Accelerating Shared Library Execution in a DBT

Tom Spink
University of St Andrews
St Andrews, United Kingdom
tcs6@st-andrews.ac.uk

Björn Franke
University of Edinburgh
Edinburgh, United Kingdom
bfranke@inf.ed.ac.uk

Abstract

User-mode Dynamic Binary Translation (DBT) has recently received renewed interest, not least due to Apple's transition towards the Arm ISA, supported by a DBT compatibility layer for x86 legacy applications. While receiving praise for its performance, execution of legacy applications through Apple's Rosetta 2 technology still incurs a performance penalty when compared to direct host execution. A particular limitation of Rosetta 2 is that code is either executed exclusively as native Arm code, or as translated Arm code. In particular, mixed mode execution of native Arm code and translated code is not possible. This is a missed opportunity, especially in the case of shared libraries where both optimized x86 and Arm versions of the same library are available. In this paper, we develop mixed mode execution capabilities for shared libraries in a DBT system, eliminating the need to translate code where a highly optimised native version already exists. Our novel execution model intercepts calls to shared library functions in the DBT system and automatically redirects them to their faster host counterparts, making better use of the underlying host ISA. To ease the burden for the developer, we make use of an Interface Description Language (IDL) to capture library function signatures, from which relevant stubs and data marshalling code are generated automatically. We have implemented our novel mixed mode execution approach in the open-source Qemu DBT system, and demonstrate both ease of use and performance benefits for three popular libraries (standard C Math library, SQLite, and OpenSSL). Our evaluation confirms that with minimal developer effort, accelerated host execution of shared library functionality results in speedups between 2.7× and 6.3× on average, and up to 28× for x86 legacy applications on an Arm host system.

LCTES '24, June 24, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0616-5/24/06
https://doi.org/10.1145/3652032.3657565

CCS Concepts: • **Software and its engineering** → **Simulator / interpreter**; **Just-in-time compilers**; • **Hardware** → *Simulation and emulation.*

Keywords: Shared libraries, Dynamic Binary Translation, Instruction Set Simulation

ACM Reference Format:
Tom Spink and Björn Franke. 2024. Accelerating Shared Library Execution in a DBT. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24), June 24, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3652032.3657565

1 Introduction

Until recently, the number of Instruction Set Architectures (ISAs) in wide-spread use has been relatively stable, and computing platforms have generally invested in a particular one. It is clear that the x86 ISA dominates the desktop and server market, whereas Arm dominates the mobile and embedded space. A notable exception to this ISA stability is Apple, who have cycled through three different ISAs on their commodity computing platforms: PowerPC, x86, and now Arm. This change has necessitated a complete overhaul of many existing applications that were originally developed for the x86 ISA, but must now support the Arm ISA. To mitigate this, Apple have introduced a Dynamic Binary Translation (DBT) platform (Rosetta 2 [1]) that allows developers to gradually port their applications over to the Arm ISA.

This DBT platform is not without its limitations, however. In particular, there is no support for mixed-mode execution, i.e. mixing ISAs at runtime, and there are features missing in the emulated x86 guest ISA, such as x86 AVX instructions.

Whilst DBT systems are necessary to execute existing codebases on a different ISA, the translations are rarely as optimised as native code, and so software experiences major performance penalties. DBT can be avoided by porting the application to the new ISA, but for legacy codebases this would be difficult or even impossible if the source-code is not available.

A particular source of portability comes in the form of *shared libraries*, where reusable application code is developed for a particular purpose, and shared amongst all the applications that want to use it.

However, existing state-of-the-art DBT systems choose not to support this functionality directly, but instead delegate this job to the guest platform's *dynamic linker*. This

```c
#include <stdio.h>
#include <stdlib.h>

extern int fib(int n); // From shared library

int main(int argc, char **argv) {
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  printf("fib(%d) = %d\n", n, fib(n));
  return 0;
}
```

Figure 1. An application that uses a library implementation of the Fibonacci function to perform its main operation.

```c
int fib(int n) {
  return n < 2 ? n : fib(n-1) + fib(n-2);
}
```

Figure 2. An implementation of the `fib(n)` function in the shared library.

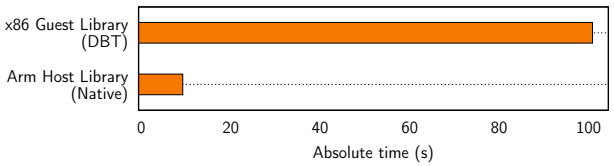


Figure 3. An illustration of the speed-up achievable when using a host native implementation of a shared library. Lower is better.

means that the DBT does not have to care about dealing with shared libraries, at the expense of having to translate the dynamic linker and the shared library itself throughout the execution of the main application.

By observing the fact that applications rely on common or popular shared libraries, there exists the opportunity to improve the performance of an application under DBT by transferring control to a *native* (host) version of a shared library, rather than translating and emulating a *guest* version.

In this paper, we overcome these challenges and performance issues by introducing a technique for supporting mixed-mode execution of a DBT system, where calls to shared library functions by the guest application are intercepted by the DBT runtime, and redirected to the respective native implementations.

1.1 Motivating Example

Consider the program in Figure 1. This program uses a shared library (`libfib`) that provides an implementation of the Fibonacci function (Figure 2). If this application were to be compiled for the x86 ISA, and ran through a DBT system (such as Qemu [3]) on an Arm host, the entirety of its execution (including an x86 version of a dynamic linker, and the corresponding x86 version of the shared library) would be translated and emulated by the DBT runtime. However, if `libfib` were available as a shared library on the host machine already compiled and optimized for the Arm architecture, then using this version would eliminate the overhead associated with translating and running the guest version.

