A Retargetable System-Level DBT Hypervisor

Tom Spink, Harry Wagstaff, and Björn Franke, *University of Edinburgh*

https://www.usenix.org/conference/atc19/presentation/spink

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

A Retargetable System-Level DBT Hypervisor

Tom Spink
University of Edinburgh

Harry Wagstaff
University of Edinburgh

Björn Franke
University of Edinburgh

Abstract

System-level Dynamic Binary Translation (DBT) provides the capability to boot an Operating System (OS) and execute programs compiled for an Instruction Set Architecture (ISA) different to that of the host machine. Due to their performance-critical nature, system-level DBT frameworks are typically hand-coded and heavily optimized, both for their guest and host architectures. While this results in good performance of the DBT system, engineering costs for supporting a new, or extending an existing architecture are high. In this paper we develop a novel, retargetable DBT hypervisor, which includes guest specific modules generated from high-level guest machine specifications. Our system simplifies retargeting of the DBT, but it also delivers performance levels in excess of existing manually created DBT solutions. We achieve this by combining offline and online optimizations, and exploiting the freedom of a Just-in-time (JIT) compiler operating in a bare-metal environment provided by a Virtual Machine (VM) hypervisor. We evaluate our DBT using both targeted micro-benchmarks as well as standard application benchmarks, and we demonstrate its ability to outperform the de-facto standard QEMU DBT system. Our system delivers an average speedup of 2.21× over QEMU across SPEC CPU2006 integer benchmarks running in a full-system Linux OS environment, compiled for the 64-bit ARMv8-A ISA and hosted on an x86-64 platform. For floating-point applications the speedup is even higher, reaching 6.49× on average. We demonstrate that our system-level DBT system significantly reduces the effort required to support a new ISA, while delivering outstanding performance.

1 Introduction

System-level DBT is a widely used technology that comes in many disguises: it powers the Android Open Source Project (AOSP) Emulator for mobile app development, provides backwards compatibility for games consoles [52], implements sandbox environments for hostile program analysis [41] and enables low-power processor implementations for popular ISAs [17]. All these applications require a complete and faithful, yet efficient implementation of a guest architecture, including privileged instructions and implementation-defined behaviors, architectural registers, virtual memory, memory-mapped I/O, and accurate exception and interrupt semantics.

The broad range of applications has driven an equally broad range of system-level DBT implementations, ranging from manually retargetable open-source solutions such as QEMU [4] to highly specialized and hardware supported approaches designed for specific platforms, e.g. Transmeta Crusoe [17]. As a de-facto industry standard QEMU supports all major platforms and ISAs, however, retargeting of QEMU to a new guest architecture requires deep knowledge of its integrated Tiny Code Generator (TCG) as it involves manual implementation of guest instruction behaviors. Consequently, retargeting is time-consuming and error-prone: e.g. the official QEMU commit logs contain more than 90 entries to bugfixes related to its ARM model alone.

In this paper we present **Captive**, our novel system-level DBT hypervisor, where users are relieved of low-level implementation effort for retargeting. Instead users provide high-level architecture specifications similar to those provided by processor vendors in their ISA manuals. In an offline stage architecture specifications are processed, before an architecture-specific module for the online run-time is generated. Captive applies aggressive optimizations: it combines the offline optimizations of the architecture model with online optimizations performed within the generated JIT compiler, thus reducing the compilation overheads while providing high code quality. Furthermore, Captive operates in a virtual bare-metal environment provided by a VM hypervisor, which enables us to fully exploit the underlying host architecture, especially its system related and privileged features not accessible to other DBT systems operating as user processes.

The envisaged use of Captive is to provide software developers with early access to new platforms, possibly hosted in a cloud environment. To facilitate this goal, ease of retargetability is as important as delivering performance levels sufficient

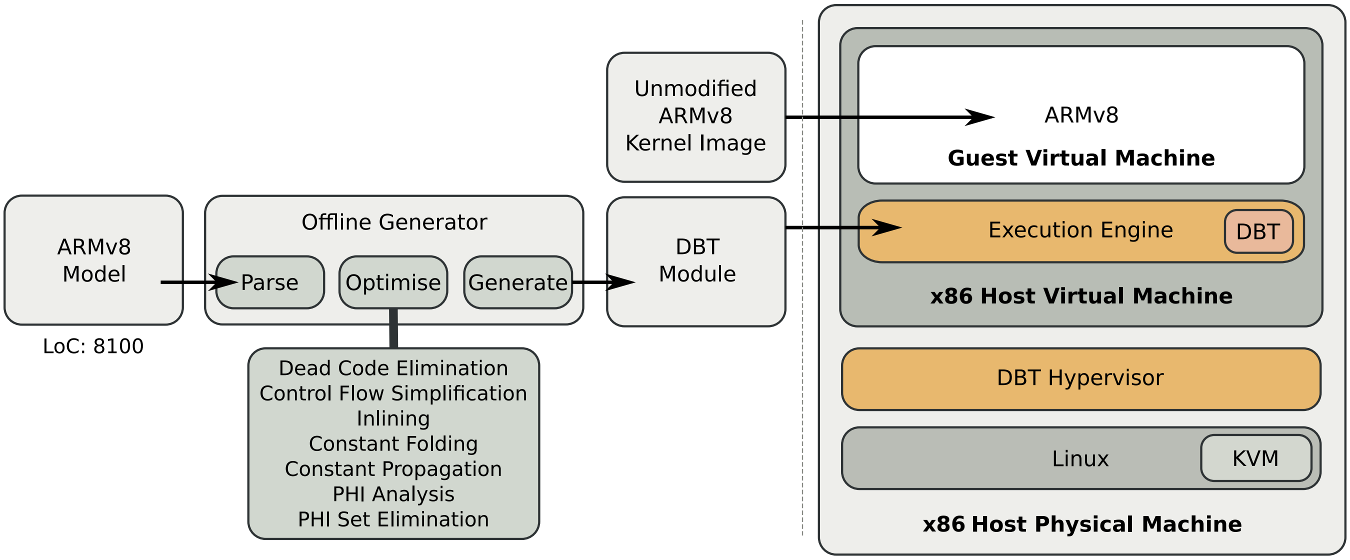


Figure 1: High-level overview of Captive.

to drive substantial workloads, i.e. software development tool chains and user applications. Whilst we currently focus on a single-core implementation, the key ideas can be translated to multi-core architectures.

We evaluate the implementation of Captive using a 64-bit ARMv8-A guest model and an x86-64 host. From a description comprising just 8100 lines of code[1] we generate a DBT hypervisor outperforming QEMU by a factor of 2.21× for SPEC CPU2006 integer applications, and up to 6.49× for floating-point workloads. This means Captive is capable of hosting a full and unmodified ARM Linux OS environment while delivering around 40% of the performance of a physical system comprising a 2.0GHz server-type Cortex-A57 processor.

1.1 Overview and Motivating Example

Figure 1 shows a high-level overview of Captive: an ARMv8-A[2] architecture description is processed by an offline tool to produce a platform-specific DBT module. Already at this stage optimizations are applied, which aid later JIT code generation. The software stack on the x86-64 host machine comprises a Kernel Virtual Machine (KVM)-based DBT hypervisor, operating on top of the host's Linux OS. This provides a *virtual* bare-metal x86-64 Host Virtual Machine (HVM) in which Captive together with the previously generated DBT module and a minimal execution engine reside to provide the Guest Virtual Machine (GVM), which can boot and run an unmodified ARMv8-A Linux kernel image. Since the JIT compiler in our system-level DBT system operates in a bare-metal HVM it has full access to the virtual host's resources and can generate code to exploit these resources.

For example, consider Figure 2. A conventional system-level DBT system hosted on an x86-64 architecture, e.g. QEMU, operates entirely as a user process in protection ring 3 on top of a host OS operating in ring 0. This means that any code generated by QEMU's JIT compiler, either guest user or system code, also operates in the host's ring 3, which restricts access to system features such as page tables. Such a system operating exclusively in ring 3 needs to provide software abstractions and protection mechanisms for guest operations, which modify guest system state. In contrast, Captive operates in VMX root mode, and provides a bare-metal HVM with rings 0-3. Our execution engine and DBT operate in the virtual machine's ring 0, and track the guest system's mode. This enables us to generate code operating in ring 0, for guest system code, and ring 3, for guest user code. This means we can use the HVM's hardware protection features to efficiently implement memory protection or allow the hypervisor to modify the HVM's page tables in order to directly map the GVM's virtual address space onto host physical memory.

