A Retargetable System-Level DBT Hypervisor

Citation for published version:

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 2019 USENIX Annual Technical Conference

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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 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.
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.

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

---

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

---
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 [11], 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.

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.

### Table 2: Optimizations applied in the offline stage.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Active in Opt. Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dead Code Elimination</td>
<td>O1–4</td>
</tr>
<tr>
<td>Unreachable Block Elimination</td>
<td>O1–4</td>
</tr>
<tr>
<td>Control Flow Simplification</td>
<td>O1–4</td>
</tr>
<tr>
<td>Block Merging</td>
<td>O1–4</td>
</tr>
<tr>
<td>Inlining</td>
<td>O1–4</td>
</tr>
<tr>
<td>Dead Variable Elimination</td>
<td>O1–4</td>
</tr>
<tr>
<td>Jump Threading</td>
<td>O2–4</td>
</tr>
<tr>
<td>Constant Folding</td>
<td>O3–4</td>
</tr>
<tr>
<td>Constant Propagation</td>
<td>O3–4</td>
</tr>
<tr>
<td>Value Propagation</td>
<td>O3–4</td>
</tr>
<tr>
<td>Load Coalescing</td>
<td>O3–4</td>
</tr>
<tr>
<td>Dead Write Elimination</td>
<td>O3–4</td>
</tr>
<tr>
<td>PHI Analysis</td>
<td>O4</td>
</tr>
<tr>
<td>PHI Elimination</td>
<td>O4</td>
</tr>
</tbody>
</table>

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

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
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.

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.
We support an adding significant overhead to the emulation of these instructions (invoked using the \texttt{syscall} instruction), the x86 port-based implementation, however, QEMU implements the \texttt{fmul} directly (lines 1—3), in much the same way as Captive, QEMU issues a function call for the floating-point multiplication (\texttt{fmul}). In contrast, Captive emits a host floating-point multiplication instruction, which operates directly on the guest register file.

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. For this reason we evaluate Captive against the widely used QEMU DBT as a baseline, supported by targeted micro-benchmarks and comparisons 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.

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. 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.

---

3 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.
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
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.

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

<table>
<thead>
<tr>
<th>DBT System (Year)</th>
<th>Guest ISA</th>
<th>Host ISA</th>
<th>Distinct Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>StarDBT [48] (2007)</td>
<td>IA-32, x86-64</td>
<td>IA-32, x86-64</td>
<td>Trace lengthening</td>
</tr>
<tr>
<td>ARCSim [11] (2011)</td>
<td>ARC/700</td>
<td>x86-64</td>
<td>Parallel JIT task farm</td>
</tr>
<tr>
<td>HQEMU [21] (2012)</td>
<td>ARMv7A</td>
<td>x86-64</td>
<td>Multi-threaded Compilation</td>
</tr>
<tr>
<td>Pydgin [29] (2015)</td>
<td>ARM/MIPS</td>
<td>x86-64</td>
<td>Meta-Tuning JIT Compiler</td>
</tr>
<tr>
<td>MAMBO-X64 [16] (2017)</td>
<td>AArch32</td>
<td>AArch64</td>
<td>Dyn. mapping of FP regs. Overflow address calculations Return address prediction</td>
</tr>
<tr>
<td>HyperMAMBO-X64 [15] (2017)</td>
<td>AArch32</td>
<td>AArch64</td>
<td>Hypervisor support</td>
</tr>
<tr>
<td>Pico [14] (2017)</td>
<td>x86-64, AArch64</td>
<td>x86-64, POWER8</td>
<td>Multi-core, multi-threaded DBT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Guest ISA</th>
<th>Host ISA</th>
<th>Static/Dynamic</th>
<th>User/System</th>
<th>Distinct Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xu et al. [54]</td>
<td>IA-32</td>
<td>IA-64</td>
<td>Dynamic</td>
<td>User</td>
<td>Compiler Metadata</td>
</tr>
<tr>
<td>Kedia and Bansal [25]</td>
<td>x86-64</td>
<td>x86-64</td>
<td>Dynamic</td>
<td>System</td>
<td>Kernel-level DBT</td>
</tr>
<tr>
<td>Spink et al. [39]</td>
<td>ARMv5T</td>
<td>x86-64</td>
<td>Dynamic</td>
<td>User</td>
<td>Support for Dual-ISA</td>
</tr>
<tr>
<td>Wang et al. [49]</td>
<td>IA-32</td>
<td>x86-64</td>
<td>Dynamic</td>
<td>User</td>
<td>Persistent code caching</td>
</tr>
<tr>
<td>Shigenobu et al. [36]</td>
<td>ARMv7A</td>
<td>LLVM-IR</td>
<td>Static</td>
<td>User</td>
<td>ARM-to-LLVM IR</td>
</tr>
<tr>
<td>Wang et al. [50]</td>
<td>ARMv6</td>
<td>x86-64</td>
<td>Dynamic</td>
<td>System</td>
<td>Learning of translation rules</td>
</tr>
<tr>
<td>Hong et al. [23]</td>
<td>ARM NEON</td>
<td>x86 AVX2/AVX-512</td>
<td>Dynamic</td>
<td>User</td>
<td>Short-SIMD to Long-SIMD</td>
</tr>
</tbody>
</table>
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 $2.17 \times$ 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 $1.68 \times$. This translates to a speed-up of $1.3 \times$ 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 $2 \times$ 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


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


