Cross-ISA Machine Instrumentation using Fast and Scalable Dynamic Binary Translation

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

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

Abstract

The rise in instruction set architecture (ISA) diversity and the growing adoption of virtual machines are driving a need for fast, scalable, full-system, cross-ISA emulation and instrumentation tools. Unfortunately, achieving high performance for these cross-ISA tools is challenging due to dynamic binary translation (DBT) overhead and the complexity of instrumenting full-system emulators.

In this paper we improve cross-ISA emulation and instrumentation performance through three novel techniques. First, we increase floating point (FP) emulation performance by observing that most FP operations can be correctly emulated by surrounding the use of the host FP unit with a minimal amount of non-FP code. Second, we introduce the design of a translator with a shared code cache that scales for multi-core guests, even when they generate translated code in parallel at a high rate. Third, we present an ISA-agnostic instrumentation layer that can instrument guest operations that occur outside of the DBT's intermediate representation (IR), which are common in full-system emulators.

We implement our approach in Qelt, a high-performance cross-ISA machine emulator and instrumentation tool based on QEMU. Our results show that Qelt scales to 32 cores when emulating a guest machine used for parallel compilation, which demonstrates scalable code translation. Furthermore, experiments based on SPEC06 show that Qelt (1) outperforms QEMU as a full-system cross-ISA machine emulator by 1.76×/2.18× for integer/FP workloads, (2) outperforms state-of-the-art, cross-ISA, full-system instrumentation tools by 1.5×-3×, and (3) can match the performance of Pin, a state-of-the-art, same-ISA DBI tool, when used for complex instrumentation such as cache simulation.

VEE '19, April 14, 2019, Providence, RI, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313811

CCS Concepts • **Software and its engineering → Virtual machines**; *Just-in-time compilers.*

Keywords Dynamic Binary Translation, Binary Instrumentation, Machine Emulation, Floating Point, Scalability

ACM Reference Format:
Emilio G. Cota and Luca P. Carloni. 2019. Cross-ISA Machine Instrumentation using Fast and Scalable Dynamic Binary Translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3313808.3313811

1 Introduction

The emergence of virtualization, combined with the rise of multi-cores and the increasing use of different instruction set architectures (ISAs), highlights the need for fast, scalable, cross-ISA machine emulators. These emulators typically achieve high performance and portability by leveraging dynamic binary translation (DBT). Unfortunately, state-of-the-art cross-ISA DBT suffers under two common scenarios. First, emulated floating point (FP) code incurs a large overhead (2× slowdowns are typical [6]), which hinders the use of these tools for guest applications that are FP-heavy, such as the emulation of graphical systems (e.g. Android) or as a front-end to computer architecture simulators that run scientific workloads. This FP overhead is rooted in the difficulty of correctly emulating the guest's floating point unit (FPU), which can greatly differ from that of the host. This is commonly solved by forgoing the use of the host FPU, using instead a much slower *soft-float* implementation, i.e. one in which no FP instructions are executed on the host.

Second, efficient scaling of code translation when emulating multi-core systems is challenging. The scalability of code translation is not an obvious concern in DBT, since code execution is usually more common. However, some server workloads [12] as well as parallel compilation jobs (e.g., in cross-compilation testbeds of software projects that support several ISAs, such as the Chromium browser [1]) can show both high parallelism and large instruction footprints, which can limit the scalability of their emulation, particularly when using a shared code cache.

In this paper we first address these two challenges by improving cross-ISA DBT performance and scalability. We then combine these improvements with a novel ISA-agnostic instrumentation layer to produce a cross-ISA dynamic binary instrumentation (DBI) tool, whose performance is higher than that of existing cross-ISA DBI tools (e.g., [20, 25, 49, 54]).

Same-ISA DBI tools such as DynamoRIO [11] and Pin [33] provide highly customizable instrumentation in return for modest performance overheads, and as a result have had tremendous success in enabling work in fields as diverse as security (e.g., [16, 29]), workload characterization (e.g., [9, 45]), deterministic execution (e.g., [3, 18, 38]) and computer architecture simulation (e.g., [13, 28, 37]). Our goal is thus to narrow the performance gap between cross-ISA DBI tools and these state-of-the-art same-ISA tools, in order to elicit similarly diverse research, yet for a variety of guest ISAs.

Our contributions are as follows:

- We describe a technique to leverage the host FPU when performing cross-ISA DBT to achieve high emulation performance. The key insight behind our approach is to limit the use of the host FPU to the large subset of guest FP operations that yield identical results on both guest and host FPUs, deferring to soft-float otherwise (Section 3.1).
- We present the design of a parallel cross-ISA DBT engine that, while remaining memory efficient via the use of a shared code cache, scales for multi-core guests that generate translated code in parallel (Section 3.2).
- We present an ISA-agnostic instrumentation layer that converts a cross-ISA DBT engine into a low-overhead cross-ISA DBI tool, with support for state-of-the-art instrumentation features such as instrumentation injection at the granularity of individual instructions, as well as the ability to instrument guest operations that are emulated outside the DBT engine (Section 3.3).

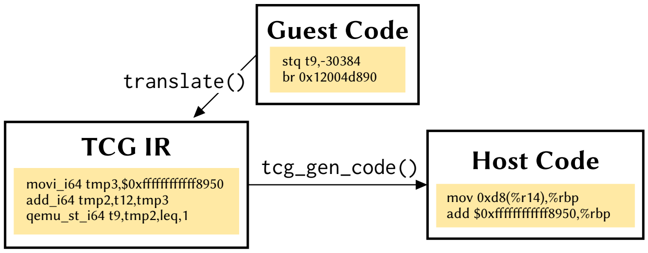
We implement our approach in Qelt, a cross-ISA machine emulator and DBI tool based on QEMU [6]. Qelt achieves high performance by combining our novel techniques with further DBT optimizations, which are described in Section 3.4. As shown in Section 4, Qelt scales when emulating a guest machine used for parallel compilation, and it outperforms (1) QEMU as both a user-mode and full-system emulator and (2) state-of-the-art cross-ISA instrumentation tools. Last, for complex instrumentation such as cache simulation, Qelt can match the performance of state-of-the-art, same-ISA DBI tools such as Pin.

2 Background

2.1 Cross-ISA DBT

Target and Host ISAs. We use the term *target ISA* (*target* for short) to refer to the ISA of the workload being emulated. Emulation is run on a physical *host* machine, whose ISA might be different from the target's.

User-mode and full-system emulation. User-mode emulation applies DBT techniques to target code, but executes system calls on the host. While user-mode emulation can employ a system call translation layer, for example to run 32-bit programs on 64-bit hosts or vice versa, it is generally limited to programs that are compiled for the same operating system as the host. Full-system emulation refers to the emulation of an entire system: target hardware is emulated, which allows DBT-based virtualization of an entire target *guest*. The guest's ISA and operating system are independent from the host's.



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

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

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

Figure 1. An example of portable cross-ISA DBT. Alpha code is translated into x86_64 using QEMU's TCG IR.

Virtual CPUs (vCPUs). In *user-mode*, a vCPU corresponds to an emulated program's thread of execution. In *full-system* mode, a vCPU is the thread of execution used to emulate a core of the guest CPU.

Translation Block (TB). Target code is translated into groups of adjacent non-branch instructions that form a so-called TB. Target execution is thus emulated as a traversal of connected TBs. TBs and basic blocks both terminate at branch instructions. TBs, however, can be terminated earlier, e.g. to force an exit from execution to handle at run-time an update to emulated hardware state.

Intermediate Representation (IR). Portability across ISAs is typically achieved by relying on an IR. The example in Figure 1 shows how QEMU's IR, called Tiny Code Generator (TCG), facilitates independent development of target and host-related code.

Helpers. Sometimes the IR is not rich enough to represent complex guest behavior, or it could only do so by overly complicating target code. In such cases, *helper* functions are used to implement this complex behavior outside of the IR, leveraging the expressiveness of a full programming language. Helpers are oftentimes used to emulate instructions that interact with hardware state, such as the memory management unit's translation lookaside buffer (TLB).

Plugins and callbacks. Instrumentation code is built into *plugins*, i.e. shared libraries that are dynamically loaded by the DBI tool. Plugins subscribe to guest events of their interest by registering functions (*callbacks*) that are called as soon as the appropriate event occurs.

2.2 Floating Point Emulation

Faithful emulation of floating point (FP) instructions is more complex than just generating the correct floating point result.

Correct emulation requires emulating the entire *floating point environment*, that is, hardware state that configures the behavior of the FP operations (e.g. setting the rounding mode) and keeps track of FP flags (e.g. invalid, divide-by-zero, under/overflow, inexact), optionally raising exceptions in the guest as the flags are set.

