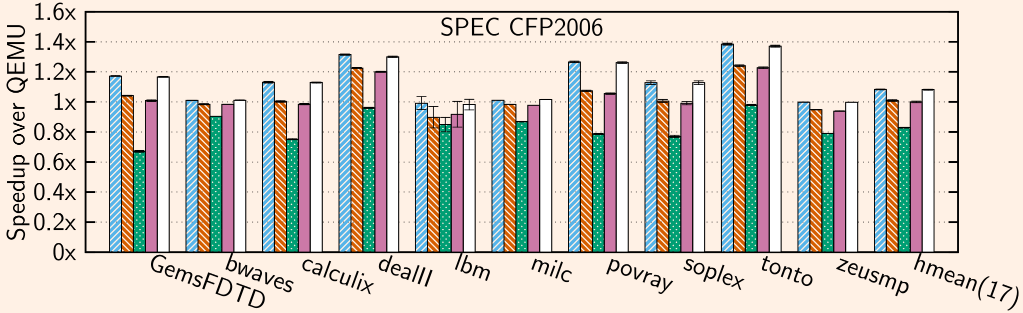
- Fallback: Emulate the LL/SC sequence with all other CPUs stopped
- Fun fact: no emulated SC ever reports failure!

[*] Blundell, Lewis, Martin. "Subtleties of transactional memory atomicity semantics", Computer Architecture Letters, 2006

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

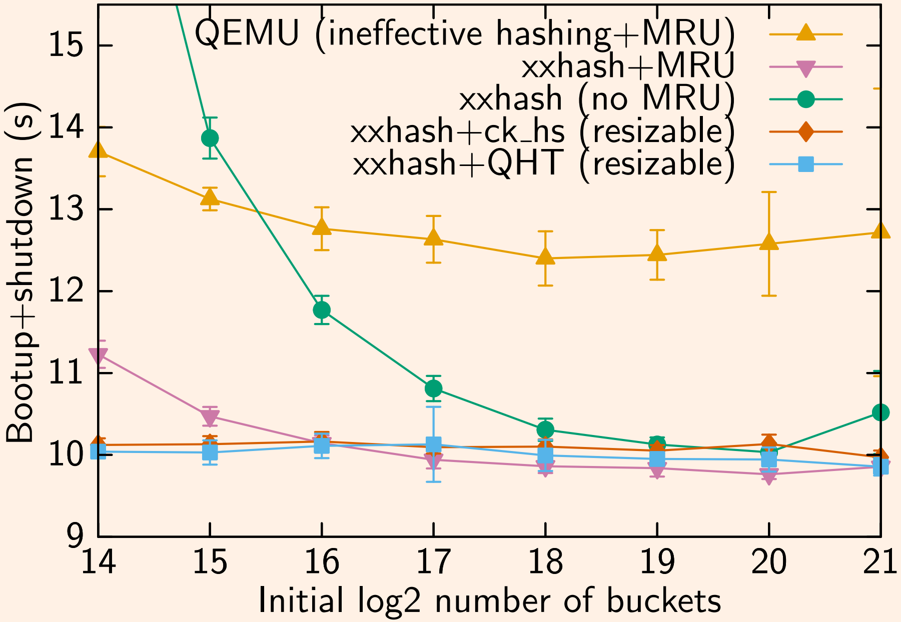
```
struct count {
    u64 val;
} __aligned(64); /* avoid false sharing */

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

Linux boot

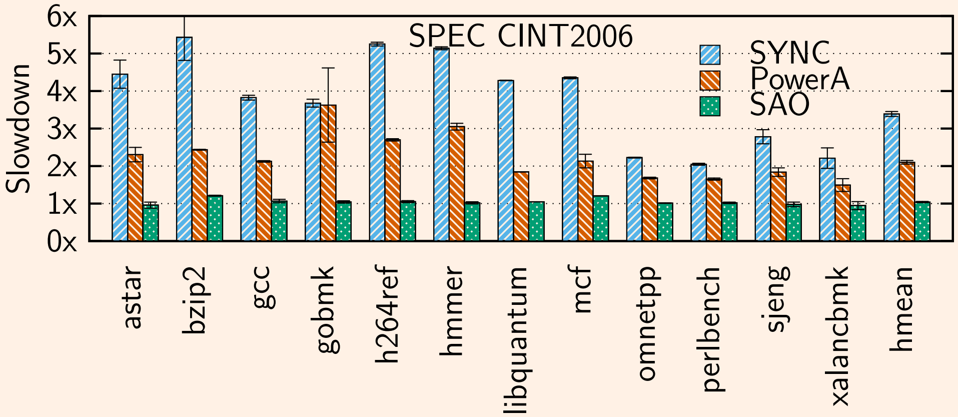
single thread

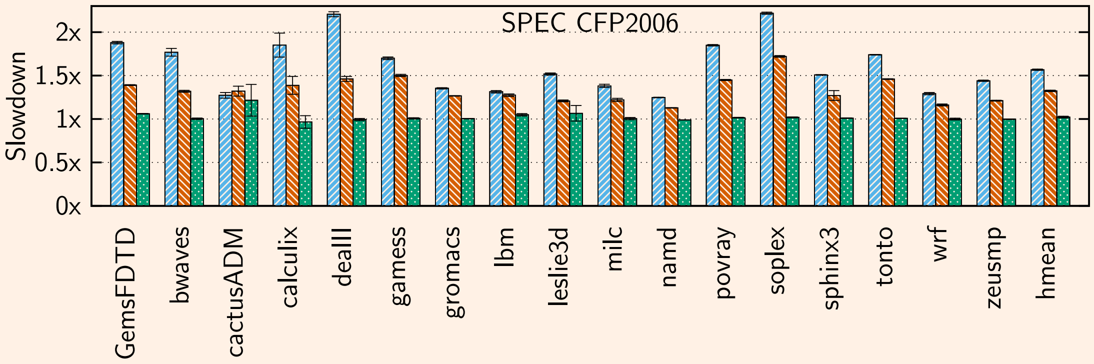


- **QHT** & **ck_hs** resize to always achieve the best perf
  - but **ck_hs** does not scale w/ ~6% update rates

Memory Consistency

x86-on-POWER





We applied ArMOR's [24] FSMs:

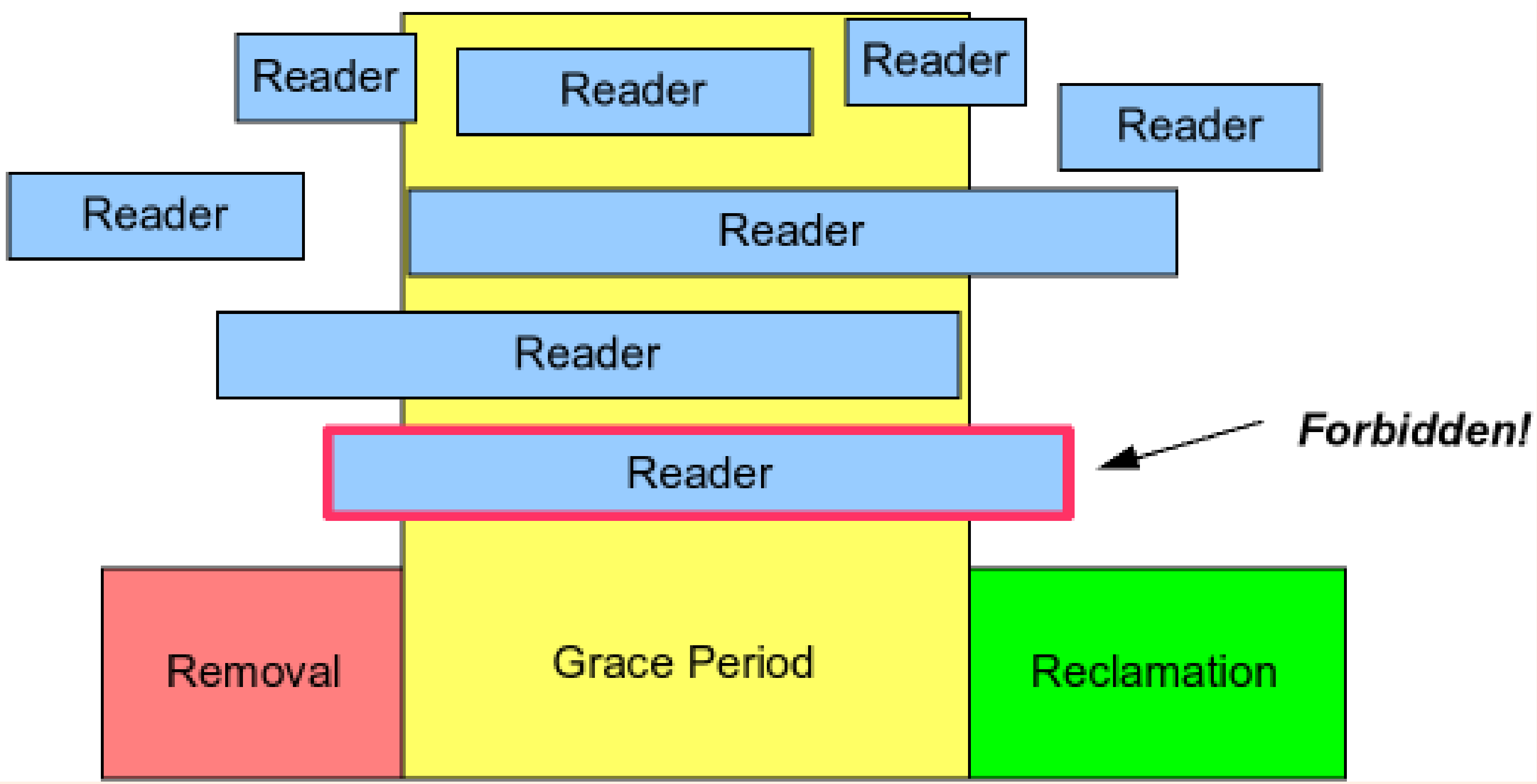
- **SYNC:** Insert a full barrier before every load or store
- **PowerA:** Separate loads with *lwsync*, pretending that POWER is multi-copy atomic

, and also leveraged

- **SAO:** Strong Access Ordering

[24] D. Lustig et al. ArMOR: defending against memory consistency model mismatches in heterogeneous architectures. ISCA, pages 388–400, 2015

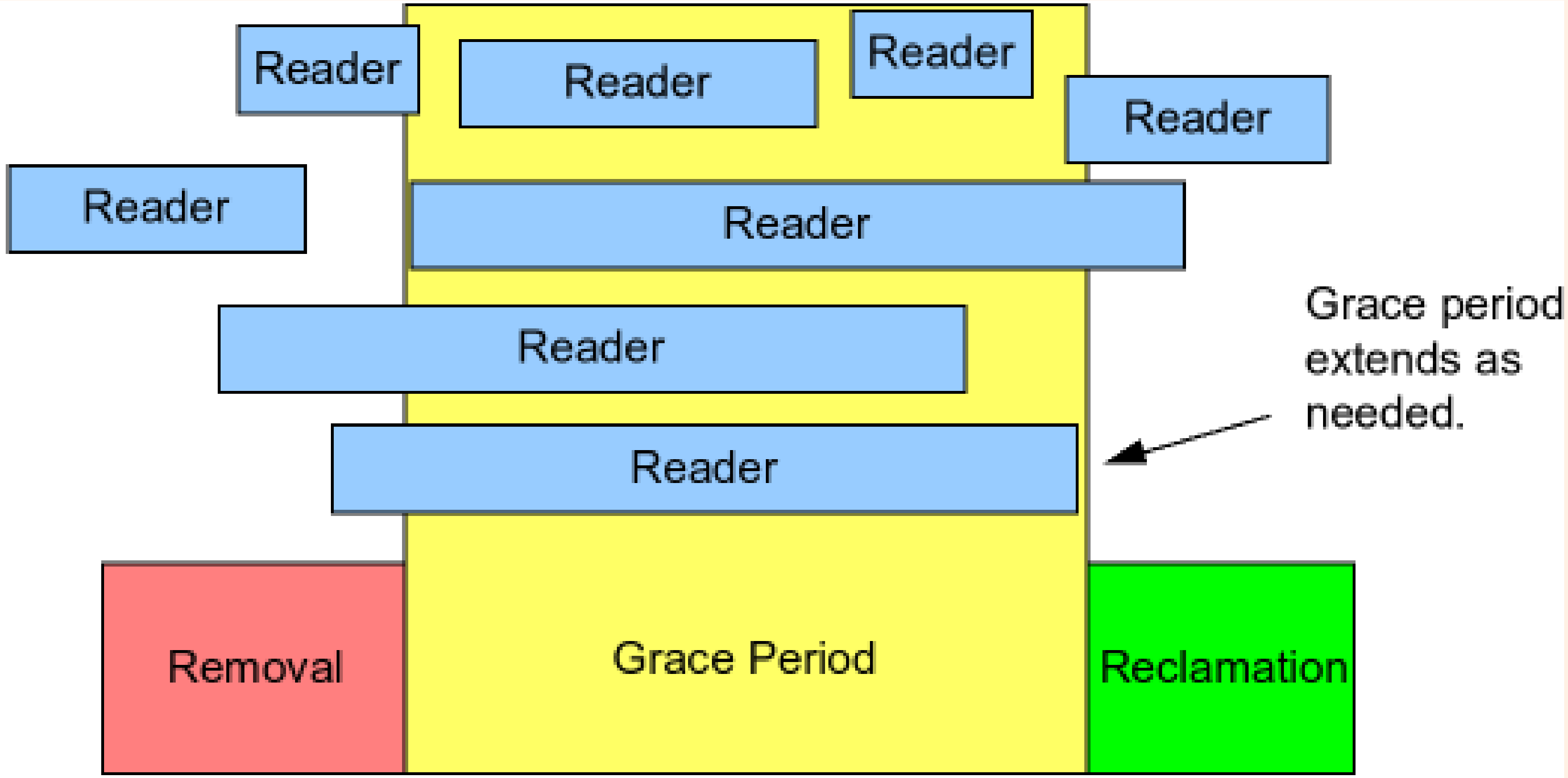
Read-Copy-Update (RCU)



Credit: Paul McKenney