Porting to a different host architecture can be accomplished by utilising similar features offered by that architecture, e.g. Arm offers virtualization extensions that are fully supported by KVM, and privilege levels `PL0` and `PL1`, which are similar to x86's ring 3 and ring 0, respectively. These similarities also enable our accelerated virtual memory system to work across platforms.

1.2 Contributions

Captive shares many concepts with existing DBT systems, but it goes beyond and introduces unique new features. We

[1] Compared to 17766 LoC for QEMU's ARM model plus a further 7948 LoC in their software floating-point implementation.

[2] Or any other guest architecture, e.g. RISC-V.

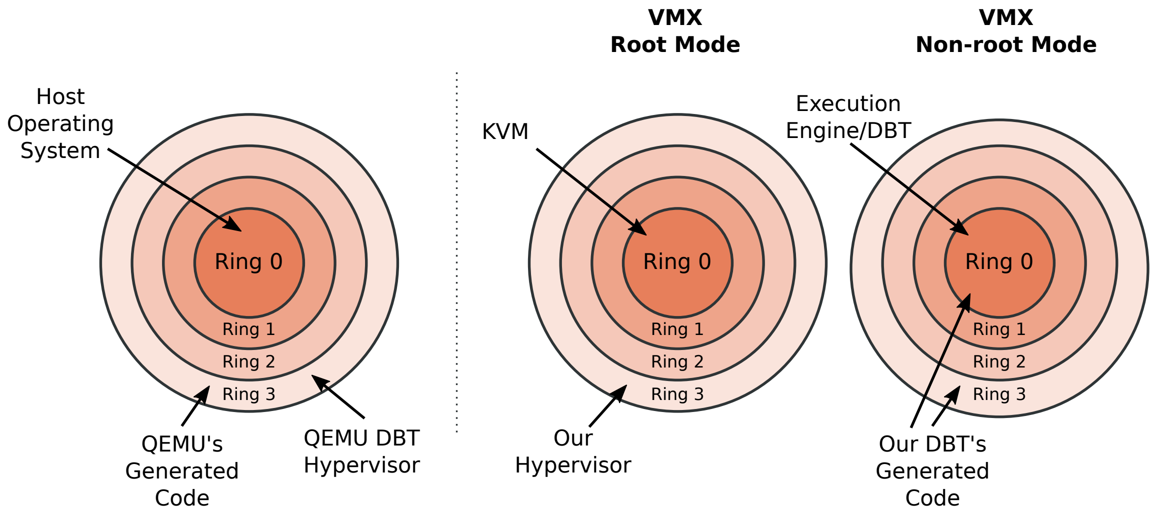


Figure 2: x86 protection rings. Ring 0 is the most privileged (kernel mode), and ring 3 is the least privileged (user mode). QEMU operates in ring 3, whereas Captive takes advantage of a host VM to operate in ring 0 and ring 3. The hypervisor component operates outside the host virtual machine, in *VMX root mode*.

	QEMU [4]	HQEMU [21]	PQEMU [18]	Walkabout [12]	Yirr-Ma [44]	ISAMAP [38]	Transmeta CMS [17]	Harmonia [31]	QuickTransit [26]	HyperMAMBO-x64 [15]	Captive (2016) [40]	MagiXen [10]	**Captive**
System-Level	✓	✗	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓
Retargetable	✓	✓	✗	✓	✓	✓	✗	✗	✓	✗	✓	✗	✓
Architecture Description Language	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓	✗	✓
Hypervisor	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
Host FP Supprt	✗	✓	✗	N/A	✗	✓	✓	✗	✓	✓	✓	✓	✓
FP bit-accurate	✓	✗	✓	N/A	N/A	N/A	✓	N/A	N/A	N/A	✗	✓	✓
Host SIMD Support	(✓)	(✓)	(✓)	✗	✗	✓	✓	✗	N/A	N/A	✗	✓	✓
64-bit support	✓	✓	✗	✓	✓	✓	✗	✗	N/A	✗	✗	✗	✓
Publicly Available	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Table 1: Feature comparison of DBT systems. Brackets indicate partial support.

provide a feature comparison in Table 1, and present further information on related work in Section 4. Among the contributions of this paper are:

1. We develop a generic system-level DBT framework, where the effort to support new guest platforms is reduced by using **high-level architecture descriptions**.

2. We use split compilation in a DBT, combining **offline and online optimization** to reduce pressure on the performance critical JIT compiler while maintaining code quality.

3. We pioneer a DBT approach where the integrated JIT compiler is part of a **DBT hypervisor** and can generate code that takes full advantage of this execution context.

Captive has been released as an open-source project, to enable community-driven development and independent performance evaluation.[3]

2 Retargetable DBT Hypervisor

2.1 Overview

In this section, we describe the key concepts of Captive, which comprises two main components, (1) an **offline** generation component, and (2) an **online** runtime component.

The offline phase involves describing the target machine architecture, and is discussed in Section 2.2. In this phase, modules for inclusion in the runtime component are generated. Complex architectural behaviour (such as the operation of the

[3] See https://gensim.org/simulators/captive

Optimization	Active in Opt. Level
Dead Code Elimination	O1–4
Unreachable Block Elimination	O1–4
Control Flow Simplification	O1–4
Block Merging	O1–4
Inlining	O1–4
Dead Variable Elimination	O1–4
Jump Threading	O2–4
Constant Folding	O3–4
Constant Propagation	O3–4
Value Propagation	O3–4
Load Coalescing	O3–4
Dead Write Elimination	O3–4
PHI Analysis	O4
PHI Elimination	O4

Table 2: Optimizations applied in the offline stage.

Memory Management Unit (MMU)) are described in regular source-code files, and compiled together with the generated source-code. The online runtime component is discussed in Section 2.3, and comprises a further two sub-components, (1) a **user-mode application** that activates and configures a KVM Virtual Machine, and (2) a **unikernel** that runs inside the KVM VM, and implements guest instruction translation and general guest machine execution.

The DBT system itself runs inside a VM with no standard OS support. Normally, a virtual machine provides a bare-metal environment in which an OS is loaded, and then user applications are executed. We instead skip the OS entirely, and implement our DBT on the virtual bare-metal hardware. Whilst this adds complexity to the implementation of the DBT, it also allows the DBT to directly use host architectural features, without having to negotiate with an OS. This is in contrast to the majority of other system-level DBTs, which typically run as user-mode applications in an OS. The trade-off here is that Captive relies on KVM, reducing host operating system portability.

2.2 Offline Stage

2.2.1 Architecture Description

The guest machine architecture is described using a high-level Architecture Description Language (ADL) that defines instruction syntax (i.e. how to decode instructions) and instruction semantics (i.e. how to execute instructions). The ADL is also used to describe architectural features, such as the register file size and layout, word sizes, endianness, etc.

The ADL is based on a modified version of ArchC [1], and our offline generator tool processes the description into an intermediate form, performs some optimization and analysis, before finally producing modules for the DBT as output.

Instruction semantics (the functional behavior of guest machine instructions) are described in a high-level C-like language. This Domain Specific Language (DSL) allows the behavior of instructions to be specified easily and naturally, by, e.g. translating the pseudo-code found in architecture manuals into corresponding C-like code.

```
1 execute(add) {
2   uint64 rn = read_register_bank(BANK0, inst.a);
3   uint64 rm = read_register_bank(BANK0, inst.b);
4   uint64 rd = rn + rm;
5   write_register_bank(BANK0, inst.a, rd);
6 }
```

Figure 3: High-level C-like representation of instruction behavior

Figure 3 provides an example description of an `add` instruction that loads the value from two guest registers (lines 3 and 4), adds them together (line 5), then stores the result to another guest register (line 6). This example shows how a typical instruction might look, and how its behavior can be naturally expressed. Of course, this is a simple example: most ‘real-world’ instruction descriptions contain branching paths to select specific instruction variants (e.g., flag-setting or not), more complex calculations, and floating point and vector operations, all of which can be handled by the ADL.