The floating point environment is defined in the specification of each ISA. Despite the compliance of many commercial ISAs with the IEEE-754 standard [2], emulation remains non-trivial for several reasons: (1) the standard leaves details to be decided by the implementation (e.g. underflow tininess detection or the raising of flags in certain scenarios), (2) some features have wide adoption yet are not part of the standard (e.g. flush-to-zero and denormals-are-zeros), and (3) some widely used implementations are not compliant with the standard (e.g. ARM NEON [4]).

It is then not surprising that cross-ISA emulators typically trade performance (e.g. 2× slowdown for QEMU [6]) for portability by invoking soft-float emulation code via helpers. One of our goals is to recover most—if not all—of this performance by leveraging the host FPU for the vast majority of guest FP instructions.

3 Qelt Techniques

We now present the techniques that allow Qelt to achieve high performance and portability. First, we accelerate the emulation of FP instructions by leveraging the host FPU for the vast majority of guest FP instructions. Next, we parallelize multi-core emulation with a DBT engine that avoids global locks while keeping a shared code cache. Third, we describe an ISA-agnostic instrumentation layer that allows us to convert a DBT engine into a cross-ISA DBI tool. Finally, we cover some additional DBT optimizations that further increase Qelt's speed.

3.1 Fast FP Emulation using the Host FPU

Speeding up the emulation of guest FP instructions using the host's FPU is deceptively simple. A naïve implementation would first clear the host's FP flags, execute the equivalent FP instruction on the host, check the host FP flags and then raise the appropriate flags on the guest. This approach would be trivial to implement, and would be correct for many FP instructions. Performance, however, would be abysmal, as we show in Section 4.4. This is due to the lack of optimizations *in the FPU hardware* for the use case of clearing/checking the FP flags, which is justified by how rare these operations are in FP workloads.

Our approach is thus to leverage the host FPU but only for a subset of all possible FP operations. Fortunately, as we discuss next, this subset covers the vast majority of FP instructions in real-world code. Our approach is guided by the following observations:

```
0   float64 float64_mul(float64 a, float64 b, fp_status *st)
1   {
2     float64_input_flush2(&a, &b, st);
3     if (likely(float64_is_zero_or_normal(a) &&
4                float64_is_zero_or_normal(b) &&
5                st->exception_flags & FP_INEXACT &&
6                st->round_mode == FP_ROUND_NEAREST_EVEN)) {
7       if (float64_is_zero(a) || float64_is_zero(b)) {
8         bool neg = float64_is_neg(a) ^ float64_is_neg(b);
9         return float64_set_sign(float64_zero, neg);
10      } else {
11        double ha = float64_to_double(a);
12        double hb = float64_to_double(b);
13        double hr = ha * hb;
14        if (unlikely(isinf(hr))) {
15          st->float_exception_flags |= float_flag_overflow;
16        } else if (unlikely(fabs(hr) <= DBL_MIN)) {
17          goto soft_fp;
18        }
19        return double_to_float64(hr);
20      }
21    }
22  soft_fp:
23    return soft_float64_mul(a, b, st);
24  }
```

Figure 2. Pseudo-code of a Qelt-accelerated double-precision multiplication.

- FP workloads operate mostly on normal or zero numbers. In other words, speeding up the handling of denormals, infinities or not-a-numbers (NaNs) is not necessary to accelerate most FP workloads.
- With some trivial checks, we can select FP operations capable of raising only three exceptions: inexact, overflow and underflow.
- FP flags are rarely cleared by FP workloads. This explains why FP flags are cumulative (or *sticky*). That is, once an exception occurs, the corresponding bit remains set until it is explicitly cleared by software.
- Due to FP's finite precision, most FP operations raise the inexact flag.
- FP workloads rarely change the rounding mode, which defaults to round-to-nearest-even.

We thus accelerate the *common case*, i.e.: the rounding is round-to-nearest-even, inexact is already set, and the operands are checked to limit the flags that the operation can raise. Otherwise, we defer to a soft-float implementation.

Figure 2 shows the application of our approach to double-precision multiplication. First, we flush the operands to zero if flush-to-zero mode is enabled (line 2). Next, we check whether this is the common case, checking both operands as well as the emulated FP environment (3-6). If so, we first perform a small optimization, checking for the trivial case of either operand being zero (7-10). As shown in Section 4.4, this can improve performance for some workloads, since we avoid accessing the host FPU's registers. The *else* branch leverages the host FPU to compute the result (12-14). Finally, we handle overflow (16-17) and resort to soft-float if there is a risk of underflow (18-19).

The key insight behind our technique is the identification of a large set of FP operations that can be run on the host FPU, while deferring corner cases (whether in the result to be computed or in the flags to be raised) to the slower soft-float code. We implement this technique in Qelt, accelerating commonly-used single and double-precision operations, namely addition, subtraction, multiplication, division, square root, comparison and fused multiply-add. Profiling shows that on average Qelt accelerates (i.e., does not defer to soft-float) 99.18% of FP instructions in SPECfp06 benchmarks.

Our approach has three main limitations. First, it does not speed up applications that frequently clear the inexact flag or that mostly operate with denormal numbers. Native hardware does not perform well in these cases either, so deferring to soft-float for these is appropriate. Second, applications with rounding other than round-to-nearest-even are not accelerated. Our approach could be changed to handle other rounding modes (particularly with regards to overflow), but we believe that the corresponding slowdown due to additional branches in the code is not justified, given how rare it is to find applications that require a non-default rounding mode. Last, while our approach does not require the guest or host to be IEEE-754 compliant (since compliance diverges only for operands outside of the *common case*), it requires the host FPU to natively support the same precision as that of the guest. This is, however, unlikely to be an issue in practice, since most FP workloads use only single and double precision, which are widely supported by commercial CPUs.

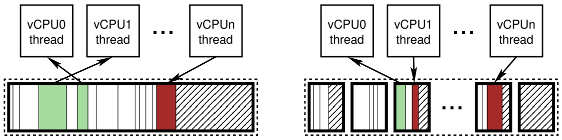
3.2 Scalable Dynamic Binary Translation

The state of the art in DBT engines with a shared code cache enables parallel guest execution, which allows parallel workloads to scale when emulated on multi-core hosts [17]. Focusing on making code *execution* parallel pays off, because the runtime of most DBT workloads is largely spent on code execution and not on translation.

While this observation holds for many workloads, others can show significant amounts of parallel code *translation.* This scenario is typical in parallel server workloads [12], e.g. during parallel compilation jobs that require large amounts of guest code execution with little code reuse. In these cases, achieving scalability for unified code cache translators is challenging, since scalability is limited by the contention imposed by global locks protecting code generation and translation block (TB) chaining state [12, 17, 19].

We address this challenge by starting from the design of Pico [17], which is now part of upstream QEMU. In Qelt, we modify this baseline design, in which a single lock protects both code translation/invalidation as well as code chaining, to make each vCPU thread work—in the fast path—on uncontended cache lines. As we describe next, we achieve this by distributing state across vCPUs, and combining lock-free operations with fine-grained locks that are unlikely to be contended.

Translator state and code cache. We distribute the translator's state by replicating it across the vCPU threads. We keep the baseline's single, contiguous (in virtual memory) buffer for the code cache, since doing otherwise would greatly complicate cross-ISA code generation. However, we partition this buffer so that each vCPU generates code into a separate *region.* Figure 3 illustrates the impact of region partitioning: while a monolithic cache forces writers to be serialized, a partitioned cache allows vCPUs to generate code in parallel. Partitioning can reduce the effective size of the code cache, since vCPUs generate code at different rates. However, in most practical scenarios this reduction is negligible due to adequate region sizing, which accounts for the number of vCPUs and the size of the code cache.



(a) Monolithic TB cache **(b)** Partitioned TB cache

Figure 3. With a monolithic translation code cache (a), TB execution (green TBs) can happen in parallel, yet TB generation (red) is serialized. Region partitioning (b) enables both parallel code execution *and* translation.

Program Counter (PC) TB lookups. PC TB lookups take the program counter (as a host virtual address) of some translated code and provide the corresponding TB descriptor. To serve these lookups we maintain a per-region binary search tree that tracks the beginning and end host addresses of the region's TBs. Operations on each of the trees are serialized with the same per-region lock used for writing code into the region. This has little to no impact on scalability, since PC TB lookups and TB invalidations are rare; the writer thread therefore acquires an uncontended lock, which is fast.