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

Read-Copy-Update (RCU)



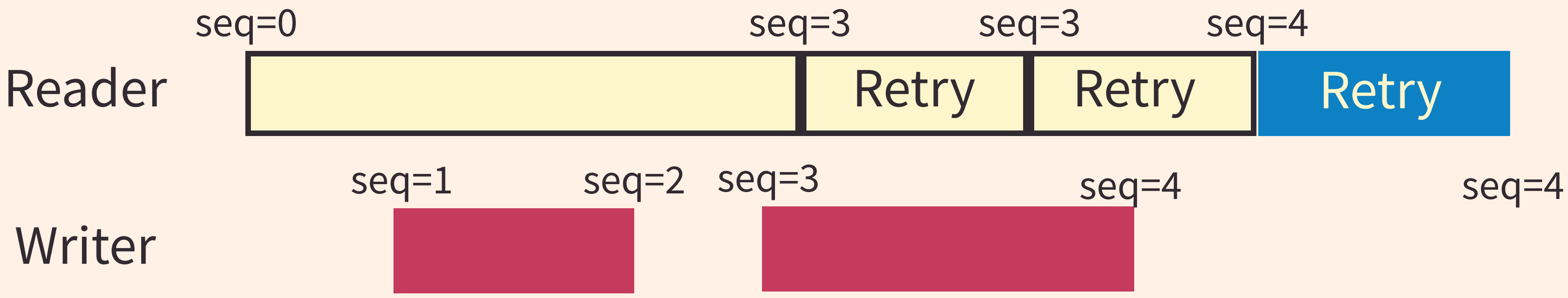
Credit: Paul McKenney

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

Sequence Locks

```
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



CLHT malloc requirement

```
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.*

[13] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. ASPLOS, pages

631–644, 2015

Multi-copy Atomicity

iriw litmus test

cpu0	cpu1	cpu2	cpu3

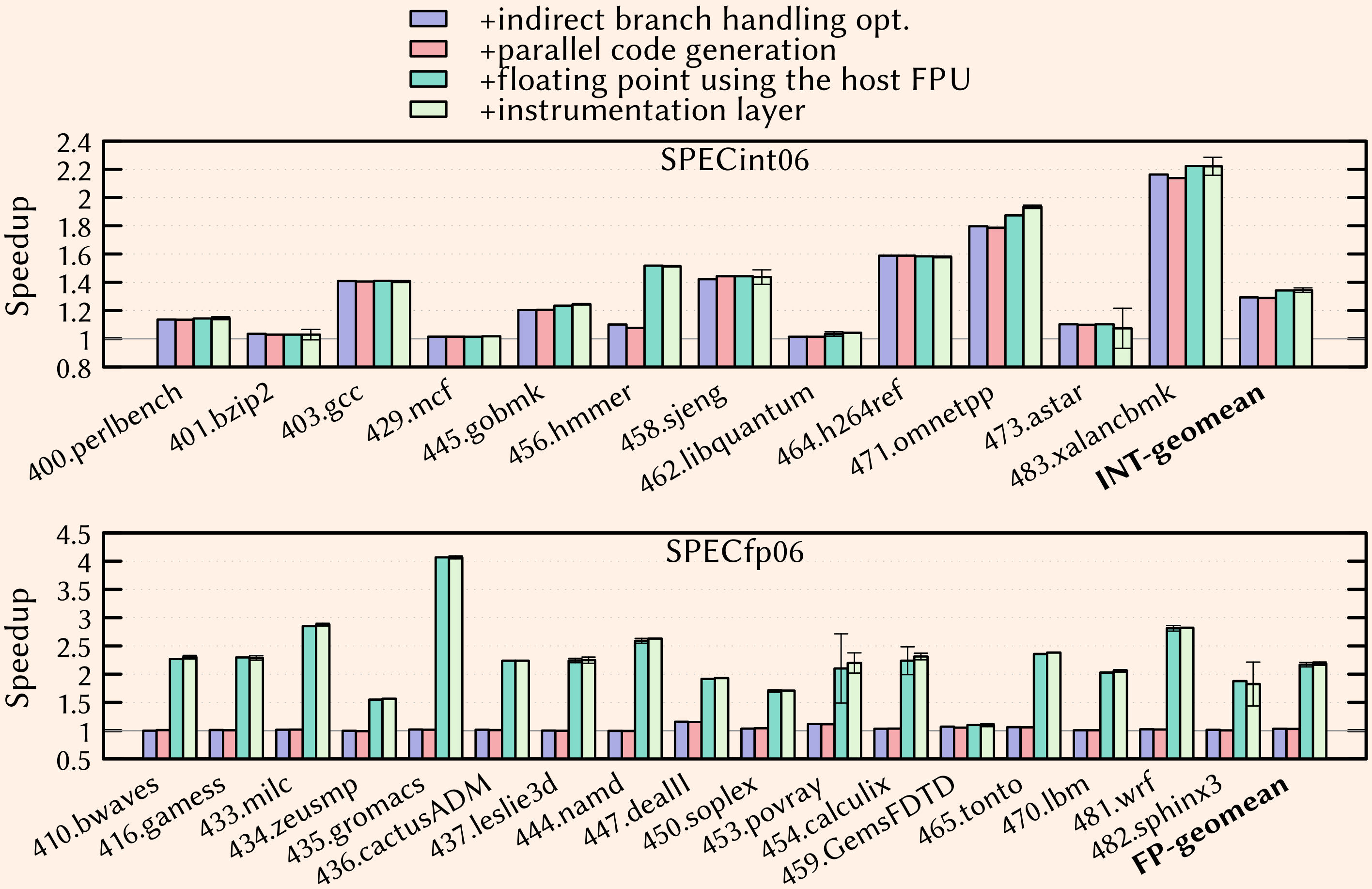
x=1	y=1	r1=x
		r2=y
			r3=y
			r4=x

- Forbidden outcome: r1 = r3 = 1, r2 = r4 = 0
- The outcome is forbidden on x86
- It is observable on POWER unless the loads are separated by a sync instruction

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

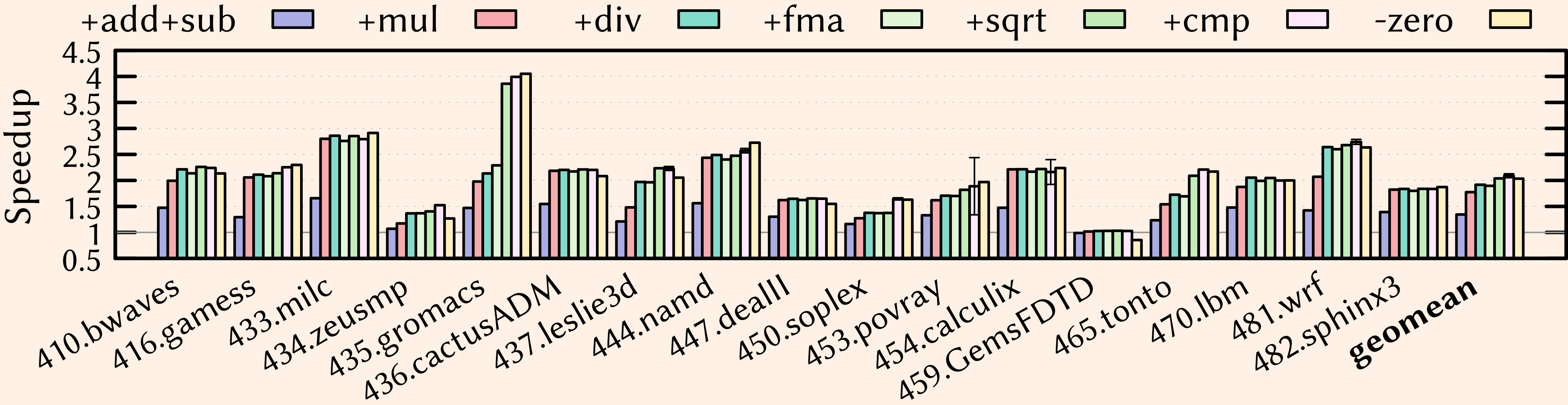
Evaluation

user-mode x86_64-on-x86_64. Baseline: Pico (i.e. QEMU v3.1.0)



FP per-op contribution

user-mode x86-on-x86



Qelt Instrumentation

- Fine-grained event subscription when guest code is translated
  - e.g. subscription to memory reads in Pin vs Qelt:

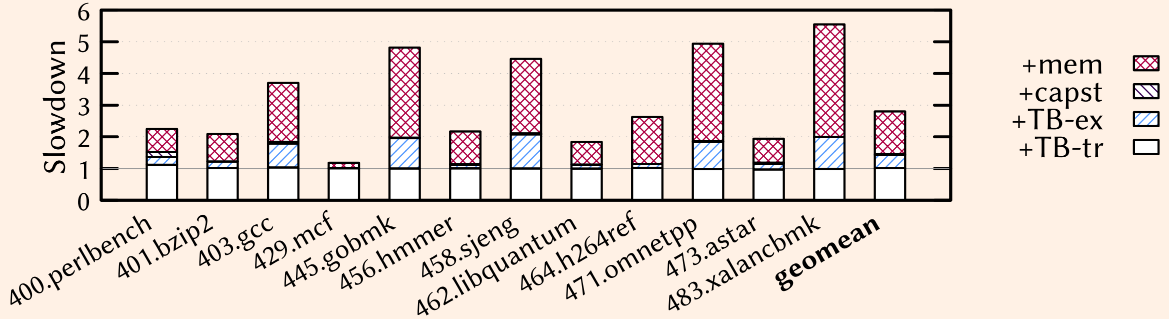
```
VOID Instruction(INS ins)
{
        if (INS_IsMemoryRead(ins))
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)MemCB, ...);
}
VOID Trace(TRACE trace, VOID *v)
{