2.2.2 Intermediate SSA Form

During the offline phase, instruction behavior descriptions are translated into a domain-specific Static Single Assignment (SSA) form, and aggressively optimized. The optimization passes used have been selected based on common idioms in instruction descriptions. For example, very few loop-based optimizations are performed, since most individual instructions do not contain loops. Optimizing the model at the offline stage makes any simplifications utilized by the designer in the description less of a performance factor in the resulting code.

The domain-specific SSA contains operations for reading architectural registers, performing standard arithmetic operations on values of integral, floating-point and vector types, memory and peripheral device access and communication, and a variety of built-in functions for common architectural behaviors (such as flag calculations and floating point NaN/Infinity comparisons).

Additionally, meta-information about the SSA is held, indicating whether each operation is *fixed* or *dynamic*. Fixed operations are evaluated at instruction translation time, whereas dynamic operations must be executed at instruction run-time. For example, the calculation of a constant value, or control flow based on instruction fields is *fixed*, but computations which depend on register or memory values are *dynamic* [46]. Fixed operations can produce dynamic values, but dynamic operations must be executed as part of instruction emulation.

Figure 4 shows the direct translation of the instruction behavior (from Figure 3) into corresponding SSA form. A

```
action void add (Instruction sym_1_3_parameter_inst) [
  uint64 sym_14_0_rd
  uint64 sym_5_0_rn
  uint64 sym_9_0_rm
] {
  block b_0 {
    s_b_0_0 = struct sym_1_3_parameter_inst a;
    s_b_0_1 = bankregread 7 s_b_0_0;
    s_b_0_2: write sym_5_0_rn s_b_0_1;
    s_b_0_3 = struct sym_1_3_parameter_inst b;
    s_b_0_4 = bankregread 7 s_b_0_3;
    s_b_0_5: write sym_9_0_rm s_b_0_4;
    s_b_0_6 = read sym_5_0_rn;
    s_b_0_7 = read sym_9_0_rm;
    s_b_0_8 = binary + s_b_0_6 s_b_0_7;
    s_b_0_9: write sym_14_0_rd s_b_0_8;
    s_b_0_10 = struct sym_1_3_parameter_inst a;
    s_b_0_11 = read sym_14_0_rd;
    s_b_0_12: bankregwrite 0 s_b_0_10 s_b_0_11;
    s_b_0_13: return;
  }
}
```

Figure 4: Unoptimized domain-specific SSA form of the `add` instruction from Figure 3.

```
action void add (Instruction sym_1_3_parameter_inst) [] {
  block b_0 {
    s_b_0_0 = struct sym_1_3_parameter_inst a;
    s_b_0_1 = bankregread 7 s_b_0_0;
    s_b_0_2 = struct sym_1_3_parameter_inst b;
    s_b_0_3 = bankregread 7 s_b_0_2;
    s_b_0_4 = binary + s_b_0_1 s_b_0_3;
    s_b_0_5 = struct sym_1_3_parameter_inst a;
    s_b_0_6: bankregwrite 0 s_b_0_5 s_b_0_4;
    s_b_0_7: return;
  }
}
```

Figure 5: Equivalent optimized domain-specific SSA form of the `add` instruction from Figure 3.

series of optimizations (given in Table 2) are then applied to this SSA, until a fixed-point is reached. Figure 5 shows the optimized form of the SSA.

The offline optimizations allow the user to be expressive and verbose in their implementation of the model, whilst retaining a concise final representation of the user's intent. For example, dead code elimination is necessary in the case where helper functions have been inlined, and subsequently subjected to constant propagation/folding, which eliminates a particular control-flow path through the function.

2.2.3 Generator Function

The domain-specific SSA itself is not used at runtime, but instead is used in the final offline stage to build simulator-specific generator functions. These functions are either compiled in, or dynamically loaded, by the DBT, and are invoked at JIT compilation time. The generator functions call into the DBT backend, which produces host machine code. When an instruction is to be translated by the DBT, the corresponding generator function is invoked.

Figure 6 shows the corresponding generator function, produced from the optimized SSA form in Figure 5. The generator function is clearly machine generated, but *host* compiler optimizations (in the offline stage) will take care of any inefficiencies in the output source-code. Additionally (and not shown for brevity) the offline stage generates source-code comments, to assist in debugging.

2.3 Online Stage

The online stage of Captive involves the actual creation and running of the guest virtual machine. This takes the form of a KVM-based DBT hypervisor, which instantiates an empty *host virtual machine*, which then loads the *execution engine* (a small, specialized unikernel) that implements the DBT. The KVM-based portion of the hypervisor also includes software emulations of guest architectural devices (such as the interrupt controller, UARTs, etc). The DBT comprises four main phases, as shown in Figure 7: **Instruction Decoding**, **Translation**, **Register Allocation**, and finally **Instruction Encoding**.

2.3.1 Instruction Decoding

The first phase in our execution pipeline is the instruction decoder, which will decode one *guest* basic block's worth of instructions at a time. The decoder routines are automatically generated from the architecture description during the offline stage, utilizing techniques such as Krishna and Austin [27], Theiling [43].

2.3.2 Translation

During the translation phase, a generator function (that was created in the offline stage) is invoked for each decoded instruction. The generator function calls into an *invocation Directed Acyclic Graph (DAG) builder*, which builds a DAG representing the data-flow and control-flow of the instruction under translation. Operations (represented by nodes in the DAG) that have side effects result in the collapse of the DAG at that point, and the emission of low-level Intermediate Representation (IR) instructions representing the collapsed nodes.

A node with side effects is one through which control-flow cannot proceed without the state of the guest being mutated in some way. For example, a `STORE` node is considered to have side-effects, as the guest machine register file has been changed.

During emission, the tree rooted at that node is traversed, emitting IR for the operations required to produce the input values for that node. This feed-forward technique removes the need to build an entire tree then traverse it later. Collapsing nodes immediately to IR improves the performance of the DBT, as instructions are generated as soon as possible.

```
bool generator::translate_add(const test_decode_test_F1& insn, dbt_emitter& emitter) {
  basic_block *__exit_block = emitter.create_block();
  goto fixed_block_b_0;
fixed_block_b_0:  {
    auto s_b_0_1 = emitter.load_register(emitter.const_u32((uint32_t)(256 + (16 * insn.a))), dbt_types::u64);
    auto s_b_0_3 = emitter.load_register(emitter.const_u32((uint32_t)(256 + (16 * insn.b))), dbt_types::u64);
    auto s_b_0_4 = emitter.add(s_b_0_1, s_b_0_3);
    emitter.store_register(emitter.const_u32((uint32_t)(0 + (8 * insn.a))), s_b_0_4);
    goto fixed_done;
  }
fixed_done:
  emitter.jump(__exit_block);
  emitter.set_current_block(__exit_block);
  if (!insn.end_of_block) emitter.inc_pc(emitter.const_u8(4));
  return true;
}
```

Figure 6: Generator function produced from ADL code shown in Figure 3

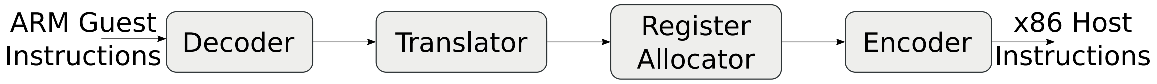


Figure 7: Online flow including decoder, translator, register allocation and instruction encoder.

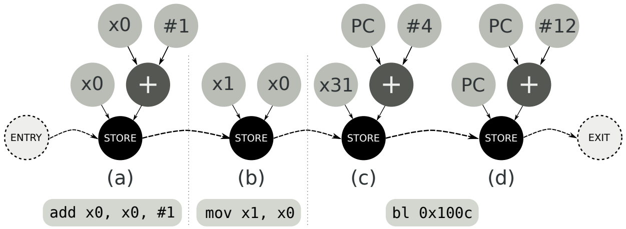


Figure 8: Example ARM assembly, and the corresponding (uncollapsed) DAG built during translation. Nodes (a), (b), (c), and (d) have side effects, causing the emission of low-level IR based on the tree rooted at that node.

```
mov (%rbp), %VREG0   ; Load guest reg. into temporary
add $1, %VREG0       ; Add one.
mov %VREG0, (%rbp)   ; Store temporary to guest reg.
mov (%rbp), %VREG1   ; Load guest reg. into temporary
mov %VREG1, 8(%rbp)  ; Store temporary to guest reg.
lea 4(%r15), %VREG2  ; Load PC+4 into temporary
mov %VREG2, 0xf8(%rbp) ; Store into guest reg.
add $12, %r15        ; Increment PC by 12
```

Figure 9: As each node with side-effects is inserted into the DAG, low-level IR is emitted that implements that node. This IR represents host instructions, but with virtual registers instead of physical registers.