Physical memory map. Descriptors of guest memory pages are kept in the memory map, which is implemented as a radix tree. We modify this tree to support lock-free insertions, and rely on the fact that the tree already supports RCU for lookups [17]. A spin lock is added to each descriptor to serialize accesses to the page's list of TBs. This list is accessed when TBs are either added or removed during code translation or invalidation, respectively. Some operations (e.g. invalidating a range of virtual pages) require atomic modifications over a range of non-contiguous physical pages. To avoid deadlock we acquire the locks of all the associated page descriptors in ascending order of physical address.

TB index. We rely on QHT, Pico's scalable hash table [17], to implement scalable TB bookkeeping. Accesses to the index are used as synchronization points. For instance, if two threads are contending to insert the same TB, the first one to complete the insertion into the hash table will win the race. The other thread will realize this at insertion time, subsequently undoing prior changes (e.g. insertion into the page's list of TBs) to then use the TB translated by the other thread.

Code chaining. TBs that are linked via direct jumps are chained together during code execution by patching the generated code to directly jump across translated code, thereby increasing performance. The linking and patching requires serialization to prevent chaining to a TB that is being invalidated and to protect the list of incoming TBs. Instead of relying on a global lock, we use a per-jump spinlock and add two tagged pointers to each TB descriptor to point to its two destination TBs. The pointers, which are accessed atomically with compare-and-swap, are tagged to ensure that no jumps from invalidated TBs can be linked.

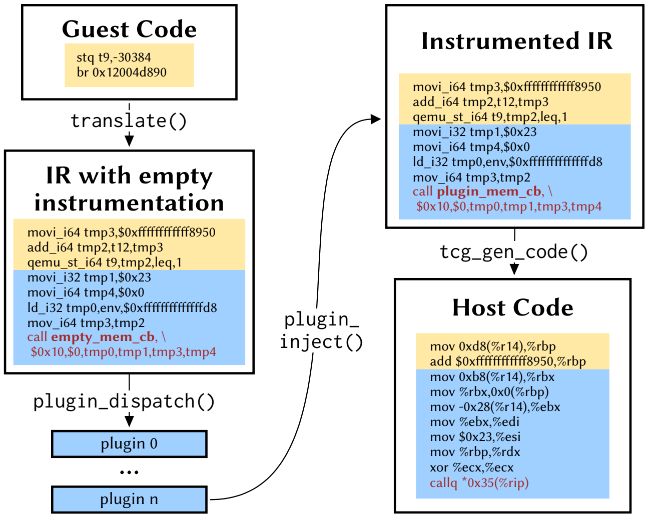
3.3 Portable Cross-ISA Instrumentation

We now describe Qelt’s technique to convert a cross-ISA DBT engine into a cross-ISA DBI tool. This technique has four main properties. First, it provides injection points at an instruction level, which is in line with state-of-the-art instrumentation tools such as Pin and DynamoRIO. Second, it is ISA-agnostic, i.e., it remains portable by only requiring modifications to the ISA-independent parts of the DBT engine. Third, it is suitable for full-system instrumentation, since it can also instrument the side-effects of emulation performed via helpers. Last, it is high performance, with support for inline callbacks and multiple event subscriptions.

ISA-agnostic instrumentation. Figure 4 shows the flow of a translation block (TB), from its creation to its translation into instrumented host code. The guest code snippet in Figure 1 is reused here; note that additional instrumentation-related code is added to both the intermediate representation (IR) and host code. In particular, this additional code implements a memory callback to a plugin.

The flow begins from a guest program counter from which code is to be executed. A single guest-to-IR translation pass is performed, and along the way, empty instrumentation is inserted. For instance, if a memory access is encountered, a callback to a placeholder “empty” memory callback is generated into the IR. Once the TB is fully formed, we dispatch it to plugins. Plugins add their instrumentation calls to the TB using the instrumentation API, which correspondingly annotates the TB’s descriptor. We then perform the *injection* pass. That is, we go through all the empty instrumentation points, and either remove them from the IR if no subscriptions to them were requested, or copy them as needed (one copy per plugin request), substituting the “empty” placeholder with the plugin-supplied callback and/or data. The flow finishes by translating the instrumented IR into host code.

Injection points. Instead of letting plugins inject instrumentation directly into the IR, we keep the IR entirely internal to the implementation, injecting during guest TB translation empty instrumentation that we can later remove if unneeded. This approach, which at first glance might seem wasteful in that it might perform unnecessary work, has several strengths:



```
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
```

```
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
```

```
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)
```

Figure 4. Example instrumentation flow of Alpha-on-x86_64 emulation. We inject empty instrumentation as we translate the guest TB. Once the TB is well-defined, we dispatch it to plugins, which annotate it with instrumentation requests. Empty instrumentation is then either removed if unneeded or replaced with the plugin’s requests. Finally, the instrumented IR is translated into host code.

- It enables the implementation of *instruction-grained* instrumentation, which lets plugins inject instrumentation for events associated to a particular instruction. For instance, a plugin can, *at translation time*, insert instrumentation before/after memory accesses associated with a particular instruction, instead of subscribing to *all* guest memory accesses and then selecting those of interest *at run-time*.
- It is ISA-agnostic, i.e., it requires no modifications to ISA-specific code. The instrumentation layer only modifies generic code, leaving instruction decoding to plugins.
- It incurs negligible cost. As we show in Section 4.5, on average the injection of empty instrumentation induces negligible overhead.

Event subscriptions. We distinguish between two event types: *regular* and *dynamic*. Dynamic events are related to guest execution (e.g. memory accesses, TBs executed), and therefore occur extremely frequently. Regular events are all others, e.g. vCPU thread starts/stops, or TBs being translated. We expand later on dynamic events, whose delivery we optimize with *inlining* and *direct callbacks*. Regular subscriptions are kept in per-event read-copy-update (RCU) [36] lists, which make callback delivery fast and scalable. RCU is a fitting tool for this purpose due to the *read-mostly* nature of the access pattern: list traversals (i.e. callbacks) strongly outnumber list insertions/removals (i.e. subscription registrations/cancellations).

Helper instrumentation. Instrumenting helpers is challenging, since at translation time we do not know what they implement or when they will execute. The magnitude of the challenge increases when we consider the amount of helpers that might be in a code base. For instance, each of the 22 target translators in QEMU uses, on average, more than a hundred helpers. Thus, the straw man solution of modifying thousands of helpers to add instrumentation-related code becomes a tedious and error-prone prospect.

We present a low-overhead approach to instrument helpers that is more practical. Our approach, which we apply to memory accesses performed by helpers, relies on the following observation. Guest memory accesses from helpers are performed via a handful common interfaces. Thus, we modify those common interfaces—about a dozen call sites in QEMU—instead of editing potentially thousands of helpers.

Most of the work is done at translation time. We start by tracking which guest instructions have emitted helpers. For each of these instructions, if plugins have subscribed to their memory accesses, we proceed in two steps. First, we allocate the subscription requests from the appropriate plugins into an array, which we track using a scalable hash table [17] so that it can be freed once the TB is invalidated. Second, we inject IR code before the guest instruction that sets *helper_mem_cb*, which is a field in the state of the vCPU that will execute the helper, to point to the subscription array. We also insert code after the instruction to clear this field.

At execution time, we rely on our modified common interfaces for accessing memory from helpers. Thus, when an instrumented helper accesses memory, the generic memory access code checks the executing vCPU’s *helper_mem_cb* field, and if set, delivers the callbacks to plugins.

Inlining. We support manual inlining of instrumentation code for *dynamic* events. Plugins can explicitly insert inline operations, which they choose via the plugin API. These operations implement typical actions needed by instrumentation code, such as setting a variable or incrementing a counter, and are independent from the IR, since the latter is internal to the translator’s implementation and therefore always subject to change. In Section 4.6 we show how inlining can increase performance for instrumentation-heavy workloads.

Direct callbacks. We treat dynamic events differently from regular ones. The reason is performance: dynamic events—such as memory accesses or TBs/instructions executed—can be generated *extremely* frequently, and therefore the overhead of instrumenting these events can easily dominate execution time. Although inlining can help mitigate this overhead, complex instrumentation (e.g. code that inserts an address into a hash table) cannot benefit from it, which brings our focus to the performance of callback delivery.