Figure 3 shows that with this simple program, a speed-up of 10× can be achieved by invoking a native version of the library, vs. emulating it. This speed-up arises because the host native code is of much higher quality than the code generated by the DBT. For programs that are compute-bound in shared library execution, using host libraries (if available) can be highly effective.

Unfortunately, however, supporting this kind of mixed-mode execution in a DBT system is non-trivial, since typically the DBT runtime would be in complete control of the execution environment and so understand exactly how the execution of guest code progresses. But, not only does the runtime need to transfer control between *translated* code and *native* code, as soon as external code is used the runtime needs to make sure that the execution environment is correct. This effectively means that function parameters need to be translated correctly, and emulated resources (such as memory or file descriptors) are correctly managed.

1.2 High-level Overview

Figure 4 shows how the state-of-the-art DBT Qemu [3] supports shared library execution in the guest, and how our accelerated scheme changes the execution flow. Unmodified Qemu translates the entire guest program, including the application binary itself, the guest dynamic linker (`ld.so`), and the guest shared library. Qemu performs on-demand translation; when new (i.e. previously unseen) basic-blocks of guest code are encountered, they are immediately translated from the guest ISA to the host ISA, and stored in a *code cache* for future re-use. The translated block is then executed.

Our proposed execution scheme intercepts the guest program's call to the shared library function, and redirects it directly to the native host shared library's implementation. This means that the guest program is still translated as normal until a shared library call is detected, but instead of following the guest execution into the dynamic linker, the translation transfers control directly to the native implementation. Once the native function returns, control is transferred back to translated code.

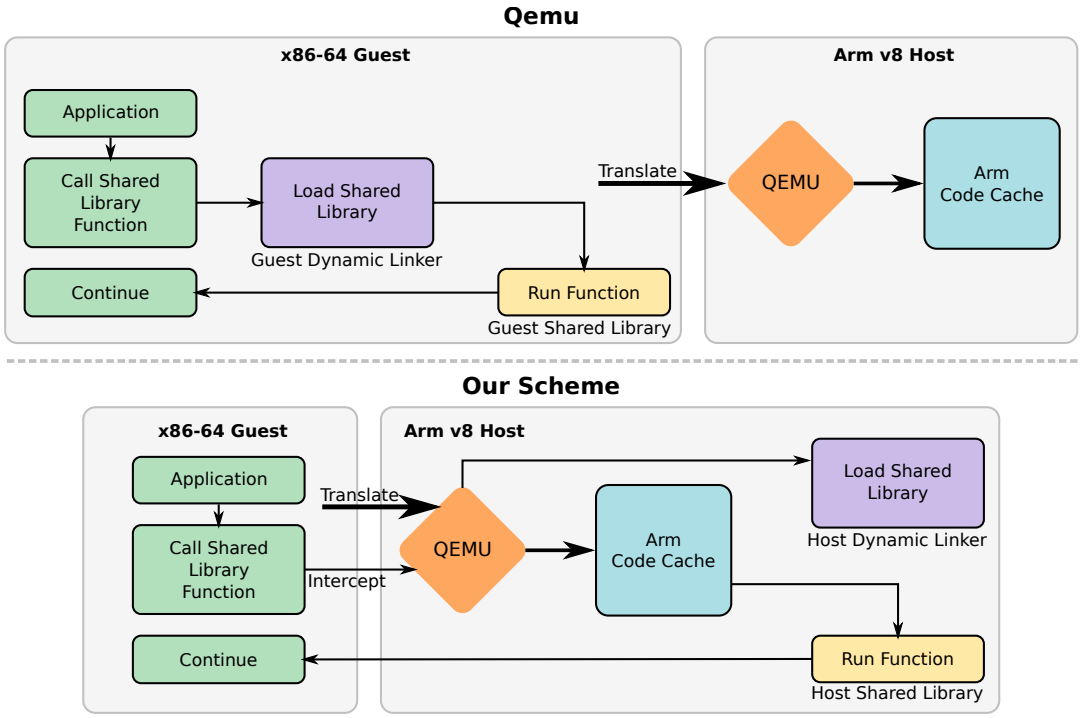


Figure 4. A high-level overview of how QEMU handles shared libraries in guest applications, and how our scheme accelerates them.

The need for accelerating shared library calls in a DBT system has already been recognized, notably by the `box86/64` [8, 9] DBT system. They term their approach "wrapped libraries", and boast impressive speed-ups that they claim arise due to this scheme. However, the approach that `box86/64` takes requires hard-coding a "glue" layer for every supported library and function within, whereas we strive for automation and minimal developer effort.

2 Background on Shared Libraries

A shared library is an external piece of software that supports the execution of an application. When a program that uses shared libraries is compiled, it will be *linked* to a particular version of the library, and when the program is executed a *dynamic linker* will load the shared library into the same address space, and resolve any calls by the main application to shared library functions. In this paper, we will focus on how shared libraries are implemented on platforms that use the Executable and Linkable Format (ELF) format.

With the ELF standard, invoking a shared library function call involves a certain amount of indirection in the main application binary. This is implemented through a *Procedure Linkage Table (PLT)* and *Global Offset Table (GOT)*. The PLT is used as a trampoline from the main application code into the shared library code, and together with the GOT contains information to instruct the dynamic linker on how to resolve the function call. Typically, shared library functions are resolved lazily, i.e. they will only be resolved on first invocation.

In the application binary, the `.plt` section contains entries for each of the shared library functions in use, as well as a special *resolver* entry. These entries are small sequences of code that read an entry from the `.got` section, and jump to that location. Initially, the `.got` entry contains the address of an instruction that transfers control to the *resolver* `.plt` entry, which in turn transfers control to the dynamic linker. Upon first invocation, the dynamic linker will load the shared library, set-up relocations, and modify the `.got` entry to point directly to the resolved function, so that future invocations bypass the dynamic linker and jump directly into the corresponding code. Figure 5 shows how this works in practice.

