

Code
AvailableCode
ReviewedCode
Reproducible

coMulator: Coordinate Cross-architecture Compilation and Emulation to Accelerate Dynamic Binary Translation

Yuhao Gu, Zhongchun Zheng, Nong Xiao*, Yutong Lu, and Xianwei Zhang*

School of Computer Science and Engineering, Sun Yat-sen University

Guangzhou, China

{guyh9, zhengzhch3}@mail2.sysu.edu.cn, {xiaon6, luyutong, zhangxw79}@mail.sysu.edu.cn

ORCID: Y. Gu (0009-0009-8761-1237), Z. Zheng (0009-0002-8176-8486),

N. Xiao (0000-0002-2166-977X), Y. Lu (0000-0001-5315-3375), X. Zhang (0000-0003-3507-4299)

Abstract—The rapid proliferation of diverse instruction set architectures (ISAs), such as x86, Arm, and RISC-V, has made cross-ISA execution a fundamental requirement for software portability and platform migration. Although dynamic binary translation (DBT) is the *de facto* solution, it incurs substantial performance overhead due to runtime interpretation and translation costs. Cross-compiling to the host ISA can avoid this penalty, but existing toolchains suffer from a binary dilemma, where any non-portable code or closed-source dependency forces full fallback to DBT. Recent hybrid approaches mitigate this by enabling partial native execution via function offloading or emulation-specific ABIs, but they remain limited: offloading requires manual engineering and cannot handle guest callbacks, while ABI-specific designs introduce tight coupling that reduces portability and scalability across ISA pairs.

To bridge this critical gap, this paper presents coMulator, a source-guided, callback-capable, and ABI-bridging coordinated design that aims for dual generality across both diverse codebases and various guest/host ISA combinations. At its core, coMulator introduces *gyhcall* (*guest yields host calls*), an innovative target-neutral invocation mechanism that leverages compile-time assistance to fully automate function offloading while preserving ABI neutrality. Mechanistically, *gyhcall* bridges guest/host ABI structural discrepancies via collaborative stubs and handles complex control flows of guest callbacks through runtime emulation reentrancy. To further mitigate the boundary-crossing overhead inherent in *gyhcall*, coMulator incorporates four critical optimizations to eliminate unnecessary switches. We evaluate coMulator using the SPEC CPU 2017 benchmarks and real-world applications across nine guest/host combinations among x86_64, aarch64, and riscv64. The results demonstrate that coMulator consistently achieves an average speedup of over 7.90 $\times$, restoring over 75% of native execution performance. Further comparisons against representative state-of-the-arts including Risotto and Arm64EC demonstrates coMulator's practicability and scalability for accelerating cross-ISA execution.

Index Terms—binary translation, emulation, cross-compiling, function offloading.

I. INTRODUCTION

Against the backdrop of the slowing Moore's Law [1], [2], the computing landscape has witnessed a rapid proliferation of diverse ISAs (such as x86, Arm, and RISC-V), making

cross-ISA execution a fundamental requirement for software portability and platform migration. Although dynamic binary translation (DBT) serves as the *de facto* solution—exemplified by Rosetta 2 [3] and Prism [4] to support legacy applications [5]—it consistently suffers from severe performance degradation. Because translating a single guest¹ instruction requires multiple host instructions, emulation is often dozens of times slower than native execution [6]–[8]. Paradoxically, since most applications are compiled from high-level, target-independent source code, cross-compiling² the codebase directly to the host ISA should theoretically eliminate DBT overhead. Yet, existing toolchains trap developers in a binary dilemma, where a single compilation failure forces the entire program to fall back to full DBT. In practice, applications frequently depend on non-portable code fragments (e.g., assembly, intrinsics) or low-level ISA-specific libraries; a commercial example arises when upstream vendors refuse to build their proprietary, closed-source libraries for a new target ISA, forcing developers to rely entirely on full DBT emulation, thereby incurring substantial performance overheads.

By exploiting the partial portability inherent in high-level source code, hybrid guest/host execution can significantly accelerate emulation via native execution. This idea has motivated two paradigms of existing works. The first is *binary-level function offloading* [9]–[13], which redirects guest calls to semantically equivalent host libraries via dedicated wrappers. For instance, a representative state-of-the-art framework, Risotto [12], employs a custom Interface Definition Language (IDL) to declare function signatures and offload host-native shared libraries. However, these binary-level techniques fail to handle guest callbacks from the host side, which, coupled with substantial manual engineering efforts, confines their

¹Throughout this paper, we use "guest" to refer to the side of the program being emulated, and "host" to denote the side of the native system that performs the emulation.

²Conventionally, cross-compilation refers to compiling and generating guest-ISA programs on host-ISA machines. Herein, we generalize its meaning to denote building project source code originally written for a legacy ISA into binaries targeting a different, new ISA.

*Xianwei Zhang and Nong Xiao are the corresponding authors.

applicability strictly to ubiquitous baseline libraries (such as `libc` and `SQLite`), leaving a vast spectrum of custom libraries and application-level codebases entirely unexploited. The second is *emulation-dedicated ABIs*, led by proprietary solutions like Microsoft’s Arm64EC [14] and Apple’s Rosetta 2 [15]. While their vertically integrated stack of compilers, emulators and system interfaces achieves near-native performance for a specific guest/host pair (e.g., `x86_64` on `aarch64`), such rigid architectural coupling renders them platform-exclusive and unable to scale to diverse open-source ecosystems or emerging cross-ISA combinations.

Existing solutions partially exploit software portability through different design choices, but they remain limited in simultaneously supporting diverse application codebases and various guest/host ISA combinations. To address this limitation, we present *coMulator*, a source-guided, callback-capable, and ABI-bridging coordinated design that leverages compile-time assistance to automate function offloading for general libraries and applications, while preserving the ABI neutrality via an innovative cross-ISA calling mechanism named *gyhcall* (**g**uest **y**ields **h**ost **c**alls). As shown in Figure 1, at the compile time, *coMulator* automatically identifies offloadable guest functions and generates their equivalent host-native versions. During execution, *gyhcall* forwards guest-side calls to the host for native performance, thereby mitigating the overhead of DBT. Internally, *gyhcall* primarily addresses two key internal challenges: first, the discrepancy between guest and host ABIs, which is resolved through **calling conversion** by collaborative guest/host stubs; second, the control flow switching of guest callbacks, which is tackled via an elegant **emulation reentrancy** at runtime. However, the guest/host switching inherent in *gyhcall* incurs much higher overhead than normal function calls (even emulated). Thus, *coMulator* further includes four **critical optimizations** to eliminate unnecessary switching incurred by function offloading. Furthermore, as *gyhcall* is designed to be **target-neutral**, supporting new ISAs requires trivial efforts, and *coMulator* smoothly works on **all 9 guest/host combinations**³ among `x86_64`, `aarch64`, and `riscv64`. Ultimately, *coMulator*

³This includes identical-ISA emulation (e.g., `x86_64-on-x86_64`). These combinations serve as a useful evidence to *coMulator*’s target neutrality and can equally be accelerated by offloading.