Most existing cross-ISA DBI tools deliver dynamic event callbacks using an intermediate helper that iterates over a list of subscriptions [20, 21, 25]. This is convenient from an implementation viewpoint, but introduces an unnecessary level of indirection. We eliminate this indirection by leveraging the injected empty instrumentation, which allows us to embed callbacks *directly* in the generated code. As we show in Section 4.5, direct callbacks result in better performance over delivering callbacks from an intermediate helper. They, however, complicate subscription management. To cancel a direct callback subscription, instead of just updating a list as in regular callbacks, we must re-translate the TB. This, while costly, is not a practical concern, since instruction-grained injection points virtually eliminate the need for frequent subscription cancellations from dynamic events.

```
static uint64_t mem_count;
static bool do_inline;

static void plugin_exit(plugin_id_t id, void *p)
{
  printf("mem accesses: %" PRIu64 "\n", mem_count);
}

static void vcpu_mem(unsigned int cpu_index,
  plugin_meminfo_t meminfo, uint64_t vaddr, void *udata)
{
  mem_count++;
}

static void vcpu_tb_translate(plugin_id_t id,
  unsigned int cpu_index, struct plugin_tb *tb)
{
  size_t n = plugin_tb_n_insns(tb);
  size_t i;
  for (i = 0; i < n; i++) {
    struct plugin_insn *insn = plugin_tb_get_insn(tb, i);
    if (do_inline) {
      plugin_register_vcpu_mem_inline__after(
        insn, PLUGIN_INLINE_ADD_U64, &mem_count, 1);
    } else {
      plugin_register_vcpu_mem_cb__after(
        insn, vcpu_mem, PLUGIN_CB_NO_REGS, NULL);
    }
  }
}

int plugin_install(plugin_id_t id, int argc, char **argv)
{
  if (argc && strcmp(argv[0], "inline") == 0)
    do_inline = true;
  plugin_register_vcpu_tb_trans_cb(id, vcpu_tb_translate);
  plugin_register_atexit_cb(id, plugin_exit, NULL);
  return 0;
}
```

Figure 5. Example Qelt plugin to count memory accesses either via a callback or by inlining the count.

An example Qelt plugin. Figure 5 shows an example Qelt instrumentation plugin that counts guest memory accesses. Execution begins in the plugin at load time, with Qelt calling the `plugin_install` function. The plugin subscribes to two events: TB translations and guest exit—i.e., termination of a user-program or shutdown of a full-system guest.

Instrumentation of guest code occurs in `vcpu_tb_translate`. For each instruction in a TB, instrumentation is added after the instruction’s memory accesses, if any. Depending on `do_inline`’s value, instrumentation is either via a direct callback to `vcpu_mem` or through an inline increment to the counter, `mem_count`. Note that to keep the example simple, the counter’s increment is not implemented with an atomic operation, which could result in missed counts when instrumenting parallel guests.

We conclude by discussing three points that are not obvious from the example. First, the API exposes no ISA-specific knowledge. For example, instructions are treated as opaque objects; this requires plugins that need instruction information to rely on an external disassembler, but as we show in Section 4.5 this has negligible overhead. Second, vCPU registers can be queried from callbacks. The example does not use this feature, and thus it disables register copying at callback time with the `CB_NO_REGS` flag. Third, instead of supporting user-defined functions like Pin [33] or QTrace [49] do, we attach a *user data* pointer (`udata`) to direct callbacks, which achieves the flexibility of user-defined functions while being more portable and simpler to implement.

3.4 Additional DBT Optimizations

We now describe DBT optimizations implemented in Qelt that are derived from those in state-of-the-art DBT engines.

Cross-ISA TLB Emulation. Guest TLB emulation in cross-ISA full-system emulators is a large contributor to performance overhead. The "softMMU", as it is called in QEMU [6], is a table kept in software that maps guest to host virtual addresses. SoftMMU overhead comes from three sources, which we list in order of importance. First, non-compulsory misses in the TLB result in guest page faults, which take hundreds of host instructions to execute. Second, even if the TLB miss rate is low, hits still incur non-negligible latency, since each guest memory access is translated into several host instructions which index and compare the contents of the TLB against the appropriate portion of the guest virtual address. And third, clearing the TLB on a flush can also incur non-trivial overhead due to frequently-occurring flushes. It is for this reason that QEMU has a small, static TLB size.

Tong et al. [48] present a detailed study in which they evaluate different options to mitigate softMMU overhead. One of these options is to resize the TLB depending on the workload's requirements. They resize the TLB only during flushes, since doing it at any other time would require rehashing the table, which is expensive. They propose a simple resizing policy: if the TLB use rate at flush time is above a certain upper threshold (e.g. 70%), double the TLB size; if the rate is below a certain lower threshold (e.g. 30%), halve it. Note that the upper threshold should not ever be too close to 100%, for otherwise we are at risk of incurring a large amount of conflict misses, given that the softMMU is direct-mapped to keep TLB lookup latency low. In addition, computing the table index dynamically incurs a slight lookup latency increase. The rationale, however, is that the reduced number of misses is likely to amortize this additional cost.

We observe that this policy can lead to overly aggressive resizing. This can be illustrated with two alternating processes, of which one is memory-hungry and the other uses little memory. With this policy, when the guest schedules out the memory-hungry process, the TLB size doubles, yet the next process will not make much use of it, which will induce a downsize. This results in a sequence of TLB size doubling/halving, which neither process can benefit from.

We improve upon this policy by incorporating history into it. We track the maximum use rate in the most recent past (e.g. a *history window* of 100ms), and resize based on that maximum observed use rate. The rationale is that if a memory-hungry process has been recently scheduled, it is likely that it will be scheduled again in the near future. This can result in an oversized TLB for processes that are scheduled next, but this cost is likely to be offset by an eventual reduction in overall misses. In other words, we incorporate some history into the policy to perform the same aggressive upsizing, yet downsize more conservatively. In Section 4.2 we compare these two policies, with 70%-30% thresholds and a history window of 100ms, against QEMU's static TLB.

Indirect branches in DBT. Speeding up the handling of indirect branches is commonly done by compiling frequently-executed sequences of translation blocks into single-entry, multi-exit *traces* [5]. Unfortunately, the applicability of trace compilation to full-system emulators is limited; even for direct jumps, the optimization is constrained to work only across same-page TBs, for otherwise the validity of the jump target's virtual address cannot be guaranteed without querying at run-time the softMMU. An approach better suited for full-system emulators is the use of caching [44], which is demonstrated on QEMU by Hong et al. [26]. They add a small cache to each vCPU thread to track cross-page and indirect branch targets, and then modify the target code to inline cache lookups and avoid most costly exits to the dispatcher. In Qelt we follow an approach similar to theirs, abstracting these lookups by adding an operation (*"lookup and goto ptr"*) to the IR, thereby minimizing the amount of target and host-specific code needed to support the optimization.

4 Evaluation

4.1 Setup

Host. We run all experiments on a host machine with two 2.6GHz, 16-core Intel Xeon Gold 6142 processors, for a total of 32 cores. The machine has 384GB of RAM, and runs Ubuntu 18.04 with Linux kernel v4.15.0. We compile all source code with GCC v8.2.0 with `-O2` flags.

Workloads. We measure single-threaded performance with SPEC06's *test* set, except for libquantum, xalancbmk, gamess, soplex and calculix. For these workloads we use the *train* set, since *test* is too lightweight (e.g. *libquantum* runs natively under 0.02s) for us to draw meaningful conclusions when running them under different DBT engines.

We run all experiments multiple times. We report the measured mean as well as error bars or shaded regions (for bar charts or line plots, respectively) representing the 95% confidence interval around the mean.

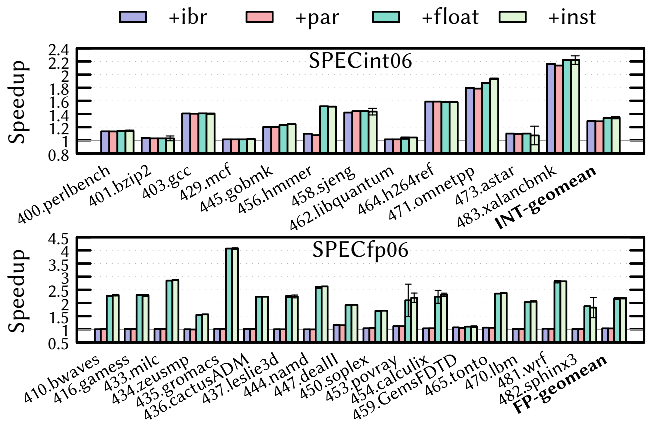


Figure 6. Cumulative speedup of Qelt’s techniques over QEMU for user-mode x86_64 SPEC06.