3 Supporting Shared Library Calls

Since the natural interface between an application and a shared library is a function call, this is the boundary at which our scheme operates. The **key idea** is to capture the point at which a guest program makes a call into a shared library, and instead of emulating the guest dynamic linker and guest shared library, use the existing host shared library directly.

Pure shared library functions (i.e. those that only operate on their input arguments) are the easiest to handle, as at call time we can simply copy the value of the function arguments from the emulated guest representation into the host

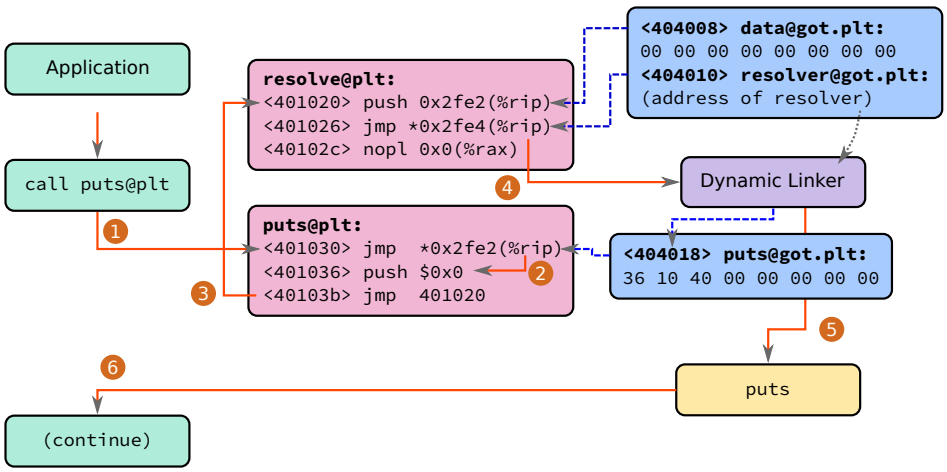


Figure 5. An application begins the call sequence for a shared library function (`puts`), and at (1) control is transferred to the PLT entry corresponding to this function. The function offset is read from the GOT, which at (2) currently points to the next instruction in the PLT (address 0x401036). Then, at (3) control is transferred to the resolver function, which extracts further information from the GOT and invokes the dynamic linker at (4). The dynamic linker writes back the address of the `puts` function into the GOT, and at (5) transfers control to the routine. Finally, when `puts` finishes, control is returned at (6) to the main application. Solid lines indicate control-flow, and dashed lines indicate data-flow.

representation, and then perform the call directly. More complex functions may not operate in isolation, and will require additional functions to be accelerated. Both of these cases, however, are supported by our proposed scheme.

The primary concern is being able to determine what a function signature looks like, so that the correct values can be transferred from guest to host. Function signatures are not defined in the instruction set, nor do they (normally) exist as metadata within the application binary, therefore we introduce a C-like Interface Description Language (IDL) to describe the functions that should be accelerated. We call this description file a **Library Interface Definition (LID)** file, and it contains the information required by the DBT runtime to transfer function arguments from guest to host.

The LID Interface Description Language. An IDL is necessary for our scheme because of how function call argument *marshalling* must be performed. It is not immediately clear from inspecting the guest machine code what a particular function's signature might look like, and in some cases it is impossible to determine the semantics of a particular parameter, e.g. is the parameter just a plain integer, or a function pointer? This is an important distinction, because even though the underlying *type* is the same, the *semantics* associated with that parameter cause the value to be marshalled differently.

To overcome this, we introduce LID files that are used to describe the signature of the functions that are to be accelerated. The signature encodes both *type* and *semantic* information that can be used by the DBT runtime to automatically generate the appropriate marshalling code. Figure 6 shows an example LID file that defines a range of shared library functions.

```
1  library "libc.so";
2
3  # String utilities
4  si32 puts([string] const ui8 *s);
5  void *memcpy([size=n] void *dest, [size=n]
       const void *src, ui64 n);
6
7  # System utilities
8  si32 close([fd] si32 fd);
9
10 # Maths
11 library "libm.so";
12 f64 exp(f64 v);
13 f64 log(f64 v);
```

Figure 6. An example of how shared library functions are defined in a LID file. Function definitions closely mirror C-style function definitions, but notably *attributes* (in square brackets) indicate the semantic properties of a parameter.

A function definition allows attributes to be placed on parameters to indicate the semantic behavior of that argument (see Lines 4, 5, and 8 in Figure 6 for examples). These semantics alter how the underlying type is transferred from guest to host, by possibly mapping values between two different domains. The types in use are also more explicit than C types; Table 1 gives an overview of the types available.

Table 1. Types available for function definitions.

Type Name	C equivalent	Description
`void`	`void`	No type
`ui1`	`_Bool`	Boolean
`ui8, si8`	`char`	(Un)signed 8-bit integer
`ui16, si16`	`short`	(Un)signed 16-bit integer
`ui32, si32`	`int`	(Un)signed 32-bit integer
`ui64, si64`	`long long`	(Un)signed 64-bit integer
`uword, sword`	`long`	(Un)signed word size
`f32, f64`	`float/double`	32/64 -bit floating-point
`fcomplex`	`_Complex`	Floating-point complex number
`struct`	`struct`	Variable sized composite type
`struct(N)`	`struct`	N-byte sized composite type
`*`	`*`	Pointer notation: `void *`
`const`	`const`	Const modifier

Each parameter is given a name, and although named parameters serve no purpose at marshalling time, they are important in the LID file for (a) readability, and (b) referencing other parameters in semantic attributes.