        for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
                for (INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins))
                        Instruction(ins);
}
```

```
static void vcpu_tb_trans(qemu_plugin_id_t id, unsigned int cpu_index, struct qemu_plugin_tb *tb)
{
        size_t n = qemu_plugin_tb_n_insns(tb);
        size_t i;

        for (i = 0; i < n; i++) {
                struct qemu_plugin_insn *insn = qemu_plugin_tb_get_insn(tb, i);

                qemu_plugin_register_vcpu_mem_cb(insn, vcpu_mem, QEMU_PLUGIN_CB_NO_REGS, QEMU_PLUGIN_MEM_R);
        }
```

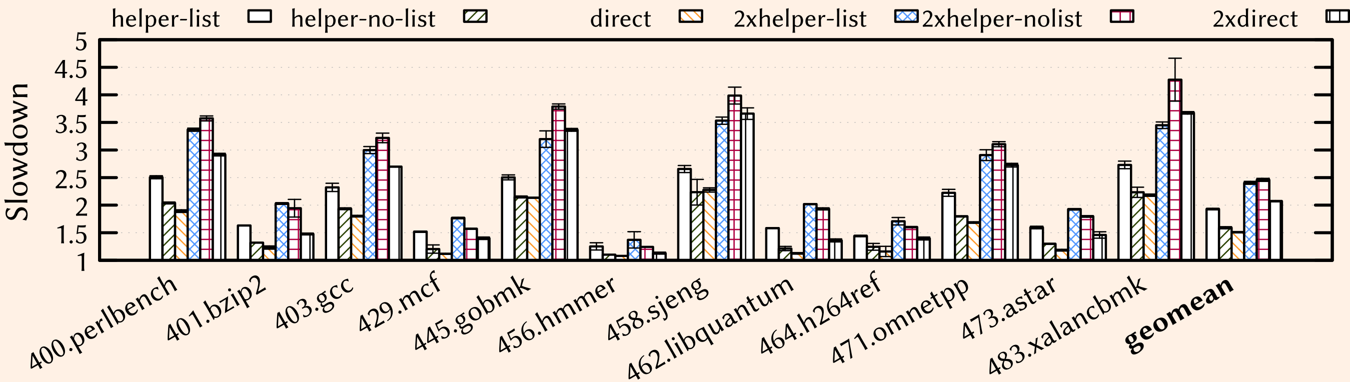
Instrumentation overhead

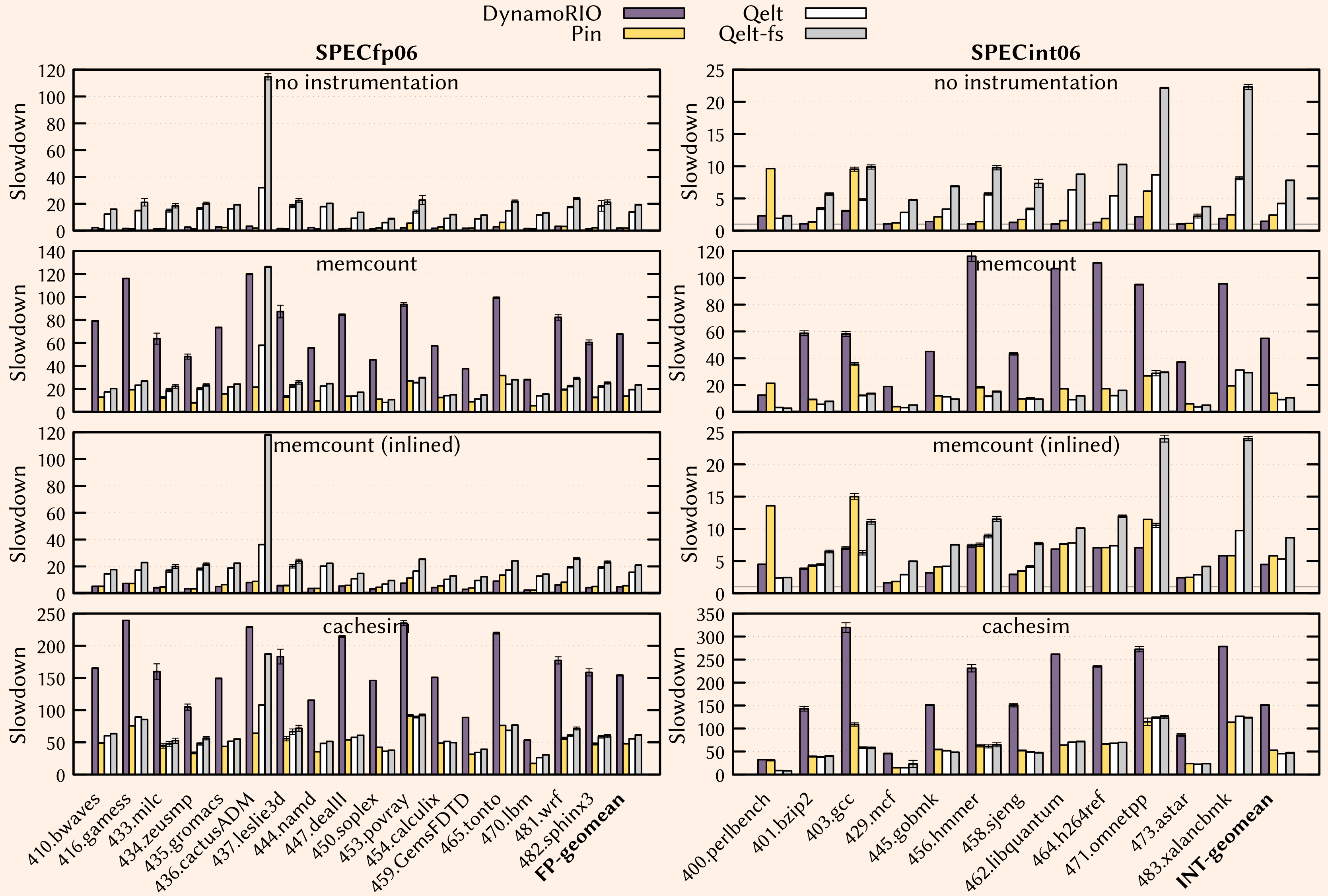
user-mode, x86_64-on-x86_64

- Typical overhead
  - Preemptive injection of instrumentation has negligible overhead



- Direct callbacks
  - Better than going via a helper (that iterates over a list) due to higher cache locality

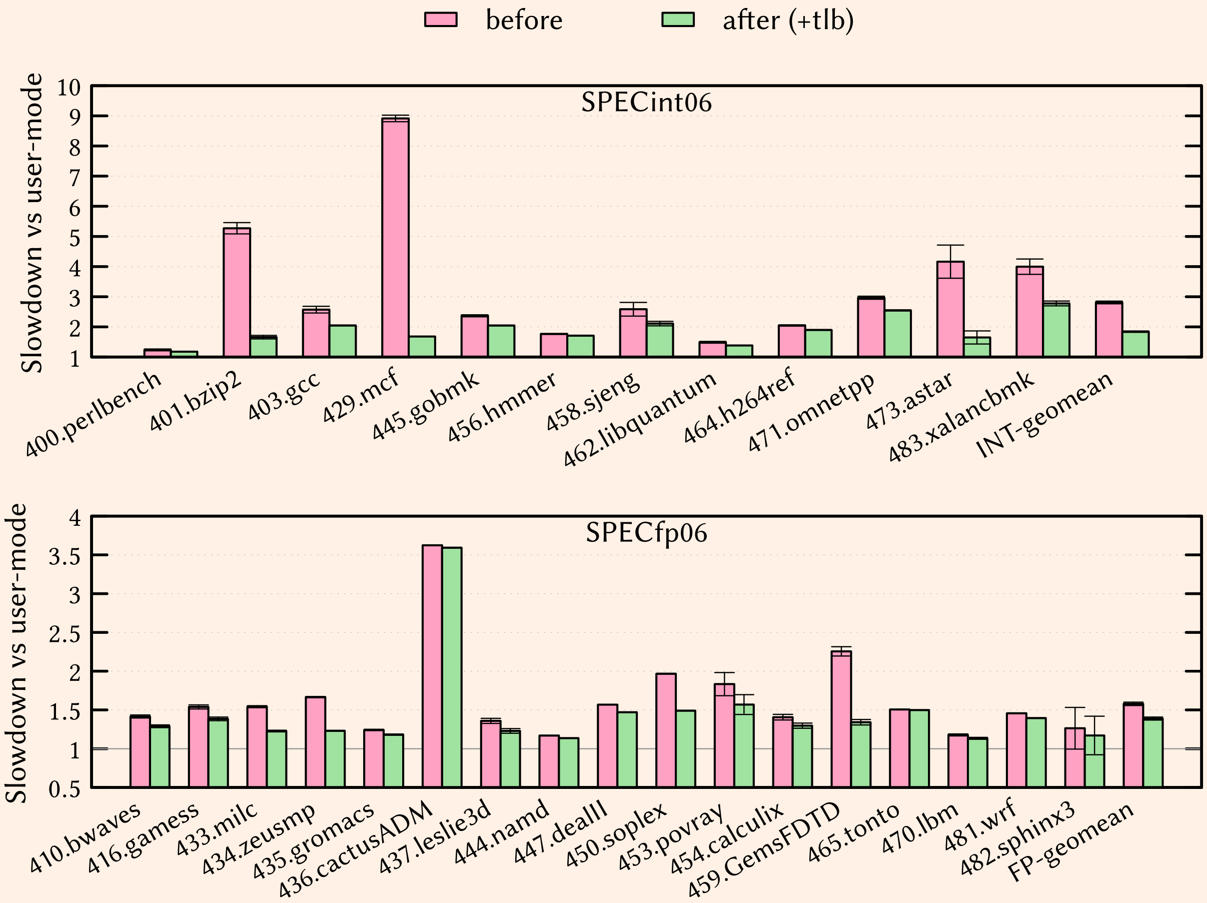




CactusADM an anomaly: TLB resizing doesn't kick in often enough (we only do it on TLB flushes)

SoftMMU overhead

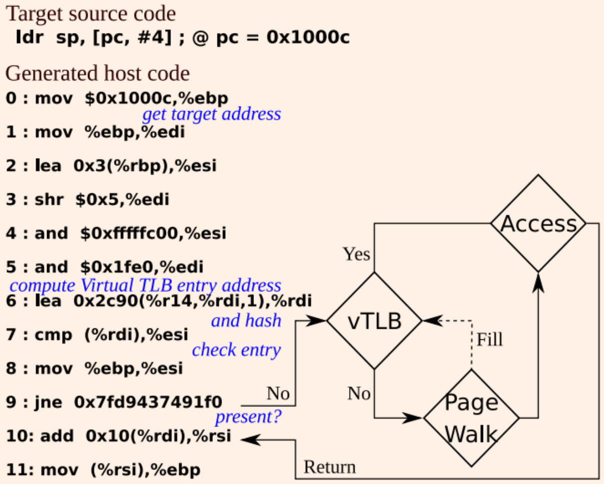
lower is better



CactusADM an anomaly: TLB resizing doesn't kick in often enough (we only do it on TLB flushes)

SoftMMU using shadow page tables

Before:
softMMU requires