This strategy enables high-level operations to take place on transparent values, and implements a weak form of tree pattern matching on demand. When a node is collapsed, specializations can be made depending on how the tree is formed at the node. For example, the STORE node ((d) in Figure 8) that updates the PC by incrementing its value, can be emitted as a single x86 instruction. Instruction selection also takes place at this level, where the generator can utilize host instructions, such as fused-multiply-add when available.

In the case of an x86 host machine, the low-level IR is effectively x86 machine instructions, but with virtual register operands in place of physical registers, as shown in Figure 9. For other host machines, the IR is similar.

2.3.3 Register Allocation

After the low-level IR has been produced by the translation phase, the register allocator makes a forward pass over these instructions to discover live ranges, and then a backward pass to split live ranges into live intervals. During live-range splitting, host machine registers are allocated to virtual registers, and conflicts are resolved. Whilst not producing an optimal solution, the register allocator is fast. The allocator also marks dead instructions, so that at encoding time those instructions are ignored. Our register allocation algorithm is similar to the *simplified graph-coloring* scheme from Cai et al. [9], but with additional dead code elimination.

2.3.4 Instruction Encoding

After register allocation is complete, the low-level intermediate form of instructions can be directly lowered into machine code. The list of instructions is traversed for a final time, and the machine code is generated directly from the instruction's meta-data, into a code buffer. Any instructions that were classified as dead during register allocation are skipped.

Once machine code emission is completed, a final pass is made to apply patches to relative jump instructions, as this value is only known once each instruction has been emitted, and therefore sized.

```
fmov d0, #1.5     ; Store constant 1.5 in d0
fmul d0, d1, d2   ; Multiply d1 with d2, and store in d0
```

Figure 10: Arm floating-point input assembly

2.4 Exploiting Host Architectural Features

System-level DBT naturally involves emulating a range of guest architectural components, most notably the MMU. Traditionally, this emulation is performed in software, where each memory access must perform an *address translation* that takes a *virtual* address, and translates it via the guest page tables to a corresponding *guest physical address*. In QEMU, a cache is used to accelerate this translation, but in Captive we utilize the host MMU directly by mapping guest page table entries to equivalent host page table entries. This reduces the overhead of memory access instructions significantly, as we do not need to perform cache look-ups, and can work with the guest virtual address directly. Larger guest page sizes are supported by the host MMU directly, as multiple host pages can represent a single larger guest page. In the case of smaller guest pages, we must emulate memory accesses carefully to ensure permissions within a page are not violated. In general, we support an $n : m$ mapping between guest and host page sizes, where n, m are powers of 2.

This technique is not possible with a DBT that runs in user-mode, as the OS retains control of the host MMU page tables (although attempts have been made to emulate this by using the `mmap` system call [51]). However, with Captive, we are operating in a bare-metal environment (see Figure 1), and are able to configure the host architecture in any way we want. By tracking the protection ring of the guest machine, and executing the translated guest code in the corresponding host protection ring, we can take advantage of the host system's memory protection mechanism, for efficient implementation of guest memory protection.

We also take advantage of the x86 software interrupt mechanism (invoked using the `int` instruction), the x86 port-based I/O instructions (`in` and `out`), and the x86 fast system call instructions (`syscall` and `sysret`). These features are used to accelerate implementations of instructions that require additional non-trivial behaviors, e.g. accessing co-processors, manipulation of page tables, flushing Translation Lookaside Buffers (TLBs), and other operations specific to system-level DBT.

2.5 Floating Point/SIMD Support

In order to reduce JIT complexity, QEMU uses a software floating-point implementation, where helper methods are used to implement floating-point operations. This results in the emission of a function call as part of the instruction execution, adding significant overhead to the emulation of these instructions. Figure 10 gives an example of two ARM floating-point instructions, which are translated by QEMU to the x86 code in Figure 11, and by Captive to the code in Figure 12. Whilst QEMU implements the `fmov` directly (lines 1—3), in much the same way as Captive, QEMU issues a function call for the floating-point multiplication (`fmul`). In contrast, Captive emits a host floating-point multiplication instruction, which operates directly on the guest register file.

```
 1 movabs  $0x3ff8000000000000, %rbp ; Store const FP value
 2 mov     %rbp, 0x8c0(%r14)         ; of 1.5 in guest
 3 movq    $0, 0x8c8(%r14)           ; register file.
 4 lea     0x8d0(%r14), %rbp
 5 mov     %rbp, %rdi
 6 mov     $0x3bd, %esi
 7 xor     %edx, %edx
 8 callq   0x55d337b70220       ; call gvec_dup8 helper
 9 lea     0x2b68(%r14), %rbp   ; Prepare arguments for
10 mov     0x9c0(%r14), %rbx    ; invocation of FP
11 mov     0xac0(%r14), %r12    ; multiply helper
12 mov     %rbx, %rdi           ; function.
13 mov     %r12, %rsi
14 mov     %rbp, %rdx
15 callq   0x55d337bd0050       ; Invoke helper
16 mov     %rax, 0x8c0(%r14)    ; Store result in
17 movq    $0, 0x8c8(%r14)      ; guest register file.
```

Figure 11: QEMU output assembly for the instruction sequence in Figure 10.

```
movabs $0x3ff8000000000000,%rax ; Store const FP value
mov    %rax,0x100(%rbp)         ; of 1.5 in guest
movq   $0x0,0x108(%rbp)         ; register file.
add    $0x4,%r15             ; Increment PC
movq   0x110(%rbp),%xmm0     ; Load FP multiply operand
mulsd  0x120(%rbp),%xmm0     ; Perform multiplication
movq   %xmm0,0x100(%rbp)     ; Store result
movq   $0x0,0x108(%rbp)
add    $0x4,%r15             ; Increment PC
```

Figure 12: Captive output assembly for the instruction sequence in Figure 10.

Not all floating-point operations are trivial, however. Notably, there are significant differences with the way floating-point flags, NaNs, rounding modes, and infinities are handled by the underlying architecture, and in some cases this incompatibility between floating-point implementations needs to be accounted for. In these cases, Captive emits fix-up code that will ensure the guest machine state is bit-accurate with how the guest machine would normally operate. Captive only supports situations where the host machine is at least as precise as the guest. This is the most common scenario for our use cases, but in the event of a precision mismatch, we can either (a) use the x86 80-bit FPU (to access additional precision), or (b) utilise a software floating-point library.

Like QEMU, Captive emits Single Instruction Multiple Data (SIMD) instructions when translating a guest vector instruction, however QEMU's support is restricted to integer and bit-wise vector operations whereas Captive more aggressively utilizes host SIMD support.

2.6 Translated Code Management

Captive employs a code cache, similar to QEMU, which maintains the translated code sequences. The key difference is that we index our translations by *guest physical* address, while QEMU indexes by *guest virtual* address. The consequence of this is that our translations are retained and re-used for longer, whereas QEMU must invalidate all translations when the guest page tables are changed. In contrast, we only invalidate translations when self-modifying code is detected. We utilize our ability to write-protect virtual pages to efficiently detect when a guest memory write may modify translated code, and hence invalidate translations only when necessary. A further benefit is that translated code is re-used across different virtual mappings to the same physical address, e.g. when using shared libraries.

2.7 Virtual Memory Management

To accelerate virtual memory accesses in the guest, we dedicate the lower half of the host VMs virtual address space for the guest machine, and utilise the upper half for use by Captive. The lower half of the address space is mapped by taking corresponding guest page table entries, and turning them into equivalent host page table entries.

To make a memory access, the guest virtual address is masked, to keep it within the lower range, and if the address actually came from a higher address, the host page tables are switched to map the lower addresses to guest upper addresses. The memory access is then performed using the masked address directly, thus benefitting from host MMU acceleration.