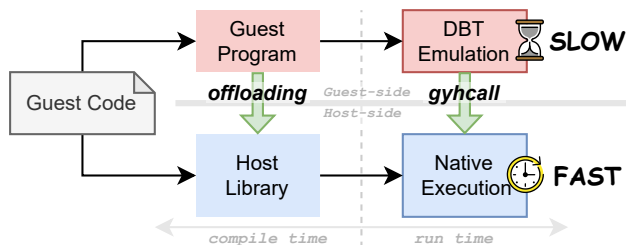


Fig. 1: *coMulator* selectively offloads guest code to the host side for native execution, thereby mitigating the DBT emulation overhead.

democratizes the offloading for vast custom codebases and guest/host ISA combinations. This even allows closed-source binaries to benefit from offloaded shared libraries.

We evaluate *coMulator* with SPEC CPU 2017 benchmarks. The results demonstrate that *coMulator* consistently achieves over a 7.90× average speedup across these architectures, reaching over 75% of native performance. Similar gains are also observed in real-world applications, demonstrating *coMulator*’s effectiveness and practicability in accelerating cross-ISA DBT.

In summary, this paper makes the following contributions:

- 1) We identify the key challenges in exploiting software partial portability for hybrid emulation, including callback handling, automated function offloading, and ABI interoperability across heterogeneous guest/host ISA combinations, and discuss the limitations of existing approaches in addressing these challenges.
- 2) We propose *gyhcall*, a callback-capable and ABI-bridging function offloading mechanism that enables bidirectional guest-host invocations while preserving ISA-native ABIs on both sides. We further introduce four complementary optimizations to reduce guest/host switching overhead and improve the efficiency of hybrid emulation.
- 3) We implement *coMulator*, a prototype integrating *gyhcall* and the proposed optimizations. Evaluations on SPEC CPU 2017 and real-world applications across multiple guest/host ISA combinations show that *gyhcall* significantly improves cross-ISA execution performance while maintaining broad applicability.

II. BACKGROUND AND MOTIVATION

A. Background

1) *Binary Translation*: Binary translation is a fundamental technique for achieving ISA compatibility, and is widely applied in application migration and in expanding the ecosystem of emerging ISAs. The core principle is to translate binary instructions from a guest ISA into equivalent instructions of a host ISA for execution. Because programs often exhibit dynamic behavior (e.g., loading external shared libraries at runtime), translation is typically performed on-the-fly. This leads to dynamic binary translation (DBT), which, despite its flexibility, suffers from substantial performance overhead. This inefficiency arises from two primary sources. First, semantic mismatches between guest and host ISAs often require multiple host instructions to emulate a single guest instruction, directly degrading execution speed. Second, the translation process itself is computationally expensive, and to ensure real-time translation, DBT typically forgoes deep optimizations during translation, further limiting execution performance. Consequently, cross-ISA emulation via DBT can be over 10× slower than the native execution [7], [12], [16], severely restricting its applicability in real-world scenarios.

2) *Cross-compilation*: Given DBT’s intrinsic limitations, cross-compilation can be considered as a superior alternative for migrating projects between ISAs. Nowadays, most applications are developed in high-level languages like C and

C++, whose source codes are largely ISA-agnostic. It is thus theoretically possible to recompile these projects directly into native binaries for the target ISA via the host compilation toolchain. In practice, however, many applications are not entirely architecture-independent and remain coupled to their original ISAs. This coupling primarily manifests in the following scenarios:

- **Target-specific code:** Architecture-dependent constructs like assembly and macros (Figure 2), often lead to compilation or linking failures in cross-ISA compilation. Additionally, build systems may be misconfigured for different ISAs, or essential toolchain utilities might be absent on the host platform. Collectively, these issues disrupt the migration workflow, frequently rendering the project unbuildable and necessitating tedious manual refactoring.
- **Dependency lock-in:** The aforementioned issues further propagate through the entire software supply chain. If upstream or downstream dependencies cannot be cross-compiled, the entire project becomes unportable. For instance, an application might rely on proprietary, closed-source libraries distributed only as ISA-specific binaries. In such cases, the entire dependency tree remains locked to the original architecture.

For these scenarios, existing cross-compilation practices face a rigid dichotomy: developers must either perform extensive manual code refactoring to completely strip out ISA dependencies, or entirely abandon cross-compilation, thereby falling back to full DBT and enduring its prohibitive emulation overhead.

B. The Opportunity

As shown in Figure 2, if the guest and host code can be run hybridly, exploiting the partial portability inherent in high-level source code, the overall emulation can be significantly accelerated by native execution. Existing works [9], [11]–[13], [17] have explored bypassing DBT by offloading certain guest function calls to semantically equivalent host functions at the binary level. However, they are largely constrained by substantial manual efforts to craft wrappers for offloaded functions; for instance, Risotto [12], a representative design, heavily relies on a custom Interface Definition Language (IDL) to explicitly declare function signatures within offloaded shared libraries. Furthermore, these methodologies have limited support for guest callbacks, where an offloaded function

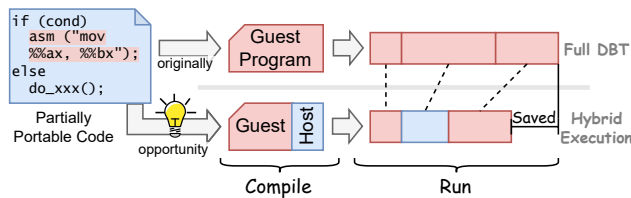


Fig. 2: By exploiting ISA-agnostic segments in source code and offloading them, the overall execution can be partially accelerated.

invokes a guest function via function pointers, thereby triggering an emulation reentrancy problem. Although such callbacks are relatively sparse at public library interfaces, they are commonly appear within internal functions. Consequently, the applicability of existing function offloading methods remains is typically limited to a narrow set of standard or common libraries, such as libc, SQLite rather than extending to broader, real-world code bases.