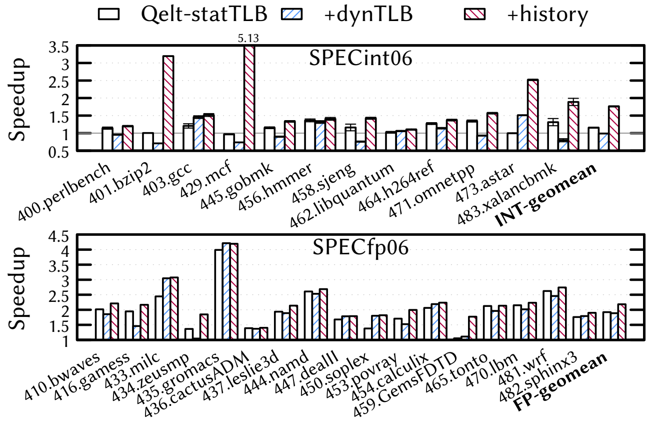


Figure 7. Cumulative speedup of Qelt’s techniques over QEMU for full-system x86_64 SPEC06.

Guest ISA. We use x86_64 guest workloads, which allows us to compare against existing DBI tools (all of them support x86_64) and to benchmark against native runs on the host machine, including full-system guests virtualized with KVM.

QEMU baseline. Our QEMU baseline is derived from QEMU v3.1.0. Given that several of Qelt’s techniques are already part of QEMU v3.1.0, our baseline (hereafter *QEMU*) is the result of reverting their corresponding changes from v3.1.0.

4.2 Performance Impact of Qelt’s Techniques

We begin our evaluation by characterizing the performance impact of implementing Qelt’s techniques in sequence on top of QEMU when running single-threaded guest workloads.

Figure 6 shows the resulting speedup for user-mode x86_64 SPEC06. Qelt’s indirect branch optimizations (+ibr, Section 3.4) yield an average 29% performance gain for integer workloads. Qelt’s parallel code generation (+par, Section 3.2) and instrumentation support (+inst, Section 3.3) show negligible performance impact. Last, Qelt’s FP emulation improvements (+float, Section 3.1) shows the largest improvement: SPECfp06’s performance increases more than 2× on average.

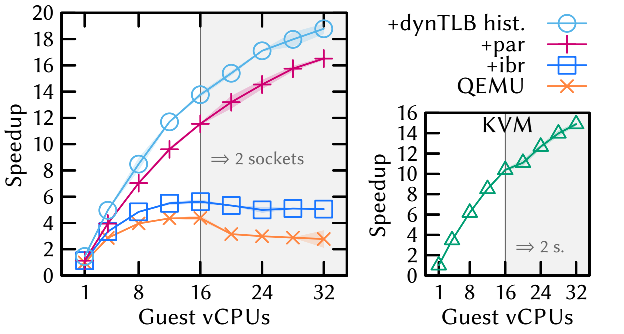


Figure 8. Cumulative Qelt speedup over 1-vCPU QEMU for parallel compilation of Linux kernel modules in an x86_64 VM. On the right, KVM scalability for the same workload.

Figure 7 shows Qelt’s speedup over QEMU for full-system x86_64 SPEC06. The techniques presented in Figure 6 are combined as Qelt-statTLB. Their resulting speedup is lower than in user-mode due to the overhead of full-system mode’s softMMU. Adding a TLB resizing policy based solely on the TLB use rate at flush time (+dynTLB, as described in [48]) results in a slowdown on average, since the system also runs system processes with low memory demands. Qelt’s policy (+history, Section 3.4) bases its resizing decisions on the use rate over the recent past, which leads to overall mean speedups of 1.76× and 2.18× for integer and FP workloads, respectively.

4.3 Scalable Dynamic Binary Translation

We now evaluate Qelt’s scalability with a workload that requires large amounts of parallel DBT. For this we build 189 Linux v4.19.1 kernel modules with `make -j N` (where N is the number of guest cores) inside an x86_64 virtual machine (VM) running Ubuntu 18.04. Figure 8 (left plot) shows the results, which are normalized over those of QEMU with a single guest core. QEMU shows poor scalability, with a maximum speedup of 4× at 16 cores. Qelt’s indirect branch optimizations (+ibr) slightly improve performance, but do not address the underlying scalability bottleneck. Qelt’s parallel translator (+par, Section 3.2) brings scalability in line with that of KVM (right plot), for a maximum speedup above 16× at 32 cores. Scalability is further improved with Qelt’s dynamic TLB resizing (+dynTLB hist.), which brings the overall speedup up to 18.78×.

4.4 Fast FP Emulation using the Host FPU

Qelt accelerates the following operations, in both single and double precision: addition, subtraction, multiplication, division, square root, comparison and fused multiply-add (FMA). We validate our implementation against real hardware (ppc64, Aarch64 and x86_64 hosts) as well as against Berkeley’s Testfloat v3e [24] and IEEE-754-compliant test patterns from IBM’s FPgen [7].

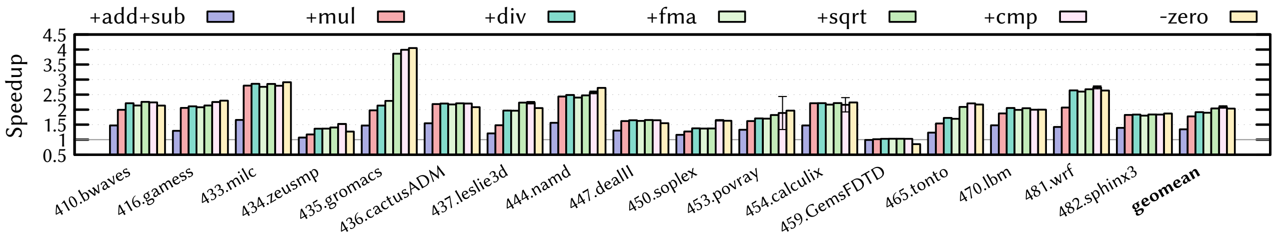


Figure 9. Cumulative speedup over QEMU of accelerating the emulation FP instructions with Qelt for user-mode x86_64 SPECfp06. The -zero results show the impact of removing Qelt's zero-input optimization.

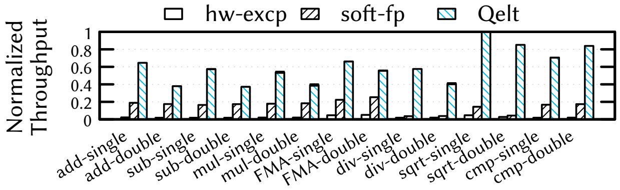


Figure 10. FP microbenchmark results. Throughput is normalized over that of an ideal native run.

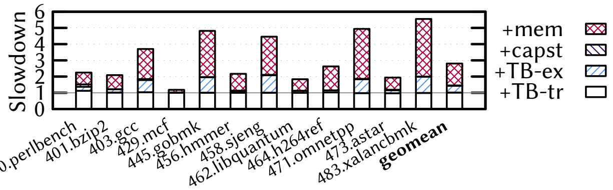


Figure 11. Impact of increasing instrumentation on user-mode x86_64 SPECint06.

Figure 9 shows the speedup of accelerating FP emulation with Qelt for SPECfp06 in user-mode x86_64. The results shown are cumulative, which reveals the impact of accelerating each group of FP instructions separately. Note that QEMU's FP operations are implemented as a C library that is shared by all ISA translators. Thus, while the final SPECfp06 speedup is similar across all ISAs, the broken-down speedups are specific to each translator. For instance, the x86_64 translator (shown here) is sensitive to changes to both addition and multiplication's performance, whereas Aarch64's (results not shown) is most sensitive to FMA. The last set of results (-zero) shows the effect of removing the zero-input optimization in Figure 2. Its removal hurts average performance, since some benchmarks frequently execute zero-input FP operations (e.g., cactusADM, GemsFDTD).

FP Microbenchmark. The above results depend on the distribution of FP instructions in SPECfp06. To better understand Qelt's impact on FP emulation, we wrote a microbenchmark that feeds random *normal* FP operations to the FP emulation library. Figure 10 shows the resulting throughput, normalized over that of an ideal, incorrect run on the host, i.e. without any checks on either the result or FP flags.

The first set of results (hw-excp) in Figure 10 corresponds to a naïve implementation that, as described in Section 3.1, for each FP instruction first clears the host's FP flags (with `feclearexcept(3)`), then executes the FP operation on the host, and finally checks the host FP flags (`fetestexcept(3)`). This approach has poor performance, even when compared against QEMU's soft-float implementation (soft-fp). We have reproduced this on other machines as well, which suggests that FPUs are optimized for fast, overlapping execution of FP instructions, and not for frequent FP flag checks.