3 Evaluation

Performance comparisons in the DBT space are difficult: most of the existing systems are not publicly available, and insufficient information is provided to reconstruct these systems from scratch. Furthermore, results published in the literature often make use of different guest/host architecture pairs and differ in supported features, which prohibit meaningful relative performance comparisons.[4] For this reason we evaluate Captive against the widely used QEMU DBT as a baseline, supported by targeted micro-benchmarks and comparisons to physical platforms.

3.1 Experimental Set-up

While we support a number of guest architectures, we choose to evaluate Captive using an ARMv8-A guest and an x86-64 host.[5] We conducted the following experiments on the host machine described in Table 3, and performed our comparison to native architectures on a Raspberry PI 3B, and an AMD Opteron A1100 (Table 4). We utilized both the integer and C++ floating-point benchmarks from SPEC CPU2006. Our comparisons to QEMU were made with version 2.12.1.

System	HP z440		
Architecture	x86-64	*Model*	Intel® Xeon® E5-1620 v3
Cores/Threads	4/8	*Frequency*	3.5 GHz
L1 Cache	I$128kB/D$128kB	*L2 Cache*	1MB
L3 Cache	10 MB	*Memory*	16 GB

Table 3: DBT Host System

System	AMD Opteron A1170		
Architecture	ARMv8-A	*Model*	Cortex A57
Cores/Threads	8/8	*Frequency*	2.0 GHz
L1 Cache	I$48kB/D$32kB	*L2 Cache*	4×1 MB
L3 Cache	8 MB	*Memory*	16 GB
System	Raspberry Pi 3 Model B		
Architecture	ARMv8-A	*Model*	Cortex A53
Cores/Threads	4/4	*Frequency*	1.2 GHz
L1 Cache	I$16kB/D$16kB	*L2 Cache*	512kB
L3 Cache	-/-	*Memory*	1 GB

Table 4: Native Arm Host Systems

3.2 Application Benchmarks

We have evaluated the performance of Captive and QEMU using the standard SPEC2006 benchmark suite with the *Reference* data set. As can be seen in Figure 13, we obtain significant speedups in most *Integer* benchmarks, with a geometric mean speedup of 2.2×. The two benchmarks where we experience a slow-down are `456.hmmer` and `462.libquantum`, which can be attributed to suboptimal register allocation in hot code. Figure 14 shows the speed up of Captive over QEMU on the *C++ Floating Point* portion of the benchmark suite.[6] Here we obtain a geometric mean speedup of 6.49×. This large speedup can mainly be attributed to QEMU's use of a software floating point implementation, while we use the host FPU and vector units directly.

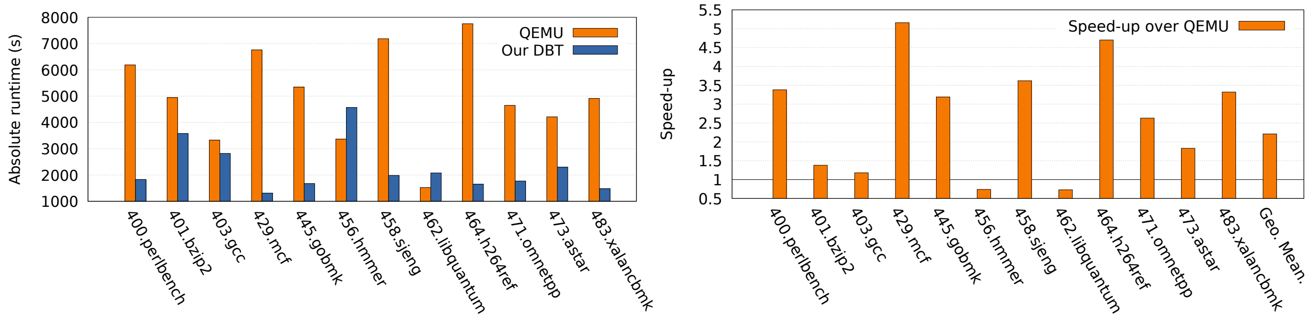
3.3 Additional Guest Architectures

We also have descriptions in our ADL for other guest architectures, detailed in Table 5. However, with the exception of ARMv7-A, these implementations currently lack full-system support. For the ARMv7-A case, we have observed similar average speed-ups of 2.5×, and up to 6× across the SPEC CPU2006 benchmark suite using Captive.

[4] For example, Harmonia [31] achieves a similar speedup of 2.2 over QEMU, but this is for user-level DBT of a 32-bit guest on a 64-bit host system whereas we achieve a speedup of 2.2 over QEMU for the harder problem of system-level DBT of a 64-bit guest onto a 64-bit host system.

[5] Additional RISC-V and x86 models will be released together with Captive.

[6] Missing Fortran benchmarks are due to the benchmarks not working both natively, and in QEMU.



(a) Absolute runtime in seconds (lower is better)

(b) Speed-up of Captive over QEMU (higher is better)

Figure 13: Application Performance: SPEC CPU2006 Integer benchmark results for Captive vs QEMU.

Architecture	Challenges	Solution
ARMv8-A	64-bit guest on 64-bit host emulation	Additional techniques for MMU emulation
ARMv7-A	If-then-else blocks, possibly spanning page boundaries	Complex control-flow handling in the JIT
x86-64	Complex instruction encoding, requiring stateful decoder.	Use of an external decoder library [23]
RISC-V	No significiant challenges	None required
TI TMS320C6x DSP	VLIW instructions, nested branch delay slots	Extensions to decoder generator, control-flow recovery
Arm Mali-G71 GPU	Complex instruction bundle headers	External “pre”-decoder for bundle headers.

Table 5: Architectures currently supported by Captive, and the architecture-specific challenges that required special attention for implementation.

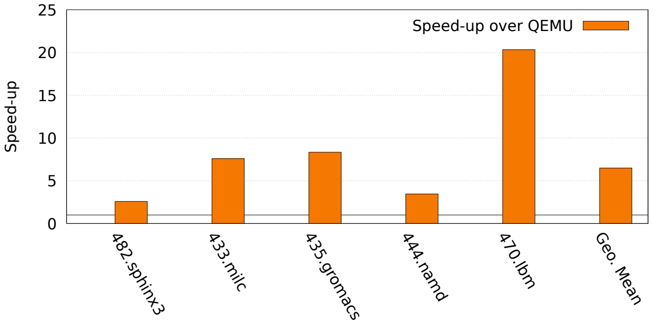


Figure 14: Speed-up of Captive over QEMU on the *Floating Point* portion of the SPEC2006 benchmark suite (higher is better)

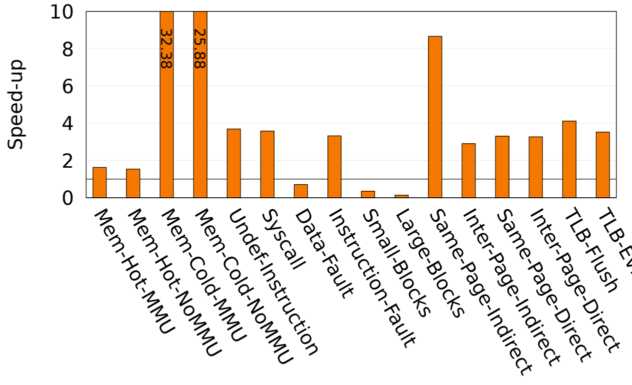


Figure 15: Speed-up of Captive over QEMU on the SimBench micro-benchmark suite

3.4 JIT Compilation Performance

Captive is on average 2.6× slower at translating guest basic blocks than QEMU. This is due in part to the more aggressive online optimizations we perform, but additionally QEMU's DBT has had years of hand-tuning, and benefits from a monolithic implementation.

However, the previous results clearly indicate that our compilation latency does not affect the runtime of the benchmarks. In fact, the extra effort we put into compilation ensures that our code quality surpasses that of QEMU's, as will be demonstrated in Section 3.6. Figure 15 shows that indeed, when using the SimBench micro-benchmark suite [47], the *Large-Blocks* and *Small-Blocks* benchmark indicate that our code generation speed is 65% and 85% slower, respectively. These benchmarks are described in Section 3.5.