many insns



Target source code

```
ldr  sp, [pc, #4] ; @ pc = 0x1000c
```

Generated host code

```
0 : mov  $0x1000c,%ebp
1 : mov  %ebp,%edi
2 : lea  0x3(%rbp),%esi
3 : shr  $0x5,%edi
4 : and  $0xfffffc00,%esi
5 : and  $0x1fe0,%edi
6 : lea  0x2c90(%r14,%rdi,1),%rdi
7 : cmp  (%rdi),%esi
8 : mov  %ebp,%esi
9 : jne  0x7fd9437491f0
10: add  0x10(%rdi),%rsi
11: mov  (%rsi),%ebp
```

Fig. 1. QEMU target memory accesses translation

after:
only 2 insns thanks to
shadow page tables

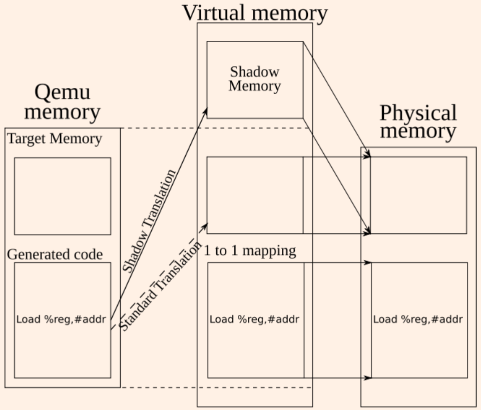
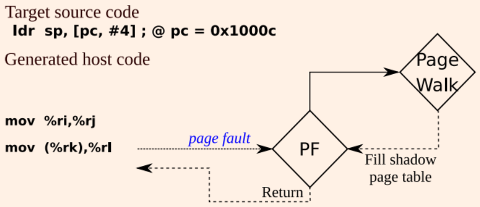


Fig. 2. Memory Layout



Target source code

```
ldr  sp, [pc, #4] ; @ pc = 0x1000c
```

Generated host code

```
mov  %ri,%rj
mov  (%rk),%rl
```

Fig. 3. QEMU target memory access with our solution

Advantages:

- High performance (almost 0 overhead for MMU emulation)
- Minimal modifications to QEMU compared to other options in the literature

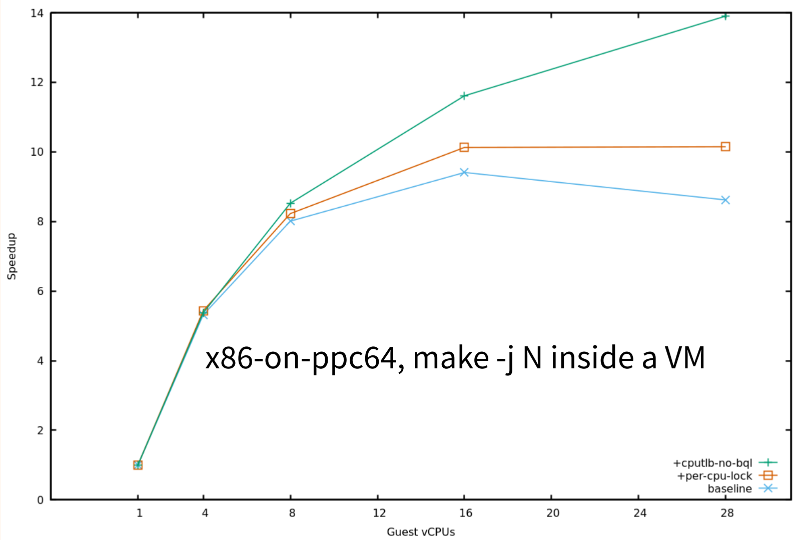
Disadvantages:

- Requires dune*, which means QEMU must be statically compiled
- Cannot work when target address space => host address space

Faravelon, Gruber, Pétrot. "Optimizing memory access performance using hardware assisted virtualization in retargetable dynamic binary translation. Euromicro Conference on Digital System Design (DSD), 2017.