On the other hand, the aforementioned problems can be addressed at the source-code level; because source code retains crucial high-level metadata (such as function signatures) without stripping, cross-ISA equivalence can be confirmed automatically by the compiler. This has driven the second category of existing works: emulation-dedicated ABIs, with Microsoft’s Arm64EC [14] and Apple’s Rosetta 2 [15] serving as prominent examples. These approaches design a specialized, emulation-compatible ABI tailored to a specific guest/host ISA combination (e.g., `x86_64` on `aarch64`), and implement a vertically integrated stack comprising compilers, emulators, and system interfaces to back-port legacy codebases onto the new ISA. Benefiting from such vertical integration, they can fully exploit the partial portability and achieve near-native execution performance. Nevertheless, this tightly coupled paradigm also means that these methods are highly rigid and deeply specialized for their designated platforms and guest/host pairs; for instance, Arm64EC is strictly restricted to emulate `x86_64` binaries on Windows-on-Arm, and incapable of adapting to any other ISAs or OS ecosystems.

Under the emulation-dedicated ABI paradigm, supporting N guest ISAs on M host ISAs requires $N \times M$ unique ABIs, each demanding a dedicated stack of compilers, emulators, and system interfaces. This combinatorial explosion of engineering effort introduces substantial engineering complexity. Existing approaches remain limited in jointly supporting diverse application codebases and heterogeneous guest/host ISA combinations. To address this limitation, we propose coMulator, a source-guided, callback-capable, and ABI-bridging coordinated design that enables automated function offloading across general software workloads and heterogeneous ISA platforms.

C. Challenges

Achieving generalizable and automatic offloading is far from trivial, as it introduces three key challenges that must be concurrently addressed to guarantee both correctness and performance.

1) **Guest/Host ABI Discrepancies:** Without a dedicated emulation-compatible ABI, offloading must tackle discrepancies between guest and host calling conventions [18], [19] regarding parameter-passing, stack layouts, etc. Existing binary-level methods rely on manual wrappers, which are error-prone and scale poorly to real-world codebases. To automate offloading for vast custom libraries and applications, our approach must robustly navigate real-world codebases, programmatically demarcate offloadability boundaries, and handle ABI conversions automatically. Furthermore, unlike regularized library interfaces, internal functions frequently introduce

architectural side effects (e.g., mutating global variables). These side effects must be accurately propagated across the cross-ISA boundary to guarantee strict semantic equivalence, rendering automated calling conversion challenging.

2) **Guest callbacks and Non-local Control Flows:** As previously discussed, offloaded functions may invoke guest functions from the host side either directly or indirectly, necessitating a bidirectional calling channel between host-native code and guest-emulated execution. Current binary-level function-offloading methods typically restrict their scope to callback-free scenarios [12], [13], such as leaf functions. Without a dedicated ABI, managing complex call stacks that involve nested callbacks remains an open challenge. Robustly handling these scenarios requires seamless synchronization between emulated and native execution phases while preserving runtime state consistency. This complexity is further exacerbated by non-local control-flow transfers (e.g., `longjmp`) prevalent in general-purpose codebases. Because these constructs abruptly disrupt standard call-return semantics, they render cross-ISA execution state management exceptionally error-prone and difficult to generalize.

3) **Overhead Inherent within Guest/Host Switching:** Generally, a boundary-crossing function call incurs significantly higher overhead than a normal intra-ISA invocation, a penalty that accumulates for frequently invoked functions. To maximize the performance benefits of host-native execution, minimizing both the latency and frequency of guest-host transitions is therefore becomes important, but an important design consideration. This trade-off underscores the critical need for deep, systematic optimizations; furthermore, it mandates a appropriate offloading strategy to select the most profitable function candidates.

III. DESIGN AND IMPLEMENTATION

coMulator is a coordinated tool-set consisting of a compiler tool-chain (based on LLVM [20]) and corresponding emulator (based on QEMU [21]). At compile time, it transforms the LLVM IR of eligible guest functions (*GFuncs*) and generates semantically equivalent host functions (*HFuncs*). At run time, the transformed *GFuncs* trigger *gyhcalls* supported by the extended QEMU to invoke the corresponding *HFuncs*.

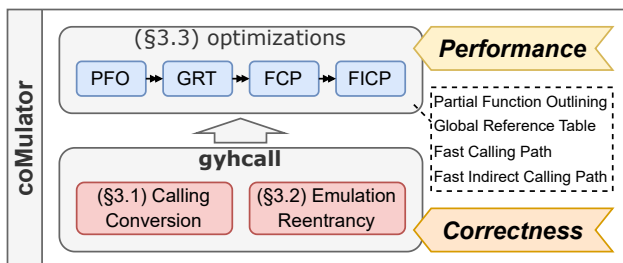


Fig. 3: Core techniques of coMulator. The *gyhcall* establishes the foundation for correctness in offloaded execution, complemented by the module of four optimizations to harvest performance benefits.

Figure 3 illustrates coMulator’s overall design, which primarily consists of two parts. The first is *gyhcall*, the foundational module that addresses how to offload eligible guest functions and invoke them at runtime. As the cornerstone for ensuring correctness and completeness, *gyhcall* is detailed in §III-A and §III-B. The second component, discussed in §III-C, comprises four progressively applied optimizations. Each optimization builds on the previous one to mitigate guest/host switching overhead inherent in *gyhcall*, thereby delivering cumulative performance gains. Finally, §III-D describes the current implementation of coMulator and discusses key technical considerations.

A. *gyhcall* Calling Conversion

Figure 4 illustrates the designed control flow transfer of *gyhcall*. For each *GFunc* selected for offloading, its original function body is completely *chopped off* at compile-time, leaving behind stub code that triggers the guest-to-host call and performs the necessary calling conversion. Since an offloaded *HFunc* may also call back guest functions, reciprocal conversions are likewise needed. Thanks to the encapsulation of functions, the modifications within *GFunc* remain transparent to the rest of the guest-side code, thereby ensuring strong compatibility and adaptability.

In principle, whether a guest function is eligible for host-side offloading is inherently context-free, depending solely on its own execution semantics; specifically, it must not exhibit non-local control-flow behaviors. Consequently, shared libraries can be optimized and offloaded independently of the applications that depend on them. By applying *gyhcall* to these libraries, the resulting drop-in replacements for the original `.so` files enable upper-level applications to reap immediate performance gains without recompilation, as the strict binary compatibility of shared libraries guarantees semantic correctness. This characteristic allows legacy, closed-source proprietary applications to seamlessly benefit from our automated offloading, matching the compatibility advantages of existing manual wrapper solutions. Such scenarios are highly prevalent where commercial software utilities utilize open-source components, thereby demonstrating profound practical utility.