Figure 16 provides a further breakdown of the time spent for JIT compilation: instruction translation (including invocation DAG generation and instruction selection) takes up more than 50% of the total JIT compilation time, followed by register allocation (including liveness analysis and dead code elimination), then host instruction encoding. Guest instruction decoding takes up 2.75% of the compilation pipeline.

We have also collected aggregate translation size statistics for `429.mcf`. We found that Captive generates larger code than QEMU, with Captive generating 67.53 bytes of host code per guest instruction, compared to QEMU's 40.26 bytes. This is due the use of vector operations in the benchmarks: while QEMU frequently emits (relatively small) function calls for

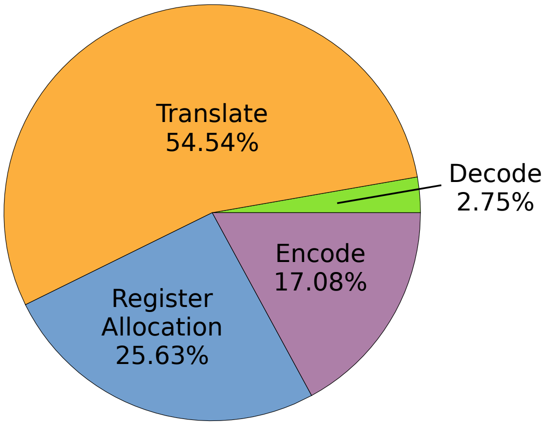


Figure 16: % time spent in each compilation phase: *Decode*, *Translate*, *Register Allocation* and *Encode*.

these operations, Captive emits vector operations directly. In particular, vector load and store operations require that vectors are packed and unpacked element by element, each of which can require 2–3 instructions.

3.5 Targeted Micro-Benchmarks

As well as using the SPEC benchmark suite, we have also evaluated the performance of both Captive and QEMU using SimBench[47]. This is a targeted suite of micro-benchmarks designed to analyze the performance of full system emulation platforms in a number of categories, such as the performance of the memory emulation system, control flow handling, and translation speed (in the case of DBT-based systems).

Figure 15 shows the results of running SimBench on Captive and QEMU, in terms of speedup over QEMU. Captive outperforms QEMU in most categories, except for code generation (Large-Blocks and Small-Blocks) and Data Fault handling. Captive's use of the host memory management systems results in large speedups on the memory benchmarks.

3.6 Code Quality

We assess code quality by measuring the individual basic block execution time for each block executed as part of a benchmark. For example, consider the scatter plot in Figure 17, where we show the measured aggregated block execution times across the `429.mcf` benchmark for Captive and QEMU. In order to limit the influence of infrastructure components of both platforms we have disabled block chaining for both platforms. Block execution times have been measured in the same way for both systems using the host's `rdtscp` instruction, inserted around generated native code regions representing a guest block.

A regression line and 1:1 line are also plotted in the log-log scale plot. Most points are above the 1:1 line, indicating that the vast majority of blocks are executed more quickly on Captive than on QEMU. In fact, we observe a code quality related speedup of 3.44 for this benchmark, represented by the positive shift of the regression line along the y-axis.

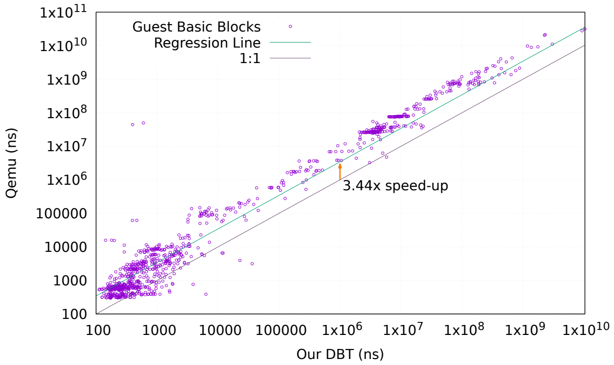


Figure 17: Measuring code quality: accumulated execution times of guest basic blocks from `429.mcf`. Blocks compiled by Captive execute, on average, 3.44× faster than their QEMU counterparts.

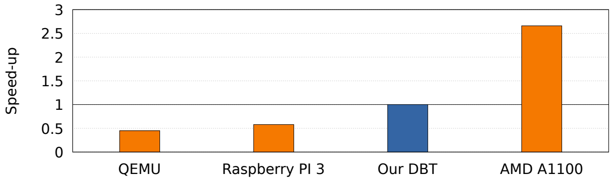


Figure 18: Comparison of Captive against native execution on two physical ARMv8-A platforms: Raspberry Pi 3 Model B & AMD Opteron A1170.

Further investigation reveals that Captive emits and executes, on average, 10 host instructions per guest instruction in addition to any block prologue and epilogue.

3.6.1 Impact of offline optimizations

Our offline generation system has four levels of optimization (`O1`–`O4`), although in practice we only use the maximum optimization level. These optimizations directly affect the amount of source code generated in the offline phase, where lower levels (e.g. `O1`) emit longer code sequences in the generator functions. This translates to more operations to perform at JIT compilation time, and therefore (a) larger JIT compilation latency, and (b) poorer code quality.

At the `O1` optimization level, only function inlining is performed, and results in the ARMv8A model comprising 271,299 lines of generated code. At `O4` (where a series of aggressive domain specific optimizations are performed), there is a reduction of 56%, to 120,162 lines of generated code.

	DBT System (Year)	Guest ISA	Host ISA	Distinct Contributions
User-Level DBT	Shade [13] (1993)	SPARC/MIPS	SPARC	DBT, code caching, tracing
	DAISY [19] (1997)	RS/6000	VLIW	Dyn. parallel scheduling
	FX!32 [11] (1998)	IA-32	Alpha	profiling & static BT
	UQDBT [45] (2000)	IA-32	SPARC	Retargetability
	Dynamo [2] (2000)	PA-RISC, IA-32	PA-RISC, IA-32	Same ISA Optimization
	Strata [34, 35] (2001)	SPARC/MIPS/IA-32	SPARC/MIPS	Extensibility
	Vulcan [42] (2001)	IA-32, IA-64, MSIL	IA-32, IA-64, MSIL	Het. binaries, distr. opt.
	bintrans [32] (2002)	PowerPC	Alpha	Dynamic liveness analysis
	Walkabout [12] (2002)	Retargetable (SPARC v8)	Retargetable (SPARC v9)	Arch. Descr. Lang. Interpreter and JIT generated
	DynamoRIO [7] (2003)	IA-32	IA-32	Dyn. Adapt. Optimization
	QuickTransit [26] (2004)	MIPS, PowerPC, SPARC	IA-32, IA-64, x86-64	KVM for memory translation
	Yirr-Ma [44] (2005)	Retargetable (SPARC, IA-32, ARM, PowerPC)	Retargetable (SPARC, IA-32, PowerPC)	Dyn. Opt., Part. Inlining Gen. from Spec.
	IA-32 EL [28] (2006)	IA-32	IA-64	SIMD Support
	StarDBT [48] (2007)	IA-32, x86-64	IA-32, x86-64	Trace lengthening
	N/A [6] (2008)	MIPS, VLIW	x86-64	LLVM JIT Compilation
	EHS [24] (2008)	ARC700	IA-32	Large translation regions
	Strata-ARM [30] (2009)	ARM	ARM, IA-32	Handling of exposed PC
	ISAMAP [38] (2010)	PowerPC	IA-32	Arch. Descr. Language
	ARCSim [5] (2011)	ARC700	x86-64	Parallel JIT task farm
	Harmonia [31] (2011)	ARM	IA-32	Reg. Map., Cond. codes Tiered compilation
	HQEMU [21] (2012)	ARMv7A	x86-64	Multithreaded Compilation
	HERMES [55] (2015)	IA-32, ARM	MIPS	Post-Optimization
	Pydgin [29] (2015)	ARM/MIPS	x86-64	Meta-Tracing JIT Compiler
	MAMBO-X64 [16] (2017)	AArch32	AArch64	Dyn. mapping of FP regs. Overflow address calculations Return address prediction
	HyperMAMBO-X64 [15] (2017)	AArch32	AArch64	Hypervisor support
	Pico [14] (2017)	x86-64, AArch64	x86-64, POWER8	multicore, multi-threaded DBT
System-Level DBT	Embra [53] (1996)	MIPS	MIPS	Multi-core, block chaining MMU relocation array
	Transmeta CMS [17] (2003)	IA-32	Custom VLIW	Aggressive speculation Hardware support Adaptive recompilation
	QEMU [4] (2204)	Retargetable	Retargetable	Pseudo Instructions
	MagiXen [10]	IA-32	IA-64	Integration with XEN
	PQEMU [18] (2011)	ARM	x86-64	Multi-core guest platform
	LIntel [37] (2012)	IA-32	Elbrus	Adapt. background opt.
	Captive [40]	ARMv7A	x86-64	VT Hardware Acceleration
	HybridDBT [33] (2017)	RISC-V	VLIW	Custom DBT Hardware
	Captive	Retargetable (ARMv8)	Retargetable (x86-64 + VT)	Aggressive offline optim. VM & bare-metal JIT

Table 6: Related Work: Feature comparison of existing DBT systems.