Qelt improves performance over soft-float, with speedups ranging from 2.16× (mul-double) to 19.84× (sqrt-double). The performance gap between Qelt and ideal FP performance quantifies the cost of correctness; recall from Figure 2 that we have to perform checks on the input as well as on the computed output (to detect under/overflow). The latter checks, however, are not needed for comparison and square root (a non-negative normal-or-zero square root cannot under/overflow), which explains the narrow performance gap between them and the ideal implementation.

4.5 Instrumentation

We now characterize the performance of Qelt's instrumentation layer. We first analyze the overhead of typical instrumentation plugins, and then evaluate the impact of different *direct callback* implementations.

Overhead. Figure 11 shows Qelt's slowdown over the baseline for typical instrumentation plugins, broken down per instrumented event. Subscribing to TB translation events (+TB-tr) incurs negligible average overhead (1.1%), with perlbench showing the maximum overhead (12%) since it is the workload that executes the most guest code.

Subscribing to TB execution callbacks (+TB-ex) has significant overhead (mean 41%, maximum 107% for *sjeng*). The overhead is caused by the high frequency of guest TB execution and, therefore, TB execution callbacks. It is for this reason that instrumentation is preferably done at translation time whenever possible. The "capstone" plugin (+capst) is an example of translation-time processing that mimics what an architectural simulator would do during decode: it disassembles each translated TB using Capstone [42] and then allocates a per-TB descriptor to be passed to the TB execution callback. This translation-time processing incurs negligible additional overhead (mean 2.6%), which supports our decision to not export via the plugin API any interfaces with ISA-specific knowledge of the target's instructions.

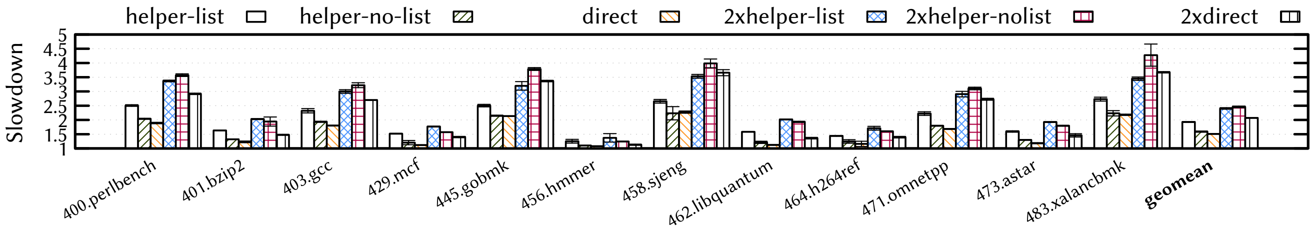


Figure 12. Slowdown of user-mode x86_64 SPECint06 for helper-based and direct callbacks.

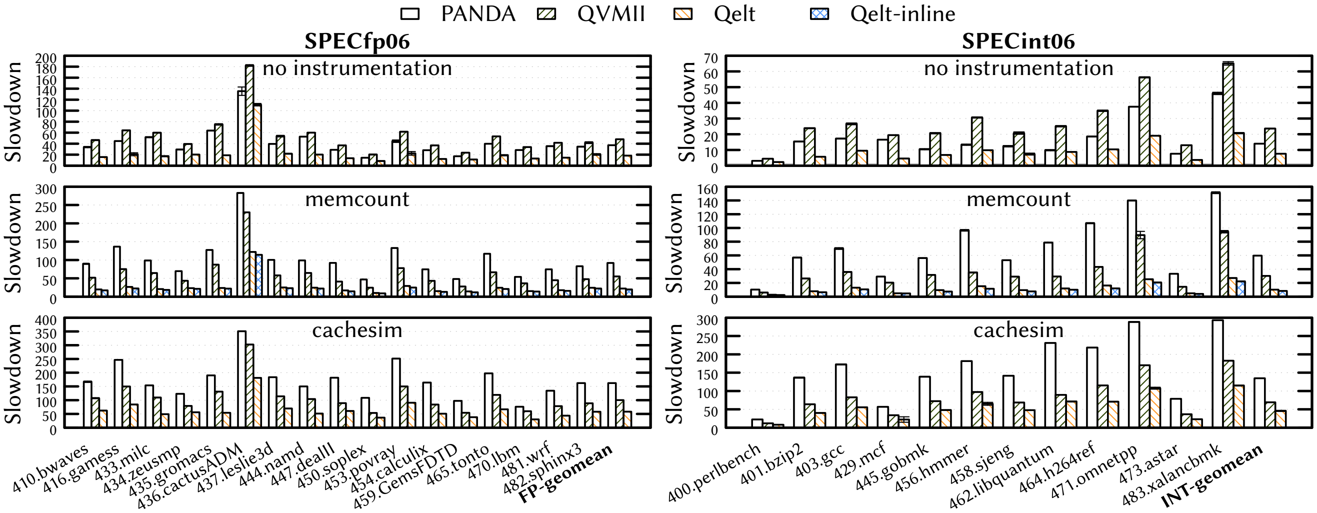


Figure 13. Slowdown over KVM execution for PANDA, QVMII and Qelt for full-system emulation of x86_64 SPEC06.

Memory callbacks (+mem) incur large overhead due to their high frequency. Fortunately, this cost can be mitigated with inlining, as shown in Section 4.6.

Direct callbacks. We now discuss the impact of instrumenting *dynamic* (i.e. high-frequency) events. Figure 12 compares the instrumentation of a dynamic event (TB execution) for three different implementations and either one or two (denoted with the 2x prefix) plugin subscriptions. The helper-list implementation uses a *helper* function from which callbacks are dispatched by iterating over a list of subscribers. When the event has a single subscriber, it pays off to avoid the list altogether, which improves performance—as helper-nolist shows—due to increased cache locality. An additional improvement is obtained by using direct callbacks (direct), which incur one less function call (i.e., the helper) per event.

With two subscribers, helper-list performs three function calls per event, for helper-nolist's four. However, the former's subsequent gains are canceled out by iterating over the subscribers' list (2.40× vs. 2.45× mean slowdown), which has poor data cache locality. Using direct callbacks outperforms them both (2.07× slowdown), since it incurs one function call per subscriber and has optimal data cache locality.

4.6 DBI Tool Comparison

We conclude our evaluation by comparing Qelt against the state of the art in full-system and user-mode DBI tools.

Full-System DBI. We compare Qelt against PANDA [20] (version d886146, Aug 3 2018) and QVMII [21] (dc7d35d, Jul 18 2018). We also considered other QEMU-derived tools such as QTrace [49], PEMU [54] and DECAF [25], but discarded them due to their slow baseline emulation performance (we measured QTrace to be on average 11.3× slower than Qelt for SPEC06int, and [54] reports a 4.33× average slowdown of PEMU over QEMU) or lack of support for x86_64 (DECAF). Figure 13 shows the resulting slowdown over using KVM to virtualize an x86_64 VM running SPEC06. For both integer and FP workloads (top row), Qelt is the fastest emulator, with PANDA coming second with performance similar to that of baseline QEMU (PANDA's fork point is close to our QEMU baseline, which we evaluated in Figure 7). QVMII is the slowest, since by default it instruments all memory accesses in case any plugins subscribe to them.

Instrumenting the execution of memory accesses with a counter increment (memcount) shows the differences in instrumentation overhead. PANDA lags behind because for simplicity of the implementation it disables key QEMU optimizations (translation block chaining and TCG softMMU lookups [6], respectively). Qelt is 3×/2.5× faster than QVMII for integer/FP workloads, with a slight improvement with inlining, a feature not supported by the other tools. The gap between QVMII and Qelt is explained by their different baseline emulation performance as well as Qelt's use of direct callbacks instead of QVMII's helper-based approach.