For example, a function may operate on a range of memory, and in order to instruct the marshalling code the size of that memory, the value may be taken from another parameter. See line 5 in Figure 6 for an example of this.

4 Implementation

We implement our scheme in the state-of-the-art DBT system Qemu [3]. When Qemu starts, the LID file is parsed, and each function definition is stored in a *function mapping table*. The corresponding native shared library is also loaded (with `dlopen`), and a pointer to the function is resolved (with `dlsym`). The function mapping table entry contains the following information: The name of the function (in string form), an in-memory representation of the function signature as defined in the LID file, and a pointer to the native implementation of the function (from `dlsym`).

4.1 ELF Parsing

After the function mapping table has been populated, our modifications to the built-in ELF loader enable us to figure out which `.plt` entries correspond to which shared library functions. This is a complex operation that requires multiple structures from the ELF file, as well as decoding some host machine instructions, to figure out the name of the function that a `.plt` entry actually points to.

Once this information has been determined, the name of the function is checked against the mapping table to see if there is a corresponding mapping table entry. If no such mapping exists, i.e. no function names matched, then it was not present in the LID file and will not be accelerated. If the mapping table entry was found, an entry in the *translation hook table* is inserted that maps the virtual address of the `.plt` entry, to the corresponding function mapping table entry.

4.2 Call Detection

Since Qemu compiles basic-blocks on-demand, there exists a point in time when a block of code is not yet compiled, i.e. the first invocation of that block (or if the code cache has been invalidated). Taking advantage of this, we can hook into Qemu's translation routines, and determine which virtual address is about to be translated. A `.plt` entry will *always* be the start of a basic-block, because they are invoked directly from the guest application by a call instruction.

Before a basic-block is translated, we look up the virtual address of the start of the block in the translation hook table. If an entry exists, instead of translating the basic-block as usual, we generate specialized code that performs the actual transfer of control-flow to the native library. Like a regular translation, the translated code is cached so that the marshalling code will always be used. If the code cache is cleared for whatever reason, the `.plt` entry will simply be re-translated in this manner again.

It is important to note that the actual contents of the `.plt` entry are never actually used in this case. Instead, we simply use the address of the `.plt` entry as a handy mechanism for detecting calls to shared libraries.

4.3 Generation

Once we have determined that we are translating a call to a shared library function, we use Qemu's existing code generator infrastructure (Tiny Code Generator (TCG)) to generate the marshalling code. This generation is specialized for each *guest* architecture, as it needs to implement the argument marshalling for the guest calling convention. The generation remains *host* architecture agnostic due to the use of the existing TCG function call infrastructure.

5 Marshalling

Marshalling is the process of translating the parameter and return values of function calls (at runtime) from the semantics of the guest machine to the semantics of the host machine. In general, this is mapping the function call Application Binary Interface (ABI) from the guest to the host by moving function arguments from guest registers into host registers (or the equivalent storage mechanism, e.g. stack). However, in addition to copying data around, the underlying semantics of these values needs to be taken into account. For example, a file descriptor is normally represented as a simple integer value, but the DBT system may maintain an internal mapping table to provide the guest with an isolated, *virtual* view of file descriptors.

5.1 Calling Convention

In order to correctly prepare the activation record for a particular function, the calling convention in use by both the guest and the host needs to be known. This is so that the

Table 2. A simplified mapping of integer function arguments for x86 System V and Arm PCS calling conventions.

Purpose	x86-64 (System V)	Arm (PCS)
Ret. Val.	`rax`	`x0`
Arg 1	`rdi`	`x0`
Arg 2	`rsi`	`x1`
Arg 3	`rdx`	`x2`
Arg 4	`rcx`	`x3`
Arg 5	`r8`	`x4`
Arg 6	`r9`	`x5`

function arguments can be transferred from the emulated guest register state, into the host's native register state.

In our example, for x86-64 we are using the standard System V ABI [7], and for Arm we are using the Procedure Call Standard for the Arm 64-bit Architecture [2]. Table 2 shows a simplified mapping of x86 function arguments to Arm function arguments, for integer types.

In the case of the x86 guest, the first six (integer) arguments are passed in registers `rdi, rsi, rdx, rcx, r8, r9`. The return value is passed back in `rax`. For the Arm host, the first six (integer) arguments are passed in registers `x0, x1, x2, x3, x4, x5` and the return value is passed back in `x0`. The situation becomes more complex when floating-point registers are in use, as interleavings of integer and float registers are used. Furthermore, as the number of parameters exceeds the number of registers available, values are spilled to the stack.

Once a call to a shared library has been detected, marshalling code is generated to move the function arguments from the representation of the guest registers, into the host registers, and a call directly into the native library is emitted. After the call instruction, code is emitted to marshal the return value into guest registers. This technique works for platforms that dictate argument passing on the stack too - instead of copying from guest register state to host register state, values are taken from the guest stack and marshalled into the appropriate locations on the host.

5.2 Primitive Types

Primitive types are regular integer or floating-point types, whose values can be directly copied from the guest machine to the host machine. For example, consider the following C function signature:

```
int test(int a, float b, int c);
```

For an x86-to-Arm translation, the following mapping will be used:

```
a:    arm host: x0  ← x86 guest: rdi
b:    arm host: q0  ← x86 guest: xmm0
c:    arm host: x1  ← x86 guest: rsi
ret: x86 guest: rax ← arm host: x0
```

Then Qemu will generate following host instructions:

```
ldr  x0, [x19, #0x38]  ; a: x0 ← rdi
ldr  q0, [x19, #0x320] ; b: q0 ← xmm0
ldr  x1, [x19, #0x30]  ; c: x1 ← rsi
adrp x30, #0xfffff7fbe000
add  x30, x30, #0x640
blr  x30    ; call native test function
str  x0, [x19]       ; ret: rax ← x0
```

5.3 Memory Pointers

Memory addresses are simply integer values, and so are marshalled as a plain value. But, the actual pointers themselves may not be valid host/guest pointers if the DBT system employs a memory mapping scheme.