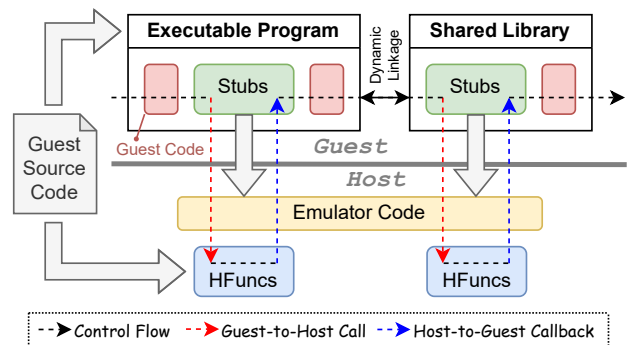


Fig. 4: The control flow transfer of *gyhcall*.

The fundamental principle of bridging ABI disparities remains consistent for both guest-to-host and host-to-guest conversions: leveraging stack memory as a common intermediary for data exchange (Figure 5). While this approach is well-established in IPC and FFI mechanisms [22], [23], *gyhcall* extends it to handle global symbols referenced by offloaded *HFuncs*. These guest-space symbols are collected by the *GFunc* stubs and passed to the host via two specialized arguments, *extRefPad* and *cbStubPad*, which resolve and forward all reference addresses to the corresponding *HFuncs*. This stack-based mediation effectively isolates and decouples the guest and host environments, serving as the linchpin for achieving target neutrality.

Unlike *HFuncs*, standard guest functions in host-to-guest callbacks are guest-ABI-compliant and cannot directly access stack pads. To solve the issue, *gyhcall* generates guest-side callback stubs. As shown in Figure 6, each stub takes three parameters: the target guest function address (*func*), plus *retPad* and *argPad* for data exchange. The stub body extracts arguments from *argPad*, performs an indirect call to *func*, and finally stores the result into *retPad*. This mechanism enables *coMulator* to invoke any arbitrary guest function correctly through a unified interface.

In summary, offloading a guest function involves five compile-time steps (Figure 6): 1) The *GFunc* body is replaced with a stub that marshals *argPad*/*retPad*/*extRefPad*/*cbStubPad*. 2) Callback stubs are generated on the guest side. 3) The original function body is extracted and refactored into an *HFunc*, adapted to the pad-based interface. 4) Two primitive functions, `__gyh_call__` and `__gyh_callback__`, are integrated to handle the cross-ISA transition. 5) Unique, consecutive identifiers are assigned to each *HFunc* and hard-coded into the *GFunc* stubs to ensure $O(1)$ lookup performance.

B. *gyhcall* Emulation Reentrancy

During execution, offloaded *HFunc* may call back to guest-side functions, which gives rise to what we refer to as the “emulation reentrancy” issue. This behavior interacts directly with the runtime structure of the emulator (QEMU), as illustrated in Figure 7. Execution begins in QEMU’s own code, which loads the guest binary and emulates it from the entry point.

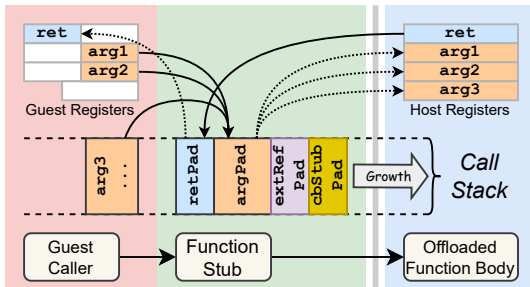


Fig. 5: Data exchange in guest-to-host calls.

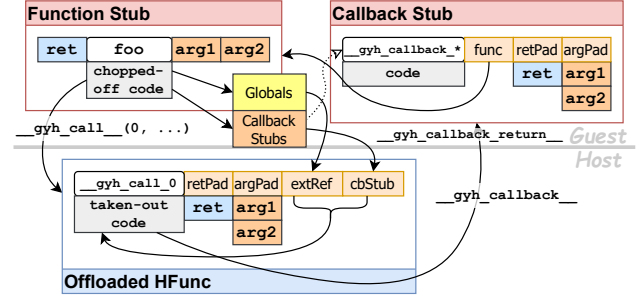


Fig. 6: Transformations needed for offloading a *GFunc*.

The runtime therefore maintains two independent call stacks: the QEMU host stack and the guest stack that follows the guest ABI. Switching between them occurs through QEMU’s `do_cpu` entry and exit points, which we extend to enable controlled guest-host transitions.

Given the aforementioned structure, invoking an *HFunc* cannot be treated as a conventional functional call, because *HFunc* executes host-side code but must operate logically on the guest stack. As a result, directly invoking *HFunc* from the host during events such as system calls would incorrectly allocate its stack frames on the host side. Non-local guest control flow operations (e.g., `set jmp/long jmp`) would then fail to unwind these frames, resulting in memory leaks and potential crashes.

To ensure architectural transparency, *gyhcall* constructs the *HFunc* execution context within the guest-side stack (Figure 8), treating the host-side frame as local data of a specialized `__gyh_call__` primitive. This design maintains stack consistency and aligns with guest control-flow semantics.

Guest-to-Host (g2h) Calls. When a *GFunc* stub is invoked, it marshals arguments into designated stack pads and calls `__gyh_call__`, which traps into our modified QEMU via a specialized syscall. The emulator aligns the guest stack with host ABI requirements, reconstructs the *HFunc* context, and switches to native execution. Upon completion, the *HFunc* stores its return value in *retPad* and reverts to the emulation context. The *GFunc* stub then resumes and returns to its caller,

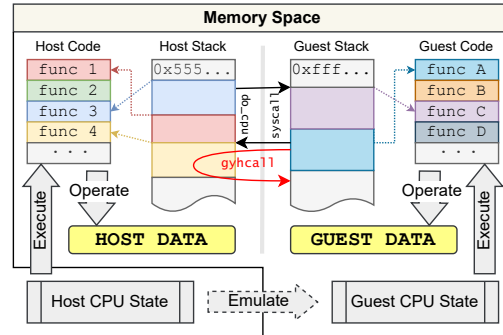


Fig. 7: Runtime structure of QEMU. *gyhcall* is triggered by a syscall but executes within the guest stack.

making the host-side execution invisible and non-intrusive to the guest.

Host-to-Guest (h2g) Callbacks. Conversely, when an *HFunc* triggers a callback, it prepares stack pads and invokes the host-side `__gyh_callback__` primitive. This primitive saves the host context onto the guest stack and reconfigures the emulator’s registers to point to the guest callback stub. To ensure proper return symmetry, the invocation is routed through a trampoline, `__gyh_callback_return__`, preventing the control flow from erroneously returning to the *HFunc*’s original guest caller. After the callback completes, QEMU restores the saved host context and seamlessly resumes the *HFunc*.