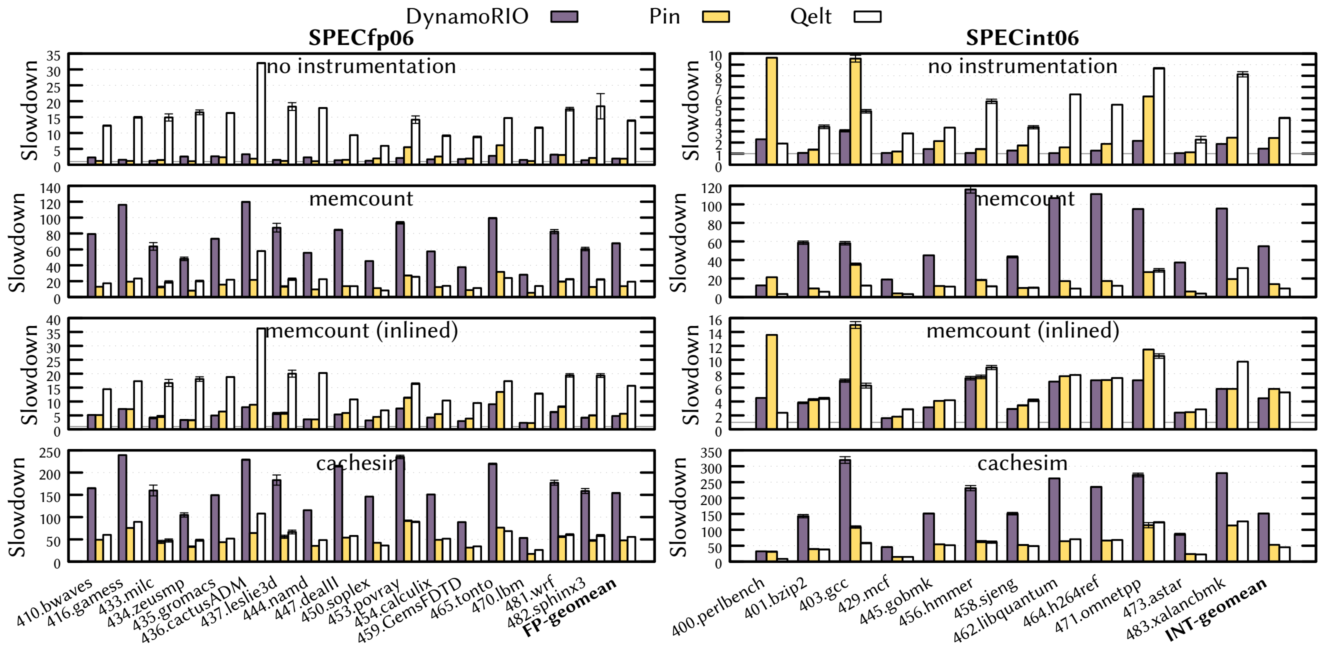


Figure 14. Slowdown over native execution for DynamoRIO, Pin and Qelt for user-mode x86_64 SPEC06.

We then perform heavy instrumentation (cachesim), similar to what an architectural simulator would do. We simulate L1 instruction and data caches, without a directory and with an LRU set eviction policy. This is implemented by instrumenting all memory accesses, as well as simulating the corresponding instruction cache accesses when a basic block executes. Instrumentation now dominates execution time for SPECint06, making QVMII 1.53× slower than Qelt. This performance gap reduction vs. baseline emulation is less pronounced for SPECfp06; Qelt is 1.72× faster than QVMII thanks to Qelt’s improved FP performance.

User-Mode DBI. Figure 14 compares Qelt against Pin [33] (v3.7-97619, May 8 2018) and DynamoRIO (v7.0.17735-0, Jul 30 2018). These tools are not cross-ISA, which to a large extent explains their superior performance over Qelt for pure emulation (top row). DynamoRIO and Pin stay below or close to 2× slowdown over native, while Qelt is 4.20×/13.89× slower than native for integer/FP. Note that despite Qelt’s FP improvement over QEMU, it still requires a *helper* function call to the FP library on every executed guest FP instruction, which explains the large SPECfp06 gap vs. Pin/DynamoRIO.

Inlining is key to DynamoRIO’s performance when instrumenting frequent events. This is a consequence of DynamoRIO’s flexibility, since it allows plugin developers to make arbitrary changes to the guest code stream. Unfortunately, when inlining cannot be performed (either by disabling it or when a function is too complex to be inlined), constructing a callback involves a significant amount of work: “switching to a safe stack, saving all registers, materializing the arguments, and jumping to the callback” [31]. On the other hand, tools like Pin in its “classic” mode (which we use) or Qelt do not allow arbitrary guest code modifications, and therefore can efficiently insert a call/trampoline to a plugin. This explains DynamoRIO’s large overhead in memcount, whereas it is the fastest tool for inline memcount.

For memcount, Pin is in most cases faster than Qelt, although Pin’s well-known performance issues with large instruction footprints (e.g., perlbench, gcc) [33] bring Pin’s SPECint06 mean slowdown slightly above Qelt’s for both out-of-line and inline memcount’s. For SPECfp06 memcount, Qelt is only slightly slower than Pin, since the frequent callbacks dominate execution time. However, with inlining Qelt is slower than Pin/DynamoRIO, because of its slower FP emulation.

The callbacks in cachesim are too complex to be inlined, which explains DynamoRIO’s large slowdown. On average, Qelt’s cachesim performance is similar to Pin’s. This is due to cachesim’s substantial overhead, which dominates over the difference in emulation speed between Qelt and Pin.

5 Related Work

Instrumentation. Valgrind [39], DynamoRIO [11] and Pin are popular same-ISA user-mode DBI tools, with Pin and DynamoRIO having similarly high performance [33]. At the other end of the performance spectrum we encounter architectural simulators such as gem5 [8], Simics [35], MARSS-x86 [41] and Manifold [50], which have detailed timing models for full-system simulation.

Machine emulators are not concerned with timing and therefore are better suited for instrumentation. Among machine emulators, QEMU [6] is the most popular, due to its high performance, wide cross-ISA support and mature, open-source code base. *Upstream* QEMU does not yet provide instrumentation capabilities, which has sparked the development of multiple QEMU-based instrumentation tools.

A popular QEMU fork is the Unicorn framework [40], which adds an instrumentation layer around QEMU's DBT engine. Unfortunately, Unicorn is not an instrumentation tool: it cannot emulate binaries or work as a machine emulator, since those features were removed when forking QEMU. Unicorn is therefore suitable for being embedded into other applications, or for reverse engineering purposes. PEMU [54] and QTrace [49] implement a rich instrumentation and introspection interface for x86 guests, although they incur high overhead. TEMU [46], PANDA [20] and DECAF [25], of which the latter two are cross-ISA, implement security-oriented features such as taint propagation at the expense of large performance overhead. Of the above QEMU-based tools, only QTrace allows for instruction-level instrumentation injection. QTrace achieves this by shadowing guest memory and registers, which might be a contributor to its low performance. In contrast, our injection model is simpler, works entirely at translation time, can instrument helpers and has negligible performance overhead.

Similarly to our work, DBILL [34], PIRATE [53] and PANDA optionally instrument helpers, although by leveraging the LLVM compiler to convert the original helper code into an instrumentable IR, as originally developed by Chipounov et al. [15]. This is a flexible approach, but incurs high translation overhead and complexity. QEMU-based instrumentation was combined with debugging and introspection capabilities in QVMII [21], which optionally provides deterministic record-and-replay execution at the expense of performance.

Fast FP Emulation. The idea of leveraging otherwise unused hardware from the host to improve the correctness and/or performance of DBT-based emulation is not novel. For instance, Pico [17] uses the host's hardware transactional memory extensions to emulate load-locked/store-conditional pairs on the host. Similarly to our work, Guo et al. [23] leverage the host FPU to emulate guest FP instructions. They, however, employ a considerable amount of soft-float operations in order to handle all possible corner cases (e.g. due to different operands, flags and rounding modes). Our approach puts greater emphasis on performance: it identifies a *fast path* (or *common case*) that can be accelerated with a minimum amount of auxiliary code, and defers all other (unlikely) cases to a *slow path* entirely implemented in soft-float.

Scalable Dynamic Binary Translation. Zhu et al. [55] and Böhm et al. [10] present systems that combine an interpreter with a DBT engine, which allows concurrent execution of code while worker threads translate traces of *hot* code. A similar approach was applied to QEMU by HQEMU [27], which leverages multi-core hosts to improve the quality of hot code by compiling it with LLVM in worker threads.

The use of worker threads is orthogonal to the consistency of the code cache. As discussed in Section 3.2, scaling parallel code translation in full-system emulators with shared code caches is challenging. A common approach is thus to give up scalability by using coarse-grained locks (e.g., QEMU [17], PQEMU [19]). Full-system translators with thread-private caches (e.g., QSim [30], Parallel Embra [32], COREMU [52]) can trivially scale during parallel code translation, yet can result in prohibitive memory usage [12]. Qelt's DBT design is, to our knowledge, the first DBT engine for full-system emulators to scale during parallel code generation and chaining while maintaining a shared code cache.

Cross-ISA TLB Emulation. Tong et al. [48] present an extensive study on how QEMU's softMMU can be optimized. Among several enhancements, they propose dynamic resizing of the softMMU, an idea that we extend with the consideration of TLB use rates in the recent past, which yields significant speedups for memory-hungry workloads.