Fortunately, in the case of user-mode DBT in Qemu, we can take advantage of the fact that on 64-bit host systems both the host and guest memory spaces are shared, and so memory pointers generated by either system *are* interchangeable. If the situation is different, and translations between host/guest memory addresses are required, then the appropriate mapping would happen during argument and return value marshalling as calls to the appropriate DBT runtime helpers.

5.4 Strings

Strings are essentially normal memory addresses, however we can assume that they behave in particular ways - namely strings are `NULL`-terminated. This enables us to compute the length of the string through normal string length counting methods, should this be required for memory pointer marshalling purposes.

5.5 File descriptors

File descriptors are generated by system calls, and are managed by the DBT runtime as a mapping from guest to host. For example, if a guest program wishes to open a file (using the open system call), the request is sanitized and passed directly through to the host operating system's system call. However, the resulting file descriptor number (which is typically a 32-bit integer) may be translated to an opaque guest value. This is to prevent the guest program from interacting with file descriptors that exist in the DBT's namespace, but to which the guest program should not be able to access.

Qemu does not perform any mapping of file descriptors, so it is safe to treat them as simple integers in our case. This also removes the complexity of mapping file descriptors that may be presented indirectly to shared library functions, e.g. through opaque structures.

5.6 Function Pointers

One of the more complex marshalling operations are function pointers, as they present a control-flow target to host code, which if invoked directly by native code would cause

the host machine to jump to raw (untranslated) guest instructions — almost certainly causing an illegal instruction exception. Currently, our prototype implementation does not yet support marshalling function pointers from guest to host, although the current (manual) solution is to generate a "trampoline" for the particular function pointer argument, which will transition from native code back into emulated guest mode for the duration of the function.

5.7 Limitations: Variadic Function Parameters

One particular edge case of function argument marshalling is when a function employs variadic arguments. The most common examples of these are the `printf` and `scanf` (and associated) routines.

In general, this cannot be solved perfectly, because it is not possible to determine from the guest binary which arguments are active. Worse, since it is not known what *types* of arguments are active either, it is not possible to know whether arguments are passed in general purpose registers, or floating-point registers. The final issue is that in the x86 ABI, if the number of arguments exceeds the number of dedicated argument registers, the stack is used.

Our current implementation does not support host execution of functions with variadic parameters, but such functions are translated and executed as guest code.

6 Application Pre-processing

Running a guest application with host shared libraries requires an offline step to determine which functions should be accelerated, and to write the function definitions in the LID file. This involves identifying the libraries involved, selecting the functions necessary to support acceleration, and inputting the function signatures into a LID file.

Given that function signature recovery can be performed by static analysis, it is possible to automate the entire process of generating a LID file, by simply analyzing the program binary to determine the shared library functions in use, then using a suitable binary lifting tool to generate the function signature definition.

Once this information is encoded in the LID file, the preparation is complete, and the guest program can be executed with accelerated function calls.This process is currently manual, however we anticipate that LID files could be automatically generated statically from shared library header files, by using a parsing framework (such as that available with Clang) to extract function definitions.

7 Evaluation

To evaluate our system, we use an Armv8-A host machine described in Table 3, running both an unmodified and our modified version of Qemu [3] (based on version 5.2.90). The emulated guest machine is the default x86-64 platform built-in to Qemu.

Table 3. Host machine used for experiments.

Make	Gigabyte R181-T92-00		
Model	Dual Cavium ThunderX2 CN9980		
Architecture	Armv8-A		
# Processors	2	**Frequency**	2.2 GHz
# Cores	64	**# Threads**	256
Memory	256 Gb	**L1 I/D$**	1 Mb / 1Mb
L2$	8 Mb	**L3$**	32 Mb

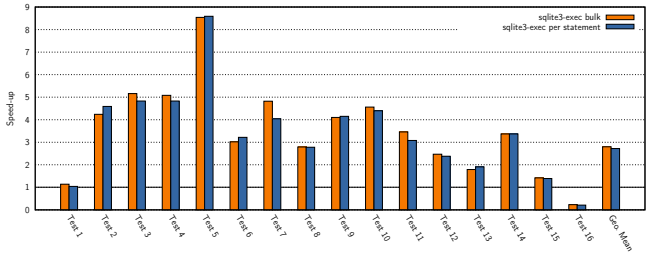


Figure 7. Speed-up of the `sqlite3` benchmarks when using our execution scheme, vs. vanilla Qemu. Two styles of invocation were used: **bulk**; where one shared library call was issued with all SQL statements, and **per statement**; where a shared library call was issued for each SQL statement.

In order to make meaningful evaluations of our execution scheme, we choose two popular open-source projects that provide a shared library to the application developer for implementing functionality: `sqlite3` for database functions, and `openssl` for cryptography functions.

7.1 SQLite

`sqlite3` [6] is a library implementation of a relational database management system, which is designed to be embedded directly into a program. In these experiments, we use the speed testing scheme as described on the official `sqlite3` website (https://www.sqlite.org/speed.html).

Each test includes a series of SQL statements, and so we took two approaches to executing the tests:

1. **Bulk execution:** one call to `sqlite3_exec` with a single string containing all SQL statements.
2. **Per statement execution:** a call to `sqlite3_exec` for *each* statement.

Figure 7 shows that nearly every test (except Test 16) performs better when using our execution scheme. The average speed-up of the benchmark is 2.8×, and we achieve a maximum speed-up of 8.54× in Test 5. The difference in performance between **bulk** and **per statement** execution is minimal, with an average overhead of executing **per statement** of 3%. This shows that any overhead introduced through the marshalling code does not significantly affect performance.