The aforementioned design for g2h and h2g switching completely encapsulates host-side execution within `__gyh_call__` and `__gyh_callback_return`, ensuring strong compatibility. By isolating the *HFunc* context from the guest control flow, `gyhcall` maintains system integrity even during non-local control transfers (e.g., `setjmp/longjmp`) or guest-side crashes. Since *HFunc* remains side-effect-free regarding QEMU’s internal state, such anomalies do not compromise the emulator’s operational logic.

Furthermore, correct *HFunc* execution hinges on consistent memory layouts between the guest and host. While ISA/ABI disparities typically manifest as differing data widths or alignment requirements at the source level, these attributes are explicitly finalized at the LLVM IR level. Our empirical results confirm that once the extracted guest IR is compiled for the host ISA, the resulting binary operates reliably on guest data. This underscores LLVM IR as the optimal abstraction layer for implementing `gyhcall` to handle guest/host switching, as it effectively bridges high-level semantic intent with low-level architectural execution.

C. Performance-Critical Optimizations

`gyhcall` establishes the foundation for bidirectional calls between guest and host environments. However, it can be foreseen that such cross-environment invocations are considerably more expensive compared to ordinary function calls — even when the latter are emulated. Only when the execution time spent in a function is sufficiently long can the benefits of ex-

ecuting fast host-native code offset the overhead of guest/host switching. Unfortunately, many programs in practice don’t have such long-run hot functions, and are instead usually filled with short functions involving numerous mutual invocations. To reduce unnecessary overheads, we incorporate four key optimizations in coMulator.

1) *Partial Function Outlining (PFO)*: PFO is employed to broaden the applicability of `gyhcall`. There are several constraints that may preclude a guest function from being offloadable:

- 1) **Variadic calls.** Functions containing variadic calls, where argument types are unknown at compile time (e.g., `printf`), cannot be safely converted for host execution.
- 2) **Variadic functions or unsupported types.** Functions that are themselves variadic, or that operate on unsupported data types on the host ISA (e.g., X86_AMX on AArch64) must remain on the guest side for execution.
- 3) **Inline assembly and non-local control flows.** Functions incorporating inline assembly or non-local control flows (such as `setjmp/longjmp`), require guest-side emulation to preserve correctness.

We address these challenges through an outlining process that isolates those code segments, thereby allowing the remaining portions of the guest function body to be offloaded. For (1), simply outlining those calls by generating new non-variadic functions that perform the variadic calls indirectly will suffice. For (2), however, such a simple trick doesn’t work. For (3), problematic code like assembly tends to have context-sensitive semantics and must be executed within their original function bodies too.

To solve (2) and (3), we propose a new tactic called “complementary outlining”. The tactic first identifies other “unpolluted” basic blocks, then extracts all of them into standalone functions and re-links them with tail call instructions. This breaks a big unoffloadable function into a collection of small offloadable functions, which can benefit from later optimizations.

2) *Global Reference Table (GRT)*: In the baseline design (Figure 6), the chopped-off stub code reconstructs `extRefPad` and `cbStubPad` for each invocation, which incurs repeated memory writes. Given that these guest ad-

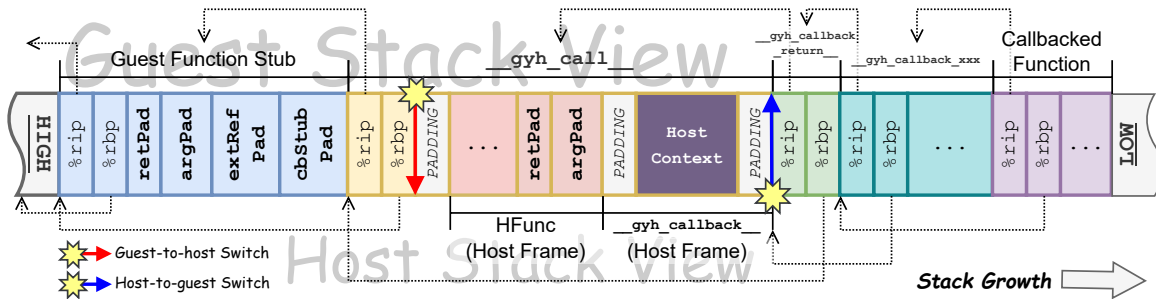


Fig. 8: The guest call stack layout with a guest-to-host call followed by a host-to-guest callback. The x86_64 guest ISA used here is just for demonstration. The layout is generally applicable to any other guest ISAs.

addresses typically remain unchanged after program loading, regathering them is generally redundant and unnecessary.

To eliminate this cost, a unified global reference table (GRT) is pre-constructed for all functions during loading the guest program. The GRT aggregates all global references and callback stubs across *HFuncs*, indexed by their assigned UIDs. Afterwards, the table's address can be passed directly to the host side. This optimization requires modifications to both the guest and host sides: 1) at the guest side, two additional global pointer array constants — `__gyh_extreftbl__` and `__gyh_cbstubtbl__` — are emitted, populated with the addresses of all globals and callback stubs from all *HFuncs* in the order of assigned UIDs; 2) at the host side, each *HFunc*'s code is modified to retrieve values from the passed global reference tables with correct UIDs.

Although saving the overhead of unnecessary argument preparation may seem trivial, GRT provides a crucial consolidated mechanism for the host to access all guest references. This foundationally enables the later FCP and FICP optimizations.

3) *Fast Calling Path (FCP)*: When two offloaded *HFuncs* call each other, using guest-to-host callbacks for such invocations would be circuitous and costly. Instead, the invocation can be performed as ordinary host-to-host function calls (i.e., Fast Calling Path (FCP)), thereby significantly eliminating overhead of guest-host switching. It must be emphasized that FCP is only available when the GRT optimization is also enabled. This is because *HFuncs* have distinct references and callback stubs, necessitating a global table containing all *HFuncs*' information to support indirectly called functions.

The FCP optimization requires modifications solely on the host side: when generating an *HFunc*, we preserve its original function signature and generate a standalone "forwarder" function to perform pad unpacking, just as the callback stubs do, so that the *HFunc* can be called directly. Besides, the forwarder function sets the passed global reference tables to host-side global variables before forwarding to share them among all offloaded *HFuncs*. Now the offloaded *HFuncs* can safely call each other as ordinary host functions, bypassing guest-host switches and preserving efficiency. Together with GRT, FCP transforms cross-invocations into native calls, as summarized in Figure 9.

4) *Fast Indirect Calling Path (FICP)*: The FCP optimization is limited to direct calls between offloaded *GFuncs*. For indirect calls, the compile-time ambiguity of the callee typically necessitates a fallback to the emulation via `__gyh_callback__`. However, this incurs redundant guest/host switching if the callee is actually an offloaded function. To resolve this, we propose FICP (Fast Indirect Call Path).