Alternative approaches leverage the host hardware. They range from keeping shadow page tables using the host's virtual memory support [51], to virtualized page tables [14, 22] to deploying a hypervisor under which to run the emulator [47]. Unfortunately, these approaches require the host's virtual address length to be greater than the guest's. It is unclear whether this limitation can be overcome without sacrificing performance.

6 Conclusions

We presented two novel techniques to increase cross-ISA DBT emulation performance: fast FP emulation leveraging the host FPU and scalable DBT generation and chaining for emulating multi-core guests. We also introduced a novel ISA-agnostic instrumentation layer, which can be used to convert cross-ISA DBT engines into cross-ISA DBI tools.

We combined these techniques together with further DBT optimizations to build Qelt, a cross-ISA machine emulator and DBI tool that outperforms the state of the art in both cross-ISA emulators and DBI tools. Further, Qelt can match the performance of Pin, a state-of-the-art, same-ISA DBI tool, when performing complex instrumentation such as cache simulation.

Qelt's implementation is based on the open-source QEMU emulator. At the time of publication, all of Qelt's contributions have been merged into *mainline* QEMU, except the instrumentation layer, which is under review.

7 Acknowledgments

We thank Richard Henderson, Alex Bennée and the rest of the QEMU developer community for their invaluable help. We also thank the anonymous reviewers for their feedback. This work was supported in part by the National Science Foundation (A#: 1527821 and 1764000). Cloudlab [43] provided infrastructure to run our experiments.

References

[1] [n. d.]. Chromium: Running unittests via QEMU. https://www.chromium.org/chromium-os/testing/qemu-unittests. Accessed: 2018-12-19.

[2] 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70.

[3] Gautam Altekar and Ion Stoica. 2009. ODR: Output-deterministic Replay for Multicore Debugging. In *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 193–206.

[4] ARM ARM. 2012. Architecture Reference Manual. ARMv7-A and ARMv7-R Edition. *ARM DDI C* 406 (2012).

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 1–12.

[6] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proc. of the USENIX Annual Technical Conference (ATC)*. 41–46.

[7] E. Bin, R. Emek, G. Shurek, and A. Ziv. 2002. Using a constraint satisfaction formulation and solution techniques for random test program generation. *IBM Systems Journal* 41, 3 (2002), 386–402.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[9] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. 2013. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture (HPCA)*. 1–12.

[10] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. 2011. Generalized Just-in-time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 74–85.

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

[12] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. 2006. Thread-shared software code caches. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*. 28–38.

[13] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. Article 52, 12 pages.

[14] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Efficient memory virtualization for Cross-ISA system mode emulation. In *ACM SIGPLAN Notices*, Vol. 49. 117–128.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 265–278.

[16] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*. 196–206.

[17] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA Machine Emulation for Multicores. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*. 210–220.

[18] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 85–96.

[19] J. H. Ding, P. C. Chang, W. C. Hsu, and Y. C. Chung. 2011. PQEMU: A Parallel System Emulator Based on QEMU. In *Proc. of the Intl. Conf. on Parallel and Distributed Systems (ICPADS)*. 276–283.

[20] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proc. of Program Protection and Reverse Engineering Workshop*. Article 4, 11 pages.

[21] Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. 2017. QEMU-based framework for non-intrusive virtual machine instrumentation and introspection. In *Proc. of the Joint Meeting on Foundations of Software Engineering and the ACM SIGSOFT Symp. on the Foundations of Software Engineering*. 944–948.

[22] Antoine Faravelon, Olivier Gruber, and Frédéric Pétrot. 2017. Optimizing Memory Access Performance Using Hardware Assisted Virtualization in Retargetable Dynamic Binary Translation. In *2017 Euromicro Conference on Digital System Design (DSD)*. 40–46.

[23] Yu-Chuan Guo, Wuu Yang, Jiunn-Yeu Chen, and Jenq-Kuen Lee. 2016. Translating the ARM Neon and VFP Instructions in a Binary Translator. *Softw. Pract. Exper.* 46, 12 (Dec. 2016), 1591–1615.

[24] John R. Hauser. accessed Feb. 24, 2019. The softfloat and testfloat packages. http://www.jhauser.us/arithmetic/

[25] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*. 248–258.

[26] Ding-Yong Hong, Chun-Chen Hsu, Cheng-Yi Chou, Wei-Chung Hsu, Pangfeng Liu, and Jan-Jan Wu. 2015. Optimizing Control Transfer and Memory Virtualization in Full System Emulators. *ACM Trans. on Architecture and Code Optimization (TACO)* 12, 4, Article 47 (2015), 24 pages.

[27] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: A Multi-threaded and Retargetable Dynamic Binary Translator on Multicores. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*. 104–113.

[28] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. 2008. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proc. of Workshop on Modeling, Benchmarking and Simulation (MoBS)*. 28–36.

[29] Angelos D Keromytis, Roxana Geambasu, Simha Sethumadhavan, Salvatore J Stolfo, Junfeng Yang, Azzedine Benameur, Marc Dacier, Matthew Elder, Darrell Kienzle, and Angelos Stavrou. 2012. The meerkats cloud security architecture. In *Intl. Conf. Distributed Computing Systems (ICDCSW) – Workshops*. 446–450.

[30] Chad D. Kersey, Arun Rodrigues, and Sudhakar Yalamanchili. 2012. A Universal Parallel Front-end for Execution Driven Microarchitecture Simulation. In *Proc. of Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*. 25–32.

[31] Reid Kleckner. 2011. *Optimization of naïve dynamic binary instrumentation Tools*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[32] Robert E. Lantz. 2008. Fast functional simulation with parallel Embra. In *Proc. of Workshop on Modeling, Benchmarking and Simulation (MoBS)*.

[33] Chi-Keung Luk et al. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 190–200.

[34] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. DBILL: An Efficient and Retargetable Dynamic Binary Instrumentation Framework Using LLVM Backend. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*. 141–152.

[35] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[36] Paul E. McKenney and John D. Slingwine. 1998. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*. 509–518.

[37] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture (HPCA)*. 1–12.

[38] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*. 284–295.

[39] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Notices*, Vol. 42. 89–100.

[40] Anh Quynh Nguyen and Hoang Vu Dang. 2015. Unicorn: Next Generation CPU Emulator Framework. In *Black Hat USA*.

[41] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSSx86: A Full System Simulator for x86 CPUs. In *Proc. of the Design Automation Conference (DAC)*.

[42] Nguyen Anh Quynh. 2014. Capstone: Next-gen disassembly framework. *Black Hat USA* (2014).

[43] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[44] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa. 2004. Overhead reduction techniques for software dynamic translation. In *Proc. of the Intl. Parallel and Distributed Processing Symposium (IPDPS)*. 200.

[45] Yakun Sophia Shao and David Brooks. 2013. ISA-independent workload characterization and its implications for specialized architectures. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*. 245–255.

[46] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Proc. of the Intl. Conf. on Information Systems Security (ICISS)*. 1–25.

[47] Tom Spink, Harry Wagstaff, and Björn Franke. 2016. Hardware-Accelerated Cross-Architecture Full-System Virtualization. *ACM Trans. on Architecture and Code Optimization (TACO)* 13, 4 (2016), 36.

[48] Xin Tong, Toshihiko Koju, Motohiro Kawahito, and Andreas Moshovos. 2015. Optimizing memory translation emulation in full system emulators. *ACM Trans. on Architecture and Code Optimization (TACO)* 11, 4 (2015), 60.

[49] X. Tong and A. Moshovos. 2015. QTrace: a framework for customizable full system instrumentation. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*. 245–255.

[50] Jun Wang, Jesse Beu, Rishiraj Bheda, Tom Conte, Zhenjiang Dong, Chad Kersey, Mitchelle Rasquinha, George Riley, William Song, He Xiao, et al. 2014. Manifold: A parallel simulation framework for multicore systems. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*. 106–115.

[51] Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, Wei-Chung Hsu, Bin Li, and Yong Guan. 2015. HSPT: Practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines. In *ACM SIGPLAN Notices*, Vol. 50. 53–64.

[52] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. 2011. COREMU: A Scalable and Portable Parallel Full-system Emulator. In *Proc. of Principles and Practice of Parallel Programming (PPoPP)*. 213–222.

[53] Ryan Whelan, Tim Leek, and David Kaeli. 2013. Architecture-independent dynamic information flow tracking. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*. 144–163.

[54] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. 2015. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*. 147–160.

[55] X. Zhu, J. D'Errico, and W. Qin. 2006. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *Proc. of the Intl. Symp. on Hardware/Software Codesign ans System Synthesis (CODES+ISSS)*. 193–198.