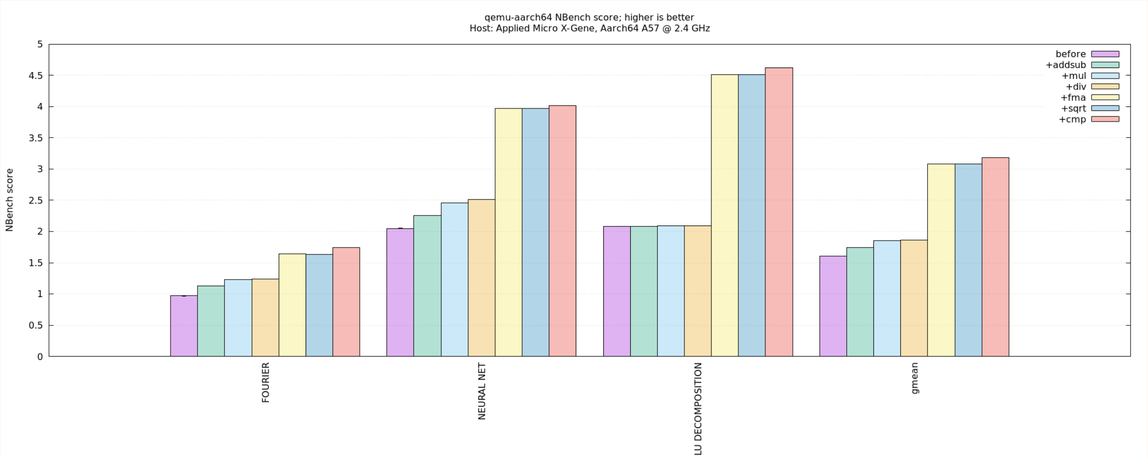
[*] Belay, Bittau, Mashtizadeh, Terei, Mazieres, Kozyrakis. "Dune: Safe user-level access to privileged cpu features." OSDI, 2012

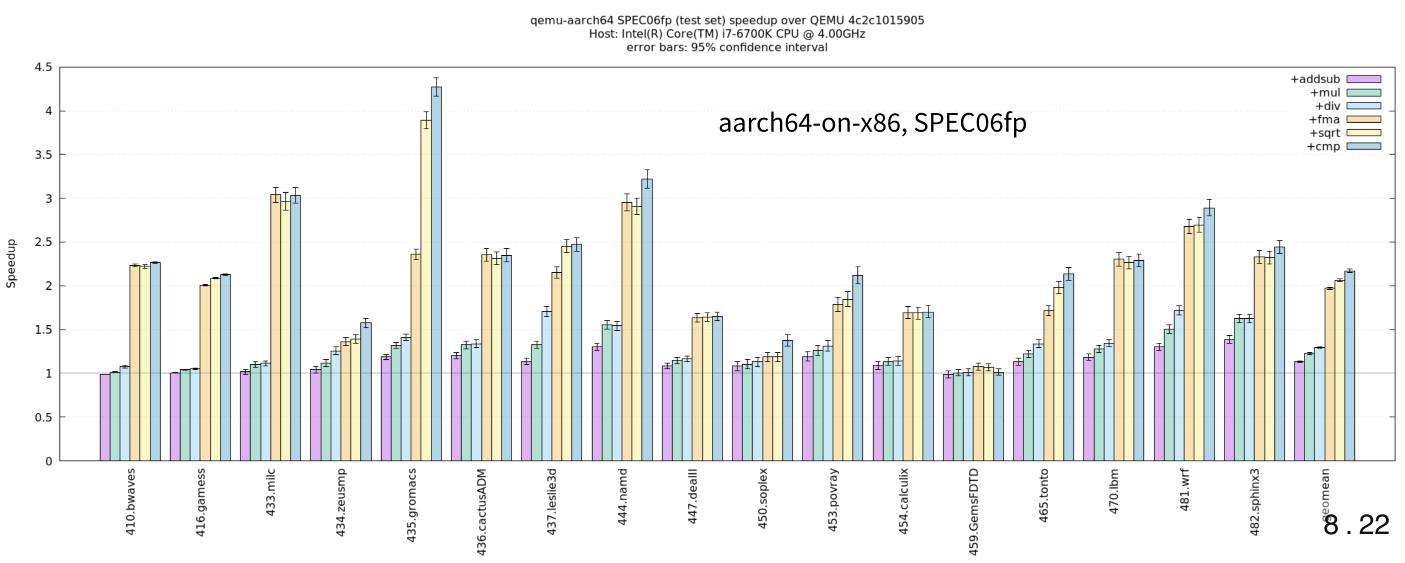
cross-ISA examples (1)



x86-on-ppc64, make -j N inside a VM

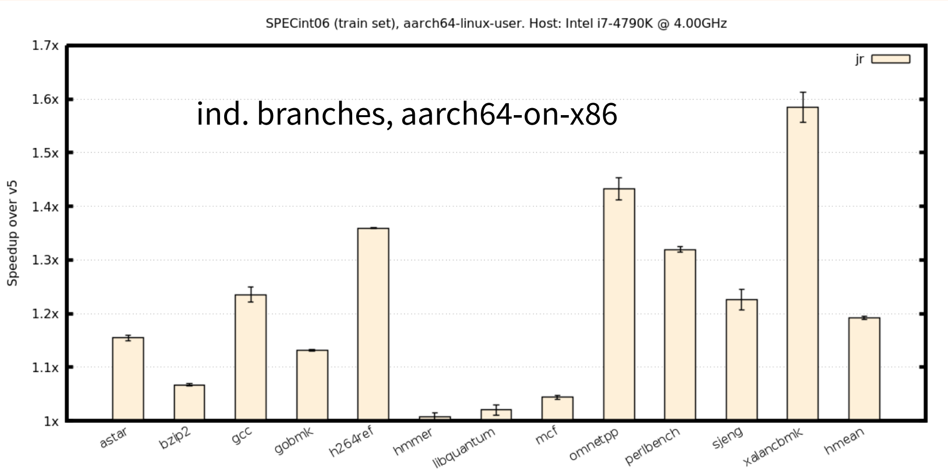
aarch64-on-aarch64, Nbench FP



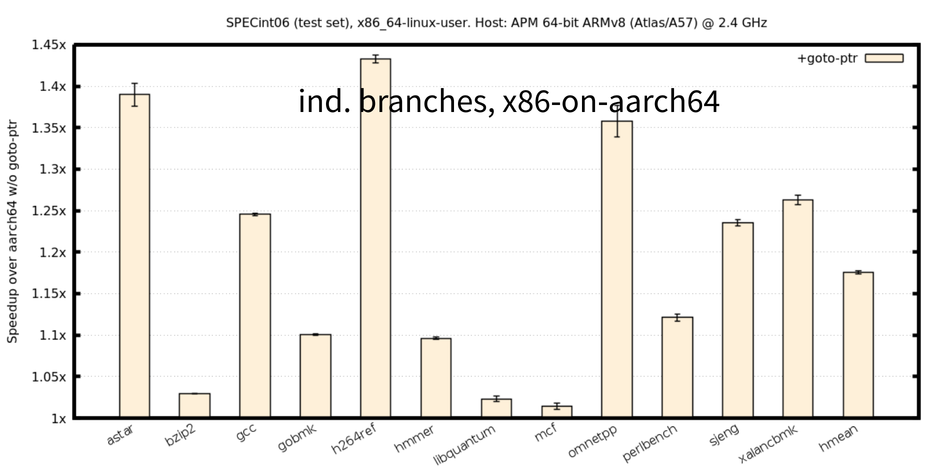


aarch64-on-x86, SPEC06fp

cross-ISA examples (2)



ind. branches, aarch64-on-x86



ind. branches, x86-on-aarch64

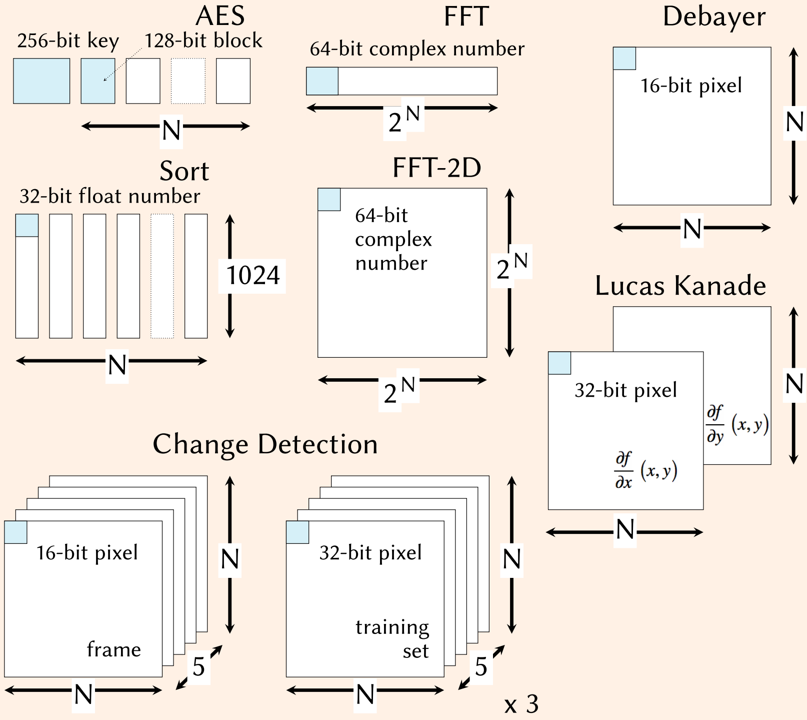
Ind. branches, RISC-V on x86, user-mode

```
bench     before  after1  after2  after3  final speedup
---------------------------------------------------------
aes       1.12s   1.12s   1.10s   1.00s   1.12
bigint    0.78s   0.78s   0.78s   0.78s   1
dhrystone 0.96s   0.97s   0.49s   0.49s   1.9591837
miniz     1.94s   1.94s   1.88s   1.86s   1.0430108
norx      0.51s   0.51s   0.49s   0.48s   1.0625
primes    0.85s   0.85s   0.84s   0.84s   1.0119048
qsort     4.87s   4.88s   1.86s   1.86s   2.6182796
sha512    0.76s   0.77s   0.64s   0.64s   1.1875
```

Ind. branches, RISC-V on x86, full-system

```
bench     before  after1  after2  after3  final speedup
---------------------------------------------------------
aes       2.68s   2.54s   2.60s   2.34s   1.1452991
bigint    1.61s   1.56s   1.55s   1.64s   0.98170732
dhrystone 1.78s   1.67s   1.25s   1.24s   1.4354839
miniz     3.53s   3.35s   3.28s   3.35s   1.0537313
norx      1.13s   1.09s   1.07s   1.06s   1.0660377
primes    15.37s  15.41s  15.20s  15.37s  1
qsort     7.20s   6.71s   3.85s   3.96s   1.8181818