At compile time, FICP reserves a 16-byte prefix (two 64-bit slots) before each *GFunc* and registers their addresses in a global table, `__gyh_functbl__`. At runtime, our modified QEMU traverses this table, marks the code segments as writable, and populates each prefix with the corresponding *HFunc* address and a checksum (the address XORed with a

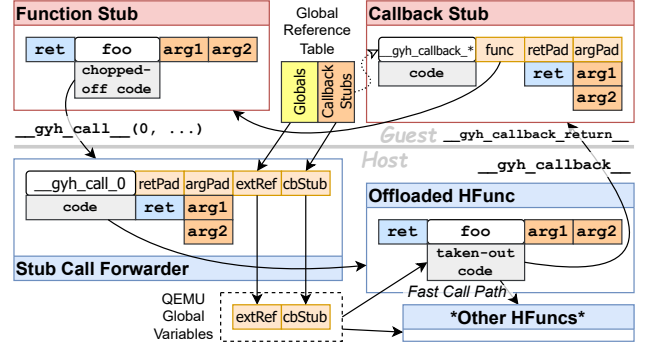


Fig. 9: Transformations needed for offloading a guest function, enabling both GRT and FCP.

predefined magic number). All indirect calls in *HFuncs* are then instrumented with a conditional guard. This guard inspects the callee's prefix: if the checksum is valid, it performs a direct host-native call using the cached address; otherwise, it falls back to `__gyh_callback__`.

Beyond resolving indirect call overhead, FICP establishes inter-module fast paths for shared libraries. Previously, *HFuncs* invisible at compile time could not be linked natively, forcing a fallback to `__gyh_callback__`. FICP enables these cross-module calls to be dynamically identified, bypassing unnecessary guest/host switching. This is particularly effective for functions in **C standard library (libc.so)**, which frequently reside at the leaf positions of the call graph (Figure 10).

D. Implementation

In this section, we discuss key aspects of coMulator's implementation.

1) *Target Neutrality*: `gyhcall`'s stack-based calling conversion grants coMulator great target neutrality. Moreover, most compile-time transformations occur at the LLVM IR level, thus remaining ISA-agnostic. The runtime also employs

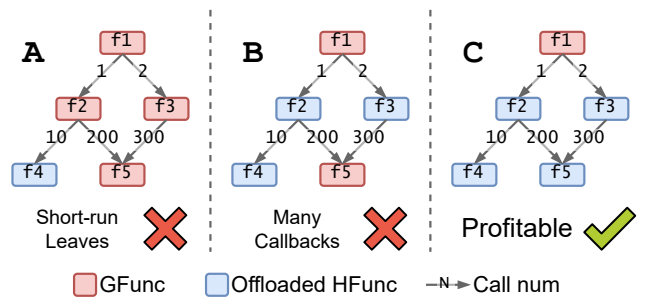


Fig. 10: Offloading is not always profitable: (A) Small leaf functions are too short-lived to compensate for the guest/host switching overhead; (B) Offloading intermediate nodes triggers frequent callbacks, offsetting native execution gains; (C) Benefits are realized only when offloading granularity is sufficiently coarse.

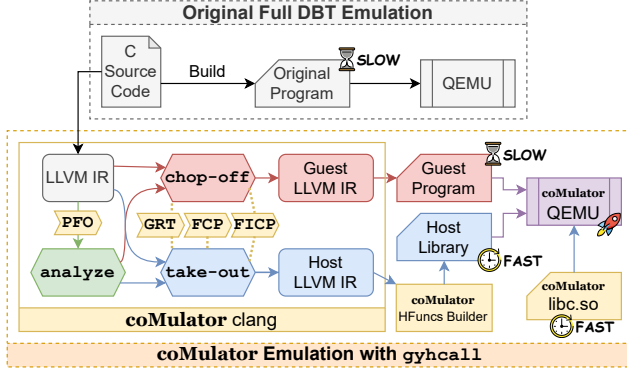


Fig. 11: The coMulator toolset and its workflow.

unified syscalls for guest/host switching. Together, these design points make supporting a new ISA require trivial effort. As an example, Listing 1 shows the two key magic functions for `gyhcall` on `x86_64`. Our current implementation already supports all 9 guest/host combinations for `x86_64`, `aarch64`, and `riscv64` with only 67–96 lines of assembly and 175–323 lines of C code per ISA.

Listing 1: Magic functions for `x86_64`

```

1 __gyh_call__:
2   pushq %rbp
3   movq  %rsp, %rbp
4   mov   __gyh_callback_return__@GOTPCREL(
5     %rip), %r9
6   movq  $50406, %rax
7   syscall
8   popq  %rbp
9   ret
10 .globl __gyh_call__
11 .type  __gyh_call__, %function
12
13 __gyh_callback_return__:
14   pushq %rbp
15   movq  %rsp, %rbp
16   call  *%rcx
17   popq  %rbp
18   movq  $50906, %rax
19   syscall
20 .globl __gyh_callback_return__
21 .type  __gyh_callback_return__, %function

```

2) *Offloading Strategy*: coMulator provides the flexibility to precisely control the offloading scope to maximize performance benefits. Currently, we employ three heuristics to guide the selection of function offloading:

- 1) *HFuncs* are always generated for FCP/FICP to avoid `__gyh_callback__` when invoked by other *HFuncs*.
- 2) Small functions with low computational density are retained as guest code unless they accumulate significant weight along the call chain.
- 3) Bottom-level functions in the call graph are prioritized for offloading to minimize boundary callbacks.

While determining the optimal offloading set for a given program — perhaps through profiling — represents an intriguing research direction, it is currently out of the focus of this paper and is thus left as future work. To establish a worst-

case lower bound, we set the threshold of heuristic rule (2) to 0 in our evaluation (§ IV), which simply offloads all eligible functions. Although the result may not be the optimal, it has already been good enough to deliver substantial performance gains and convincingly validating the core potential and practical utility of coMulator.

3) *Toolset and Workflow*: coMulator is distributed as a toolset consisting of four core components, with workflow being depicted in Figure 11:

- **clang Wrapper**: Loads an LLVM plugin to automate transformations and generate `host.bc` IR.
- **HFunc Builder**: Compiles the `host.bc` IR into host-native shared libraries.
- **Modified QEMU**: A customized version that dynamically loads *HFunc* libraries and manages execution.
- **Offloaded C Standard Library (libc.so)**: Provides fast-path *HFunc* implementations for side-effect-free modules (e.g., `<malloc.h>`, `<string.h>`) via `LD_PRELOAD`, complementing FICP optimizations.