The two notable exceptions here are Test 1, where there is little or no performance improvement, and Test 16 where

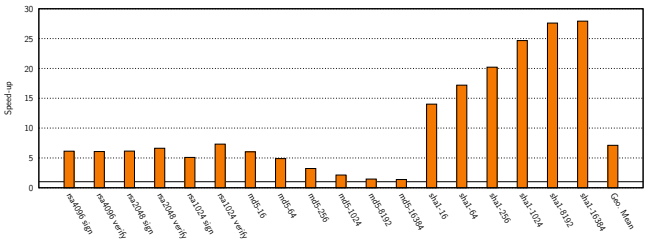


Figure 8. Speed-up of certain `openssl` algorithms when using our accelerated execution scheme, vs. unmodified QEMU.

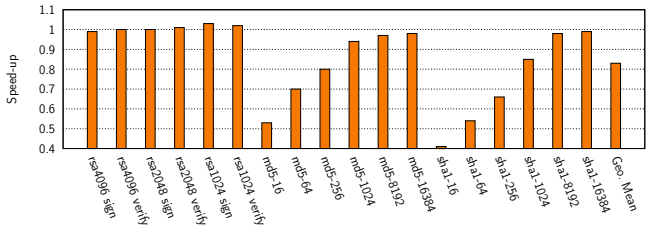


Figure 9. Speed-up of certain `openssl` algorithms when using our accelerated execution scheme, vs. native execution on the host machine.

there is a performance degradation. To better understand this result, we also ran the benchmark natively on the host machine and observed that in fact our scheme is operating almost exactly at native speed. This means that QEMU is actually running this test *faster* than the native library. Both tests are primarily I/O-bound, and so it appears that QEMU's interaction with the OS inadvertently outperforms the native I/O performance for these two specific benchmarks.

7.2 OpenSSL

`openssl` [13] is a popular cryptography library that implements a wide range of cryptographic algorithms, primarily targeted at securing communications.

For this experiment, we use the x86 compiled version of the `openssl` application with its built-in speed tests, running it through both unmodified QEMU, and QEMU with our accelerated execution scheme. The benchmarks are designed to run for a predetermined amount of time, and measure the rate of operations per second, i.e. the throughput.

Figure 8 shows that for the built-in speed tests, every operation performs better when using our execution scheme. Across these speed tests, on average there is a 6.26× speed-up, but interestingly the characteristics of the speed-up differ by class of algorithm. On the left of the graph, are the RSA algorithms (with key sizes ranging from 1024-bit to 4096-bit), all of which show a consistent speed-up of around 6.2×. However, the `md5` and `sha1` blocks yield a different insight. `md5` shows a steady decrease in speed-up, as the block size increases, whilst `sha1` shows the opposite — a steady increase in speed-up as the block size increases. This behavior can be attributed to the characteristics of the particular algorithms, which we will discuss in subsection 7.4.

7.3 Comparison against native execution

Since the native host shared library is now being used directly by the guest program, we also measure the performance of the native host application, i.e. the native Arm version of the `openssl` application, to see how our scheme compares to a fully native run. Figure 9 shows that on average we are within 20% of native execution, and routines such as RSA are effectively executing at native performance.

The trend of the `md5` and `sha1` benchmarks in these results are to approach native performance as the block size increases. This is due to the fact that the benchmarks are running in native code for longer, as there is more data to process. Our execution scheme performs best when the majority of execution takes place in highly optimized native code, as this is where the gains are made, vs. poorly optimized translated code.

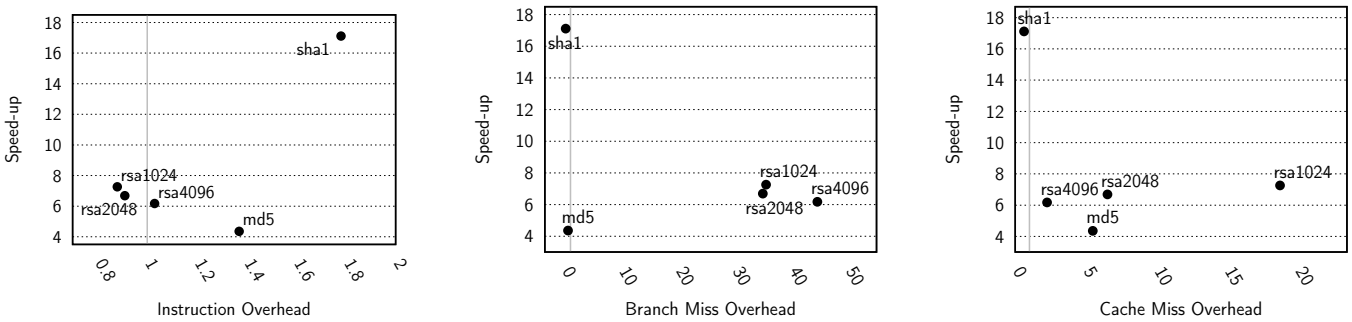
7.4 Contributing Factors for Speed-up

In this section, we investigate what factors influence the observed speed-ups, and focus on the `openssl` benchmark.

We consider three major factors that influence how the speed-up is achieved: *instruction count*, *branch misses*, and *cache misses*. We choose these metrics as typically native code is shorter and denser than translated code, so instruction count plays a large role in obtaining performance improvements. QEMU translates on a basic-block granularity, and chains executions of these blocks to build the control-flow. Furthermore, each translated block involves an interrupt check in its header, which results in additional control-flow. This results in very branch-heavy host machine code, and so exercises the branch predictor quite hard. Finally, through the very nature of DBT, additional data accesses are required for maintaining the execution state of the guest (e.g. the guest register state). The interleavings of these accesses with *real* data accesses creates a significant amount of cache pressure.