Reference	Guest ISA	Host ISA	Static/Dynamic	User/System	Distinct Contribution
Xu et al. [54]	IA-32	IA-64	Dynamic	User	Compiler Metadata
Bansal and Aiken [3]	PowerPC	IA-32	Static	User	Peephole translation rules learned by superoptimizer
Kedia and Bansal [25]	x86-64	x86-64	Dynamic	System	Kernel-level DBT
Hawkins et al. [20]	x86-64	x86-64	Dynamic	User	Optimization of Dyn. Gen. Code
Spink et al. [39]	ARMv5T	x86-64	Dynamic	User	Support for Dual-ISA
Wang et al. [49]	IA-32	x86-64	Dynamic	User	Persistent code caching
Shigenobu et al. [36]	ARMv7A	LLVM-IR	Static	User	ARM-to-LLVM IR
Wang et al. [50]	ARMv5	x86-64	Dynamic	System	Learning of translation rules
Hong et al. [22]	ARM NEON	x86 AVX2/AVX-512	Dynamic	User	Short-SIMD to Long-SIMD

Table 7: Related Work: Individual compilation techniques for Binary Translation systems.

3.6.2 Hardware Floating-point Emulation

In contrast to QEMU, Captive utilises a hardware emulated floating-point approach, where guest floating-point instructions are directly mapped to corresponding host floating-point instructions, if appropriate. Any fix-ups required to maintain bit-accuracy are performed inline, rather than calling out to helper functions. This increases the complexity of host portability, but significantly improves performance.

To determine the effect of this, we utilised a microbenchmark that exercised a small subset of (common) floating-point operations, and observed a speed-up of 2.17× of Captive (with hardware floating-point emulation) over QEMU (with software floating-point emulation). We then replaced our DBT's floating-point implementation with a software-based one (taken directly from the QEMU source-code), and observed a speed-up of 1.68×. This translates to a speed-up of 1.3× within Captive itself.

3.7 Comparison to Native Execution

We also compare the performance of Captive against two ARMv8-A hardware platforms: a Raspberry Pi 3 Model B and an AMD Opteron A1170 based server (see Table 4). The results of this comparison can be seen in Figure 18 and enable us to compare absolute performance levels in relation to physical platforms: across the entire SPEC CPU2006 suite Captive is about twice as fast as a 1.2GHz Cortex-A53 core of a Raspberry Pi 3, and achieves about 40% of the performance of a 2.0GHz Cortex-A57 core of the A1170. While outperformed by server processors it indicates that Captive can deliver performance sufficient for complex applications.

Finally, we compare the performance of Captive against native execution of the benchmarks compiled for and directly executed on the x86-64 host. Across all benchmarks we observe a speedup of 7.24 of native execution over system-level DBT, i.e. the overhead is still substantial, but Captive has significantly narrowed the performance gap between native execution, and system-level DBT.

4 Related Work

Due to their versatility DBT systems have found extensive interest in the academic community, especially since the mid-90s. In Table 6 we compare features and highlight specific contributions of many relevant DBT systems and techniques presented in the academic literature. The vast majority of existing DBT systems only provide support for user-level applications, but there also exist a number of system-level DBT approaches to which we compare Captive. In addition, numerous individual compilation techniques have been developed specifically for binary translators. Those relevant to our work on Captive are summarized in Table 7.

Captive is inspired by existing system-level DBT systems and we have adopted proven features while developing novel. Like Shade [13], Embra [53], and QEMU [4] Captive is interpreter-less and uses a basic block compiler with block chaining and trace caching. Our binary translator, however, is not hand-coded, but generated from a machine description. This allows for ease-of-retargeting comparable to Pydgin [29], but at substantially higher performance levels. Unlike Walkabout [12], Yirr-Ma [44], or ISAMAP [38], which similarly rely on machine descriptions, Captive employs split compilation and applies several optimizations offline, i.e. at module generation time, rather than relying on expensive runtime optimizations only. Instead of software emulation of floating-point (FP) arithmetic like QEMU or unsafe FP implementation like HQEMU [21], our FP implementation is bit-accurate, but still leverages the host system's FP capabilities wherever possible. Similar to IA-32 EL [28, 54] Captive translates guest SIMD instructions to host SIMD instructions wherever possible, but this mapping is generalized for any guest/host architecture pair. Like QuickTransit [26] or HyperMAMBO [15] Captive operates as a hypervisor, but provides a full-system environment rather than hosting only a single application. Captive shares this property with MagiXen [10], but provides full support for 64-bit guests on a 64-bit host rather than only 32-bit guests on a 64-bit host (which avoids address space mapping challenges introduced by same word-size system-level DBT).

5 Summary & Conclusion

In this paper we developed a novel system-level DBT hypervisor, which can be retargeted to new guest systems using a high-level ADL. We combine offline and online optimizations as well as a JIT compiler operating in a virtual bare-metal environment with full access to the virtual host processor to deliver performance exceeding that of conventional, manually optimized DBT systems operating as normal user processes. We demonstrate this using an ARMv8-A guest running a full unmodified ARM Linux environment on an x86-64 host, where Captive outperforms the popular QEMU DBT across SPEC CPU2006 application benchmarks while on average reaching 2× the performance of a 1.2GHz entry-level Cortex-A53 or 40% of a 2.0GHz server-type Cortex-A57.

5.1 Future Work

Our future work will consider support for multi- and many-core architectures, heterogeneous platforms, and support for various ISA extensions, e.g. for virtualization or secure enclaves, inside the virtualized guest system. We also plan to investigate possibilities for synthesizing guest and host architecture descriptions in the spirit of Buchwald et al. [8], or using existing formal specifications. We are also investigating a tiered compilation approach, to aggressively optimize hot code, and adding support for host retargeting, by using the same ADL as for our guest architectures.

References

[1] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5): 453–484, Oct 2005. ISSN 1573-7640. doi: 10.1007/s10766-005-7301-0. URL `https://doi.org/10.1007/s10766-005-7301-0`.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349303. URL `http://doi.acm.org/10.1145/349299.349303`.

[3] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 177–192, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855741.1855754`.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1247360.1247401`.

[5] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 74–85, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993508. URL `http://doi.acm.org/10.1145/1993498.1993508`.

[6] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. Fast and accurate simulation using the LLVM compiler framework. In *Workshop on Rapid Simulation and Performance Evalution: Methods and Tools (RAPIDO)*, 2008.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL `http://dl.acm.org/citation.cfm?id=776261.776290`.

[8] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 300–313, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5617-6. doi: 10.1145/3168821. URL `http://doi.acm.org/10.1145/3168821`.

[9] Z. Cai, A. Liang, Z. Qi, L. Jiang, X. Li, H. Guan, and Y. Chen. Performance comparison of register allocation algorithms in dynamic binary translation. In *2009 International Conference on Knowledge and Systems Engineering*, pages 113–119, Oct 2009. doi: 10.1109/KSE.2009.16.

[10] Matthew Chapman, Daniel J. Magenheimer, and Parthasarathy Ranganathan. Magixen: Combining binary translation and virtualization. Technical Report HPL-2007-77, Enterprise Systems and Software Laboratory, HP Laboratories Palo Alto, 2007.

[11] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. Fx!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March 1998. ISSN 0272-1732. doi: 10.1109/40.671403. URL `http://dx.doi.org/10.1109/40.671403`.