Beyond these core components, the toolset also contains various utilities for debugging and profiling. While these are indispensable for development and bug localization, we omit their detailed discussion for the sake of brevity.

E. Discussion

While previously touched upon implicitly, for the sake of clarity and comprehensiveness, we explicitly summarize the current applicability scope of coMulator in Table I, categorized by basic C language features, multi-threading capabilities, and advanced C++ language features.

TABLE I: The feature support summary of coMulator. The table is categorized into three parts: C language features, multi-threading capabilities, and advanced C++ language features. “OK” means the feature is allowed but excluded from offloading.

Feature	Supp.	Feature	Supp.
Variadic arguments	OK	Thread local storage	YES
Inline assembly	OK	Multi-threading	YES
setjmp/longjmp	OK	Atomics	OK
Syscalls	OK	Stack trace	NO
Signals	YES	Exceptions	NO
C Language	YES	C++ (-fno exp.)	YES

Most of these features are excluded from offloading, while coMulator focuses on accelerating a subset of compatible functions. These features are marked as “OK” in Table I. However, the exception feature in languages like C++ is one such exception here: Although PFO could theoretically isolate exception-related instructions, `clang++` typically replaces all standard `call` instructions with `invoke` when exceptions are enabled. This “pollutes” the entire IR module, making most code unaffordable and excluded by coMulator. As a result, coMulator currently has limited applicability to codebases that rely heavily on C++ exceptions. However, C++ projects compiled with the `-fno-exceptions` flag are unaffected

by these issues, and can be supported by coMulator in the same manner as C projects.

Stack trace support is currently not feasible under coMulator’s execution model, because `gyhcall` encapsulates the execution context of an offloaded *HFunc* within a `__gyh_call__` stack frame, making it invisible to the guest environment, which leads to information loss during guest-side stack walking. Actually this is one of the underlying reasons why exceptions are not supported. Language-specific runtime support is required for non-local control flow mechanisms like `setjmp/longjmp`. This is one of the key reasons why coMulator currently focuses primarily on C code bases only. C++ is added additionally thanks to its compatibility with C when `-fno-exceptions` is enabled.

Since the offloaded *HFuncs* are semantically equivalent to the *GFuncs* from the guest view, they work well in multi-threading environments. To be specific, the thread management calls (e.g. `fork()`, `join()`, etc) in *HFuncs* simply yield callbacks to the guest side and are handled just original emulation. Due to their side-effect-free property, *HFuncs* are thread-independent and can be invoked from any thread context or signal handlers without compromising semantic correctness. Also, since QEMU just uses host threads to emulate guest threads, forming a one-to-one mapping, thread local storage can be supported naturally by simply making those globals in `gyhcall` thread-local too. Atomic instructions or intrinsics can be outlined just like other tricky features.

Finally, we clarify the interplay between coMulator and LLVM. While our prototype leverages LLVM IR, the fundamental techniques, `gyhcall` and its optimizations inside coMulator are not tightly coupled to LLVM. The core offloading logic can potentially be adapted to other ISA-agnostic IRs or source-level transformations; indeed, the design principles detailed in Section III are formulated without strict reliance on LLVM-specific primitives. However, LLVM IR strikes a superior balance for coMulator: it offers a robust ISA-neutral abstraction while providing a highly regularized format that reduces the irregularities of source-level constructs.

IV. EVALUATION

A. Experimental Methodology

1) *Testbed*: We evaluate coMulator across all nine supported guest/host ISA combinations of `x86_64`, `aarch64`, and `riscv64`. The detailed hardware and system configurations are summarized in Table II.

TABLE II: Testbed configurations.

	x86_64 (PC)	aarch64 (Raspberry Pi 4B)	riscv64 (StarFive VisionFive 2)
CPU	AMD Ryzen 9 5950X @ 3.4GHz	Broadcom BCM2711 @ 1.8GHz	StarFive JH-7110 @ 1.5GHz
RAM	32GB	4GB	4GB
OS	Ubuntu 24.04		
LLVM	18.1.8 (APT default)		
QEMU	9.1.50 (modified)		

2) *Workloads*: Our evaluation is conducted mainly using the *SPEC CPU 2017*’s *rate* suite. Owing to the current implementation’s focus on C language support (`libc.so` in §III-D3), our study incorporates all available C benchmarks along with two C++ benchmarks that are compatible with the `-fno-exceptions` compilation flag. The complete set of eligible benchmarks is detailed in Table III.

3) *Metrics and Schemes*: Our evaluation focuses on performance, specifically measured as the inverse of execution time. For convenience, all metrics are normalized to native execution. The following schemes are compared:

- *native*: programs built and executed natively, representing the upper bound of achievable performance.
- *QEMU*: guest programs emulated with the vanilla QEMU on the host machine, serving as the baseline.
- *coMulator variants*: coMulator using `gyhcall` with optimizations progressively enabled from *none* to *full*, i.e., *coM*, *coM-p* (+PFO), *coM-pg* (+PFO+GRT), *coM-pgf* (+PFO+GRT+FCP), and finally *coM-pgfi* (+PFO+GRT+FCP+FICP)⁴.

⁴With identical calling overhead between FCP and FICP, the performance variance in *coM-pgf* and *coM-pgfi* solely measures `libc.so` offloading efficiency, as SPEC benchmarks depend on no other external libraries.

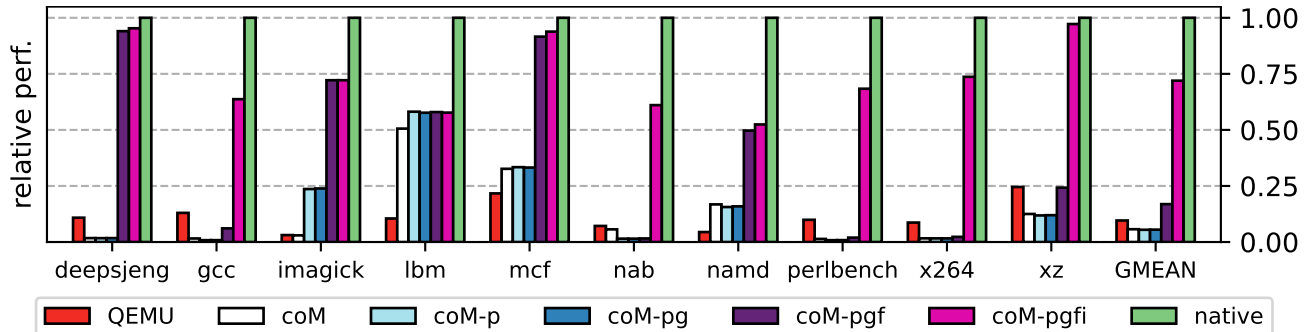


Fig. 12: Performance of the `x86_64` emulation on `aarch64`. The bars show relative performance to native execution.