Figure 10a plots the *instruction count overhead*, i.e. the ratio of instruction count in the original scheme vs. our accelerated scheme, where a value > 1 indicates that *more* instructions are executed by the original scheme. Figure 10b plots the *branch miss overhead*, where a value > 1 indicates that the branch miss ratio is higher for the original scheme. Similarly, Figure 10c plots the *cache miss overhead*, where a value > 1 indicates that the cache miss ratio is higher for the original scheme.

These graphs show that there is not one particular factor that contributes to the speed-up, but is a combination of all three characteristics.



(a) Speed-up vs. Instruction Overhead. Points to the right of the graph indicate a reduction in executed instructions for our accelerated scheme.

(b) Speed-up vs. Branch Miss Overhead. Points to the right of the graph indicate a reduction in branch misses for our accelerated scheme.

(c) Speed-up vs. Cache Miss Overhead. Points to the right of the graph indicate a reduction in cache misses for our accelerated scheme.

Figure 10. An evaluation of the factors contributing towards the observed speed-up. Gray lines are plotted at 1, to identify zero overhead. Values > 1 on the *X*-axis indicate that the original scheme incurs overhead vs. our accelerated scheme.

sha1. The `sha1` benchmark achieves its performance solely from a reduction in executed instruction count. This is because it reduces the number of executed instructions by 78%, whilst incurring slightly more overhead from branch and cache misses.

md5. The `md5` benchmark reduces executed instructions by 37%, and also reduces cache misses by 5.49×, thus the speed-up arising from a combination of more streamlined instruction execution, and improved cache performance.

rsa{1024,2048,4096}. The `rsa` benchmarks show a slight increase in the number of executed instructions, but they all show significant gains in reducing the branch miss overhead, with up to a 44.5× reduction in branch misses. Furthermore, they show between 2.2× and 18.7× improvements in reducing cache misses. As noted previously the benchmarks run for a fixed amount of time, and so these reductions in branch and cache misses account for why more instructions can be executed in the same amount of time — there are fewer CPU cycles spent on dealing with cache and branch misses, and more cycles spent running instructions pertaining to the algorithm, hence increasing throughput.

7.5 Comparison to *box64*

`box{86,64}` [8, 9] is an open-source emulator, which implements a native host library "wrapping" technique. Similar to Tan *et al.* [12], the approach taken by `box{86,64}` requires hard-coding a "glue" layer that contains the code necessary to support guest-to-host transitions and argument marshalling, whereas in our scheme this interface layer is automatically generated from an interface description.

We compare our implementation of the shared library acceleration scheme in Qemu against the `box64` implementation, by running a compression benchmark relying on the zlib compression algorithm provided by the `zlib` library.

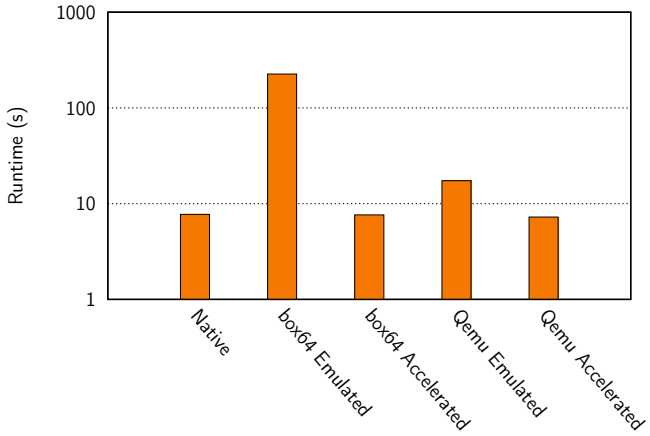


Figure 11. A comparison of the performance of a compression benchmark (`libz`) between the box64 emulator, and Qemu. The *emulated* version is where the guest shared library is translated by the DBT, and the *accelerated* version is where the host shared library is used directly. Lower is better.

For both `box64` and Qemu we run the compression benchmark in both emulated and accelerated mode, and compare performance.

Figure 11 shows that both `box64` and our scheme perform as good as native execution of the benchmark. This is to be expected for programs that spend the majority of their time in native code, and shows that our approach can achieve the same performance as a hand-crafted implementation.

Native execution speed is the maximum possible execution speed, and so this is the performance limit for any shared-library acceleration strategy. Since both `box64` and our modified Qemu achieve native performance, we show that our "user friendly" IDL-based strategy does not introduce any runtime overhead.

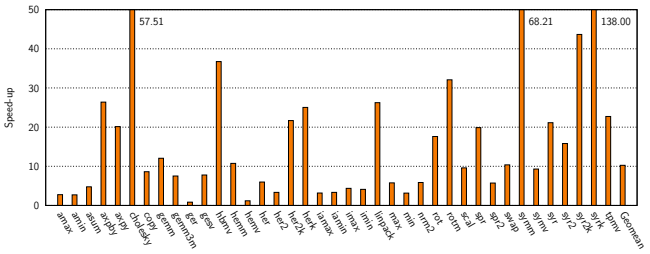


Figure 12. Speed-up of the OpenBLAS benchmarks.

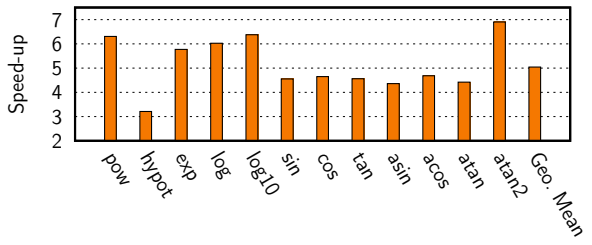


Figure 13. Speed-up of floating-point operations when using the host `libm.so` math library.