sha512    1.07s   1.04s   0.90s   0.90s   1.1888889
```

Applications

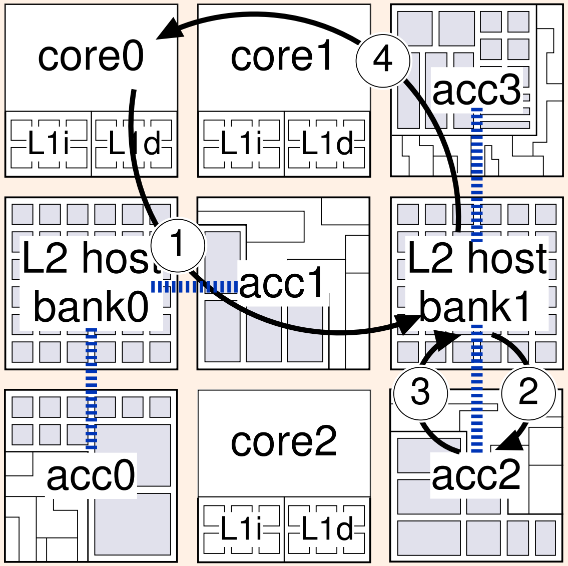


Application	Footprint N	Footprint Size (bytes)	Accelerator Area (um^2)	Scratchpad Area (um^2)	Scratchpad Size (bytes)
AES	5 - 1000	80 - 16K	192,792	–*	192
FFT	8 - 12	2K - 32K	337,770	299,605 (88%)	40K
FFT-2D	4 - 10	2K - 8M	146,199	98,273 (67%)	16K
Sort	8 - 128	32K - 524K	302,672	210,636 (69%)	25K
Debayer	16 - 1024	512 - 2M	207,206	196,522 (94%)	32K
Lucas Kanade	32 - 512	8K - 2M	588,001	538,775 (91%)	41K
Change Detection	32 - 512	71K - 18M	189,826	134,954 (71%)	16K

- Seven high-throughput applications from the PERFECT Benchmark Suite[*]
- Used High-Level Synthesis for productivity

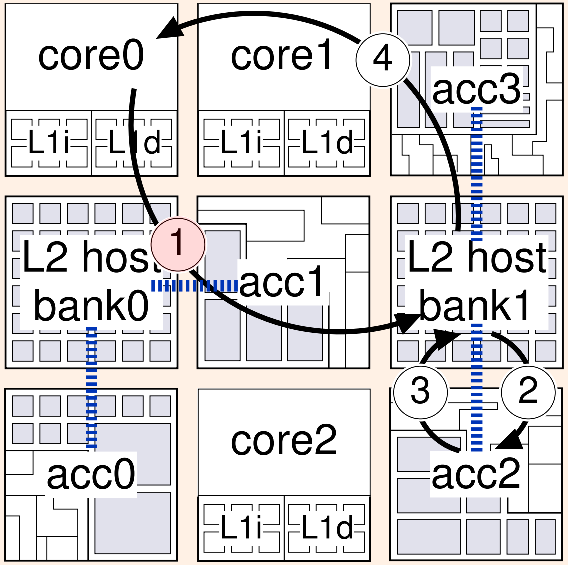
[*] http://hpc.pnl.gov/PERFECT/

High-Level Operation



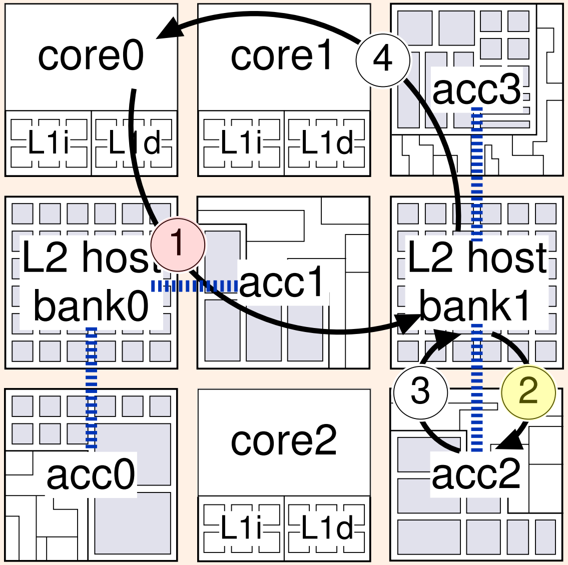
High-Level Operation

1. core0's L1 misses on a read from 0xf00, mapped to the L2's *logical bank1*



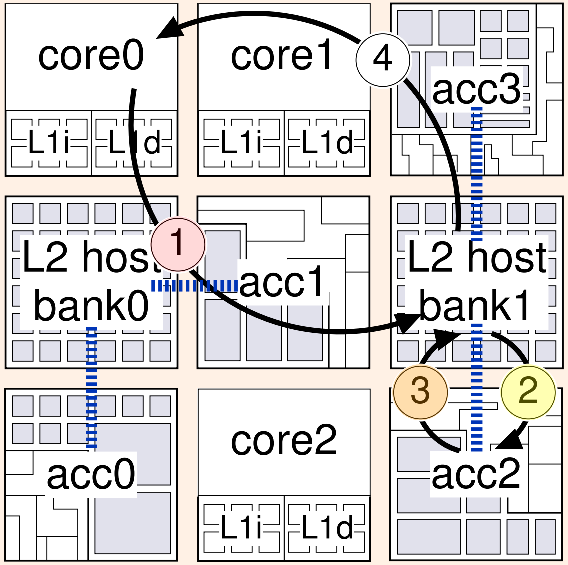
High-Level Operation

1. core0's L1 misses on a read from 0xf00, mapped to the L2's *logical bank1*
2. L2 bank1's tag array tracks block 0xf00 at acc2; sends request to acc2



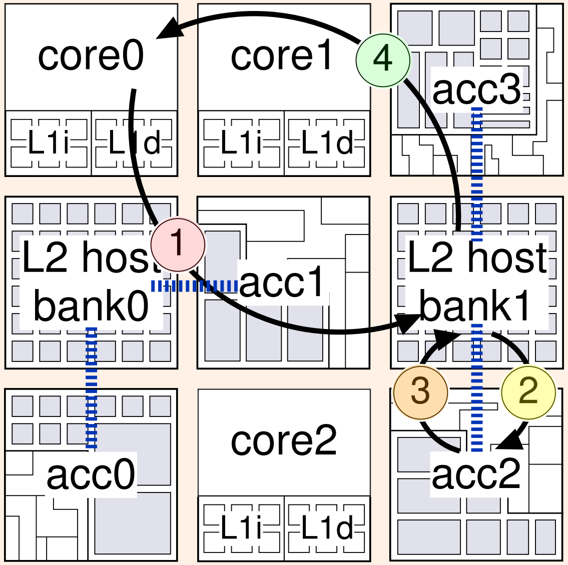
High-Level Operation

1. core0's L1 misses on a read from 0xf00, mapped to the L2's *logical bank1*
2. L2 bank1's tag array tracks block 0xf00 at acc2; sends request to acc2
3. acc2 returns the block to bank1



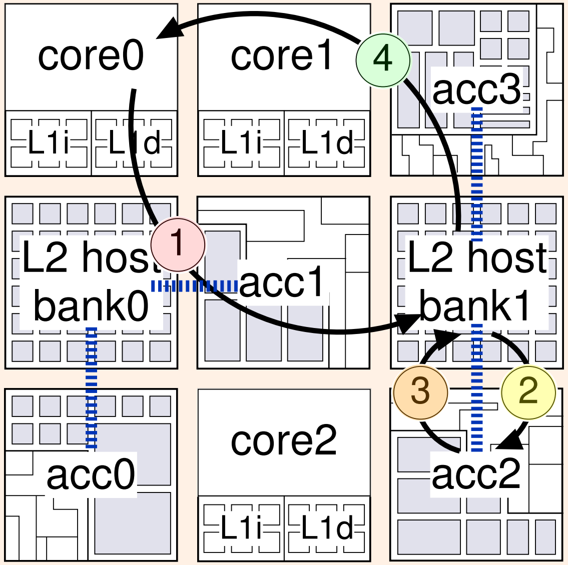
High-Level Operation

1. core0's L1 misses on a read from 0xf00, mapped to the L2's *logical bank1*
2. L2 bank1's tag array tracks block 0xf00 at acc2; sends request to acc2
3. acc2 returns the block to bank1