[12] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout: A retargetable dynamic binary translation framework. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2002.

[13] Robert F. Cmelik and David Keppel. Shade: A fast instruction set simulator for execution profiling. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1993.

[14] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang, editors, *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pages 210–220. ACM, 2017. ISBN 978-1-5090-4931-8. URL `http://dl.acm.org/citation.cfm?id=3049855`.

[15] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, John Goodacre, and Mikel Luján. Hypermambo-x64: Using virtualization to support high-performance transparent binary translation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 228–241, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4948-2. doi: 10.1145/3050748.3050756. URL `http://doi.acm.org/10.1145/3050748.3050756`.

[16] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low overhead dynamic binary translation on arm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 333–346, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062371. URL http://doi.acm.org/10.1145/3062341.3062371.

[17] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing&trade; software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL http://dl.acm.org/citation.cfm?id=776261.776263.

[18] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 276–283, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4576-9. doi: 10.1109/ICPADS.2011.102. URL https://doi.org/10.1109/ICPADS.2011.102.

[19] Kemal Ebcioğlu and Erik R. Altman. Daisy: Dynamic compilation for 100 In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 26–37, New York, NY, USA, 1997. ACM. ISBN 0-89791-901-7. doi: 10.1145/264107.264126. URL http://doi.acm.org/10.1145/264107.264126.

[20] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 68–78, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. URL http://dl.acm.org/citation.cfm?id=2738600.2738610.

[21] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259030. URL http://doi.acm.org/10.1145/2259016.2259030.

[22] Ding-Yong Hong, Yu-Ping Liu, Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. Improving simd parallelism via dynamic binary translation. *ACM Trans. Embed. Comput. Syst.*, 17(3):61:1–61:27, February 2018. ISSN 1539-9087. doi: 10.1145/3173456. URL http://doi.acm.org/10.1145/3173456.

[23] Intel. Intel xed, 2018. URL https://intelxed.github.io/. Retrieved on 01/11/2018.

[24] Daniel Jones and Nigel Topham. High speed cpu simulation using ltu dynamic binary translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 50–64, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-540-92989-5. doi: 10.1007/978-3-540-92990-1_6. URL http://dx.doi.org/10.1007/978-3-540-92990-1_6.

[25] Piyus Kedia and Sorav Bansal. Fast dynamic binary translation for the kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 101–115, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522718. URL http://doi.acm.org/10.1145/2517349.2522718.

[26] Paul Knowles. Transitive and QuickTransit overview, 2008.

[27] Rajeev Krishna and Todd Austin. Efficient software decoder design. *Technical Committee on Computer Architecture (TCCA) Newsletter*, October 2001.

[28] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. Optimizing dynamic binary translation for simd instructions. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.27. URL http://dx.doi.org/10.1109/CGO.2006.27.

[29] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing jit compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, March 2015. doi: 10.1109/ISPASS.2015.7095811.

[30] Ryan W. Moore, José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Addressing the challenges of dbt for the arm architecture. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '09, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3. doi: 10.1145/1542452.1542472. URL http://doi.acm.org/10.1145/1542452.1542472.

[31] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. Harmonia: A transparent, efficient, and harmonious dynamic binary translator targeting the intel architecture. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0698-0. doi: 10.1145/2016604.2016635. URL http://doi.acm.org/10.1145/2016604.2016635.

[32] M. Probst, A. Krall, and B. Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 35–44, 2002. doi: 10.1109/WCRE.2002.1173062.

[33] S. Rokicki, E. Rohou, and S. Derrien. Hardware-accelerated dynamic binary translation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1062–1067, March 2017. doi: 10.23919/DATE.2017.7927147.

[34] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL http://dl.acm.org/citation.cfm?id=776261.776265.

[35] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. Technical report, University of Virginia, Charlottesville, VA, USA, 2001.

[36] K. Shigenobu, K. Ootsu, T. Ohkawa, and T. Yokota. A translation method of arm machine code to llvm-ir for binary code parallelization and optimization. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, volume 00, pages 575–579, Nov. 2018. doi: 10.1109/CANDAR.2017.75. URL doi.ieeecomputersociety.org/10.1109/CANDAR.2017.75.

[37] R. A. Sokolov and A. V. Ermolovich. Background optimization in full system binary translation. *Programming and Computer Software*, 38(3): 119–126, Jun 2012. ISSN 1608-3261. doi: 10.1134/S0361768812030073. URL https://doi.org/10.1134/S0361768812030073.

[38] Maxwell Souza, Daniel Nic$#225;cio, and Guido Araújo. Isamap: Instruction mapping driven by dynamic binary translation. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, pages 117–138, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-24321-9. doi: 10.1007/978-3-642-24322-6_11. URL http://dx.doi.org/10.1007/978-3-642-24322-6_11.

[39] Tom Spink, Harry Wagstaff, Björn Franke, and Nigel P Topham. Efficient dual-isa support in a retargetable, asynchronous dynamic binary translator. In *SAMOS*, pages 103–112, 2015.

[40] Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Trans. Archit. Code Optim.*, 13(4): 36:1–36:25, October 2016. ISSN 1544-3566. doi: 10.1145/2996798. URL http://doi.acm.org/10.1145/2996798.

[41] Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. Mobile-sandbox: Combining static and dynamic analysis with machine-learning techniques. *Int. J. Inf. Secur.*, 14 (2):141–153, April 2015. ISSN 1615-5262. doi: 10.1007/s10207-014-0250-0. URL http://dx.doi.org/10.1007/s10207-014-0250-0.

[42] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft Research, April 2001. URL https://www.microsoft.com/en-us/research/publication/vulcan-binary-transformation-in-a-distributed-environ

[43] Henrik Theiling. Generating decision trees for decoding binaries. In *Proceedings of the 2001 ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, OM '01, pages 112–120, New York, NY, USA, 2001. ACM. ISBN 1-58113-426-6. doi: 10.1145/384198.384213. URL http://doi.acm.org/10.1145/384198.384213.

[44] Jens Tröger. *Specification-driven dynamic binary translation*. PhD thesis, Queensland University of Technology, 2005. URL https://eprints.qut.edu.au/16007/.

[45] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO '00, pages 41–51, New York, NY, USA, 2000. ACM. ISBN 1-58113-241-7. doi: 10.1145/351397.351414. URL http://doi.acm.org/10.1145/351397.351414.

[46] H. Wagstaff, M. Gould, B. Franke, and N. Topham. Early partial evaluation in a jit-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *2013 50th ACM/EDAC/IEEE Design*

Automation Conference (DAC), pages 1–6, May 2013. doi: 10.1145/2463209.2488760.

[47] H. Wagstaff, B. Bodin, T. Spink, and B. Franke. Simbench: A portable benchmarking methodology for full-system simulators. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 217–226, April 2017. doi: 10.1109/ISPASS.2017.7975293.

[48] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, pages 4–15, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74308-1, 978-3-540-74308-8. URL `http://dl.acm.org/citation.cfm?id=2392163.2392166`.

[49] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A general persistent code caching framework for dynamic binary translation (dbt). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 591–603, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-30-0. URL `http://dl.acm.org/citation.cfm?id=3026959.3027013`.

[50] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing cross-isa dbt through automatically learned translation rules. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 84–97, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3177160. URL `http://doi.acm.org/10.1145/3173162.3177160`.

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

[52] Tom Warren. Microsoft built an xbox 360 emulator to make games run on the xbox one, 2015. URL `https://www.theverge.com/2015/6/15/8785955/microsoft-xbox-one-xbox-360-emulator-software`.

[53] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, pages 68–79, New York, NY, USA, 1996. ACM. ISBN 0-89791-793-6. doi: 10.1145/233013.233025. URL `http://doi.acm.org/10.1145/233013.233025`.

[54] Chaohao Xu, Jianhui Li, Tao Bao, Yun Wang, and Bo Huang. Metadata driven memory optimizations in dynamic binary translator. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 148–157, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. doi: 10.1145/1254810.1254831. URL `http://doi.acm.org/10.1145/1254810.1254831`.

[55] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. Hermes: A fast cross-isa binary translator with post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 246–256, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. URL `http://dl.acm.org/citation.cfm?id=2738600.2738631`.