7.6 Microbenchmarks

In this section, we drill-down into the performance and behavior of our scheme by looking at a series of microbenchmarks, that are designed to exercise the actual mechanism of calling shared libraries.

7.6.1 OpenBLAS. OpenBLAS [14] is a framework for linear algebra, and implements the BLAS and LAPACK APIs.

Figure 12 shows that in all cases, our accelerated scheme outperforms Qemu and on average achieves a speed-up of 10.26×, with the greatest speed-up of 138× from the `syrk` benchmark. These results reinforce the claim that our system performs best when the native library function runs for a long time, in particular the complex BLAS operations achieve the greatest speed-up, because they spend most of their time executing in the kernel function.

7.6.2 Floating-point Performance. In Qemu, FP operations are emulated by emitting a function call that transfers control into the Qemu runtime to perform the operation. This results in well-known severe performance implications [3, 5, 10] for floating-point heavy code.

In guest programs, regular floating-point arithmetic operations (such as add, subtract, multiply) are implemented directly as machine instructions (e.g. `addss`, `subss`, `mulss` on x86), which are then translated by Qemu. However, some routines (such as the trig. functions) are calls to library functions, giving us the opportunity to intercept these library calls, and use the host implementation, rather than using the emulated implementation.

In this experiment, we target some common functions from the Standard C math library (`libm`), and Figure 13 shows that in all cases, using our accelerated scheme improves the performance of these functions significantly.

8 Related Work

Microsoft employ binary translation for supporting x86 applications on the Arm ports of the Windows OS. This technology is part of the Windows-on-Windows (WoW) emulation layer (originally for supporting x86-32 applications on x86-64 hosts), and has been extended to support x86 applications on Arm hosts. In this technology, Microsoft identified the need for mixed-mode execution, and have chosen to ease mode switches by supporting an ABI that is directly compatible on both the host and guest platforms [11].

Tan *et al.* [12] take a similar approach to our scheme, except they use Qemu's built-in helper function infrastructure, adding an additional layer of indirection through a nested function call, and requiring the developer to hard-code helper implementations that perform argument marshalling directly into Qemu's source-code. This approach requires a recompilation of Qemu every time a new function is added.

An older system, "DIGITAL FX!32" [4] pioneered the DBT movement by introducing a dynamic binary translator for running x86 Windows NT applications on Windows NT for the Alpha architecture. Their implementation describes so-called "jackets", which are applied at program load time by modifying the COFF Import Address Table (IAT) to redirect library calls to the host native counterpart. The "jacket" is the code necessary to perform function argument marshalling between the two calling conventions. While conceptually similar to our approach, their implementation relies on handling an illegal x86 instruction to transition into native code which adds an amount of overhead, whereas our scheme generates efficient transition code directly.

9 Summary & Conclusions

In this paper we have shown that there exists both opportunity and demand for accelerating dynamic binary translation, and that significant gains can be made by redirecting calls to shared library functions in guest code to native host code. We introduce an interface description language for the description of library APIs, which are then processed to automatically generate a "glue" layer for interfacing between library function calls and host-native implementations. We implement this scheme in the Qemu user-mode DBT system, and demonstrate that with minimal developer effort, average speed-ups of 2.8× and 6.26× can be achieved, with over 25× on some individual benchmarks. We show that our scheme introduces minimal runtime overhead, and in the best case can operate at native (i.e. untranslated) speeds.

We plan to eventually eliminate the need for manual interface description, and extract the relevant function signature information through fully automated binary code analysis and ML guided inference.

References

[1] Apple. 2021. About the Rosetta Translation Environment. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment

[2] Arm. 2021. Procedure Call Standard for the Arm 64-bit Architecture (AArch64). https://github.com/ARM-software/abi-aa/releases/download/2021Q1/aapcs64.pdf

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[4] Anton Chernoff and Ray Hookway. 1997. {DIGITAL} FX! 32-Running 32-Bit x86 Applications on Alpha {NT}. In *Large-Scale System Administration of Windows {NT} Workshop (Large-Scale System Administration of Windows {NT} Workshop)*.

[5] Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. 2015. Improving SIMD code generation in QEMU. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1233–1236.

[6] Richard D Hipp. 2020. SQLite. https://www.sqlite.org/index.html

[7] H.J. Lu, Michael Matz, Girkar Milind, Jan Hubi, Andreas Jaeger, and Mark Mitchell. [n. d.]. System V Application Binary Interface AMD64 Architecture Processor Supplement. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf

[8] ptitSeb. 2021. box64. https://github.com/ptitSeb/box64.

[9] ptitSeb. 2021. box86. https://github.com/ptitSeb/box86.

[10] Qiang Shi and RongCai Zhao. 2016. Floating Point Optimization Based on Binary Translation System QEMU. In *2016 2nd Workshop on Advanced Research and Technology in Industry Applications (WARTIA-16)*. Atlantis Press, 1338–1343.

[11] Marc Sweetgall. 2021. Announcing ARM64EC: Building Native and Interoperable Apps for Windows 11 on ARM. https://blogs.windows.com/windowsdeveloper/2021/06/28/announcing-arm64ec-building-native-and-interoperable-apps-for-windows-11-on-arm/.

[12] Jie Tan, Jian-min Pang, and Shuai-bing Lu. 2018. Using Local Library Function in Binary Translation. In *Current Trends in Computer Science and Mechanical Automation Vol. 1*. De Gruyter Open Poland, 123–132.

[13] The OpenSSL Project. 2003. OpenSSL: The Open Source toolkit for SSL/TLS. (April 2003). www.openssl.org.

[14] Zhang Xianyi and Martin Kroeker. 2021. OpenBLAS. http://www.openblas.net/

Received 2024-02-29; accepted 2024-04-01