4. bank1 sends the block to core0



High-Level Operation

1. core0's L1 misses on a read from 0xf00, mapped to the L2's *logical bank1*
2. L2 bank1's tag array tracks block 0xf00 at acc2; sends request to acc2
3. acc2 returns the block to bank1
4. bank1 sends the block to core0

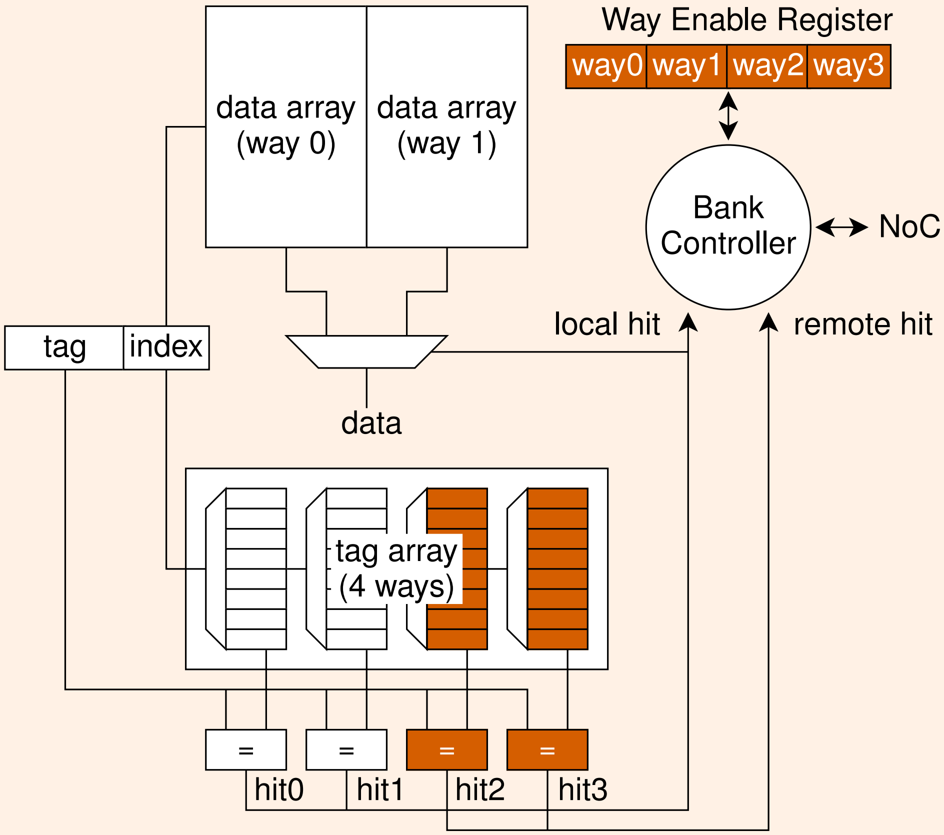


✗ **Additional latency for hits** to blocks stored in accelerators

✓ Return via the host bank guarantees the **host bank** is the **only coherence synchronization point**

- No changes to coherence protocol needed

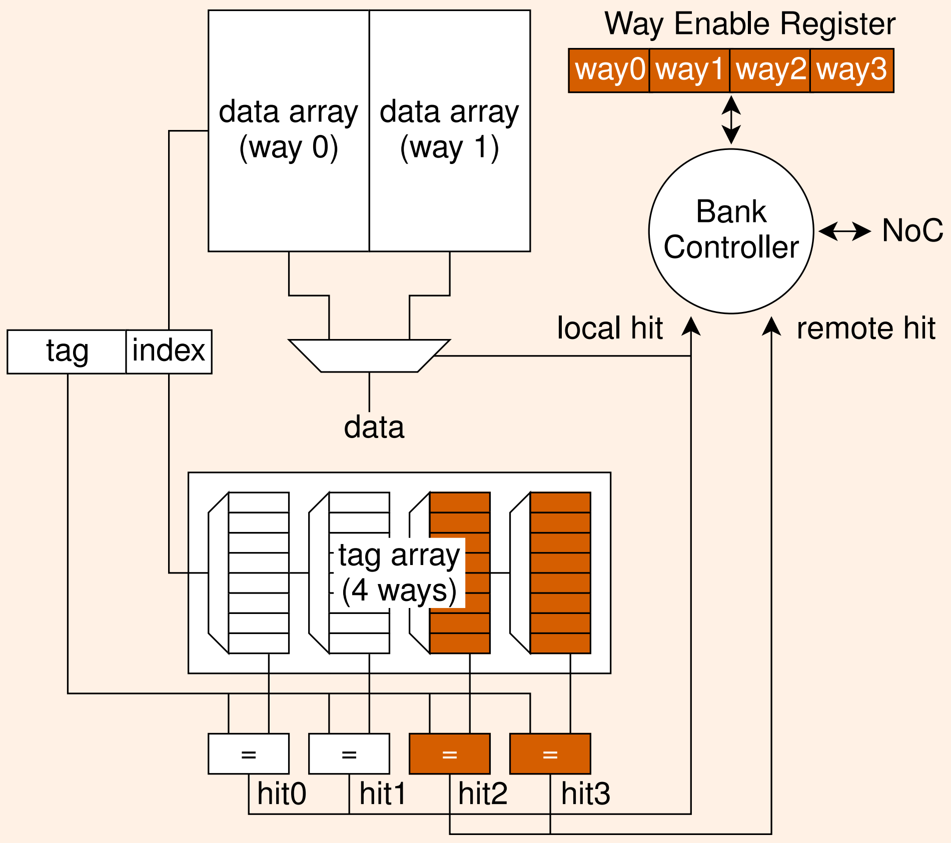
ROCA Host Bank



4-way example: 2 local, 2 remote ways

- **Enlarged tag array** for accelerator blocks
  - Ensures **modifications** to accelerators are **simple**

ROCA Host Bank

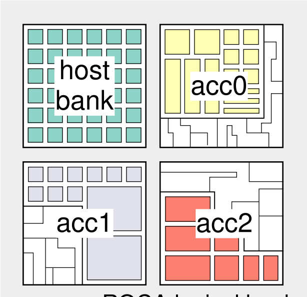


4-way example: 2 local, 2 remote ways

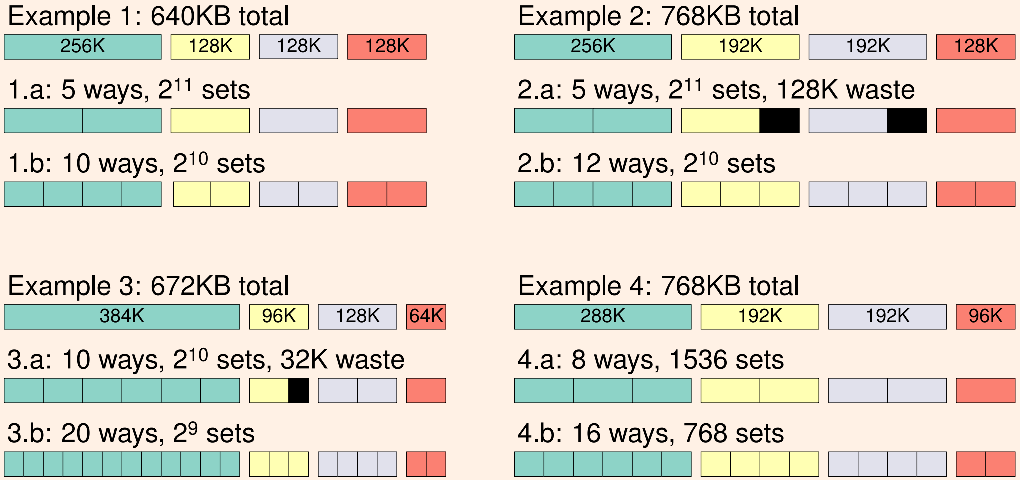
- **Enlarged tag array** for accelerator blocks
  - Ensures **modifications** to accelerators are **simple**
- Leverages Selective Cache Ways [*] to **adapt to accelerators' intermittent availability**
  - Dirty blocks are flushed to DRAM upon accelerator reclamation

[*] David H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation", ISCA'99

Logical Bank Way Allocation



ROCA logical bank



Example 1: 640KB total
256K 128K 128K 128K

1.a: 5 ways, 2^{11} sets

1.b: 10 ways, 2^{10} sets

Example 2: 768KB total
256K 192K 192K 128K

2.a: 5 ways, 2^{11} sets, 128K waste

2.b: 12 ways, 2^{10} sets

Example 3: 672KB total
384K 96K 128K 64K

3.a: 10 ways, 2^{10} sets, 32K waste

3.b: 20 ways, 2^{9} sets

Example 4: 768KB total
288K 192K 192K 96K

4.a: 8 ways, 1536 sets

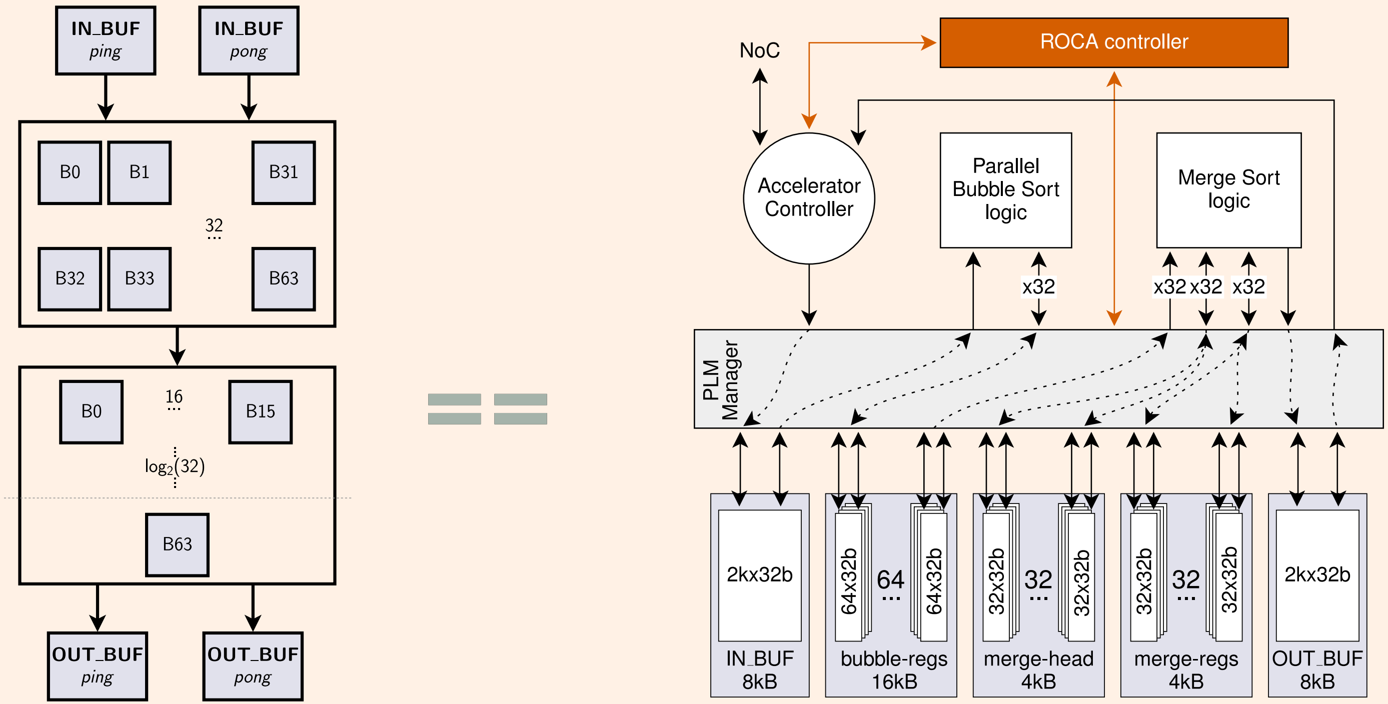
4.b: 16 ways, 768 sets

- Increasing associativity helps minimize waste due to uneven memory sizing across accelerators (Ex. 2 & 3)

- Power-of-two number of sets not required (Ex. 4), but
  - complicates set assignment logic [*]
  - requires full-length tags: modulo is not bit selection anymore

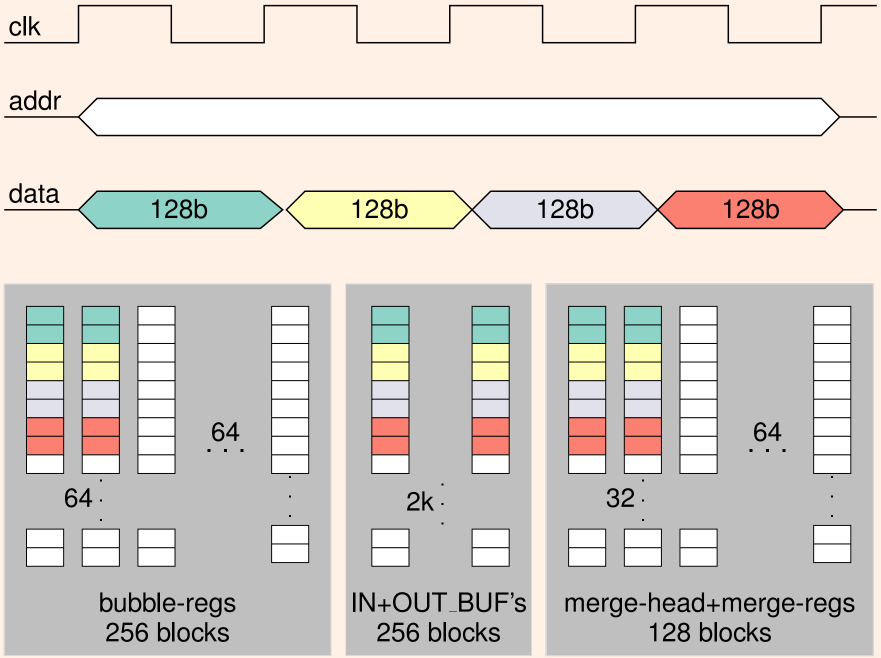
[*] André Seznec, "Bank-interleaved cache or memory indexing does not require Euclidean division", IWDDD'15

Coalescing PLMs



- **PLM manager** exports same-size dual-ported SRAM banks as multi-ported memories using MUXes
- ROCA requires an **additional NoC-flit-wide port**, e.g. 128b

Coalescing PLMs



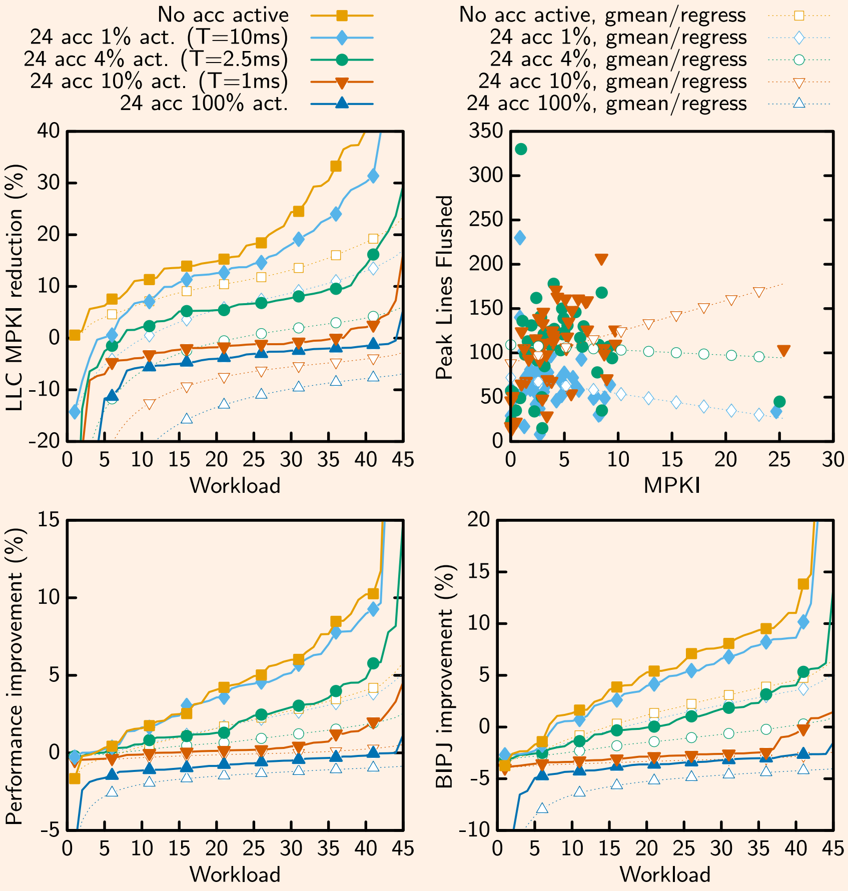
- SRAMs are accessed in parallel to match the NoC flit bandwidth
  - Bank offsets can be computed cheaply with a LUT + simple logic
  - Discarding small banks and SRAM bits a useful option

ROCA: Area Overhead

- Host bank's **enlarged tag array**
  - **5-10%** of the area of the data it tags (2b+tag per block)

- Tag storage for **standalone directory** if it wasn't there already
  - Inclusive LLC would require prohibitive numbers of recalls
  - Typical overhead: **2.5%** of LLC area when LLC = 8x priv

- **Additional logic:** way selection, PLM coalescing logic
  - **Negligible** compared to tag-related storage



- **Sensitivity studies** sweeping accelerator activity over
  - **space** (which accelerators are reclaimed)
  - **time** (how frequently they are reclaimed)
- **Key result:** Accelerators with **idle windows >10ms** are prime candidates for ROCA
  - perf/eff. within 10/20% of that with 0% activity