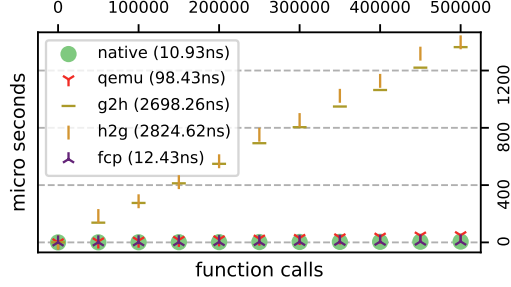


Fig. 13: Correlation between runtime and number of function calls. The numbers in the legend represent the slope of the regression fit, which approximates the time overhead of a single call.

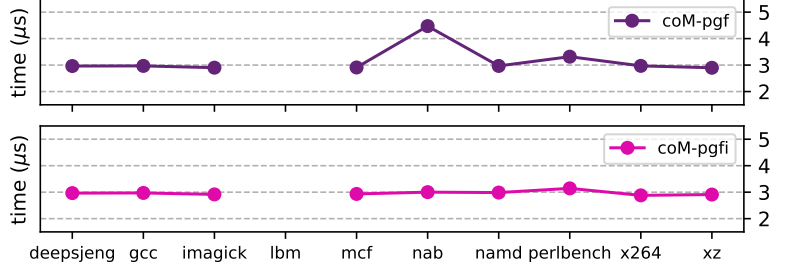


Fig. 14: Ratio of saved time to reduced *gyhcalls* for all benchmarks in *coM-pgf* and *coM-pgfi*. The *lbm* has a NaN value as its *gyhcall* number remains unchanged. The consistent distribution around 3 μ s corroborates the performance model in Figure 13.

TABLE III: Evaluated SPEC benchmarks.

Wkld	Lang	KLOC	Wkld	Lang	KLOC
perlbench	C	362	x264	C	96
gcc	C	1304	deepsjeng	C++	10
mcf	C	3	imagick	C	259
namd	C++	8	nab	C	24
lbm	C	1	xz	C	33

B. Performance Improvement

Given the consistent trends across all nine experimental configurations (summarized in Figure 15c), we focus our detailed analysis on the most representative scenario: *aarch64* simulating *x86_64* binaries. Figure 12 illustrates the performance of all schemes normalized to *native*, clearly depicting the incremental impact of the optimizations discussed in §III-C.

Overall, with all optimizations enabled, *coMulator* achieves an average of 72% of native performance (*native*), representing a $7.49\times$ speedup over the vanilla QEMU (*QEMU*). In the best-case scenario, *coMulator* reaches 0.97 of native performance (*xz* benchmark) and a $11.81\times$ speedup (*namd* benchmark), demonstrating the efficacy of *coMulator*. Conversely, the unoptimized *coM* implementation performs even worse than the baseline emulation (*QEMU*) (0.057 vs. 0.096), and performance remains suboptimal when optimizations are incomplete (*coM-p* 0.0547, *coM-pg* 0.0552, *coM-pgf* 0.1690). This underscores the necessity of our integrated optimization suite. As shown in Figure 12, nine out of ten benchmarks achieve substantial performance improvements through FCP (*coM-pgf*) and FICP (*coM-pgfi*). As anticipated, without fast-path support, naive host-side execution is insufficient to offset the prohibitive overhead of frequent guest/host switching.

However, the other optimizations remain indispensable for two reasons. First, from a design perspective, these optimizations are **incremental rather than orthogonal**; the preceding layers are prerequisites that make FICP possible. Second, individual benchmarks highlight the specific efficacy of certain techniques. For instance, in *imagick*, *coM-p* shows a significant improvement over *coM* (0.23 vs. 0.030), indicat-

ing that PFO effectively offloads more hotspots. Similarly, FCP alone yields major gains in *deepsjeng*, *imagick*, *mcf*, and *namd*, reaching performance levels comparable to *coM-pgfi*. This suggests that the hotspots in these applications are focused on internal computation rather than frequent C library invocations—a hypothesis confirmed by our profiling.

Notably, the GRT optimization appears to have a marginal impact; *coM-pg* shows almost no deviation from *coM-p* across all benchmarks. This indicates that the overhead of constructing *extRefPad* and *cbStubPad* is indeed negligible relative to total execution time. Nevertheless, given its foundational role in enabling subsequent optimizations, we consider GRT a vital structural component.

In summary, these results align with our expectations and validate our design. These conclusions are equally applicable to the other eight guest/host ISA combinations.

C. Performance Attribution

In this section, we conduct a series of targeted experiments to identify the root causes of the observed performance improvements and verify that the gains stem directly from our proposed design.

1) *Effect of PFO*: The PFO optimization only yields significant improvements in the *imagick* and *lbm* benchmarks (*coM-p* 0.23, 0.58 vs. *coM* 0.030, 0.51). By performing a deep-dive analysis of the generated code for both versions, we found that in *imagick*, PFO enables a larger number of functions to be offloaded (*coM-p* 2061 vs. *coM* 1271). This allows a greater portion of the workload to execute natively on the host rather than being emulated, thereby enhancing overall performance.

In *lbm*, while PFO yields only a marginal increase in the number of offloaded functions (*coM-p* 18 vs. *coM* 10), it successfully captures pivotal hotspots such as *LBM_performStreamCollideTRT*. Offloading these few dominant functions is sufficient to drive substantial performance gains. In summary, PFO indirectly boosts performance by expanding the scope of offloadable functions, serving as a foundational optimization for *coMulator*.

2) *Overhead of Guest/Host Switching:* In fact, we have anticipated that guest/host transitions in `gyhcall` would incur significantly higher overhead compared to standard function calls during our design and implementation. To quantify this overhead, we developed a minimalist micro-benchmark centered on a controlled loop of null-function calls, precisely triggering a specified number of guest-to-host `g2h` calls or host-to-guest `h2g` callbacks.

We correlated the observed total execution time with the number of invocations (ranging from 0 to 500,000). As shown in Figure 13, the data points exhibit a near-perfect linear correlation ($0.997 \leq R^2 \leq 0.998$). By extracting the slopes of the regression lines, we found that the overhead of a single `g2h` (2698 ns) or `h2g` (2824 ns) transition can be over two orders of magnitude higher than an ordinary function call in either native execution (10.93 ns) or QEMU emulation (98.43 ns). On the other hand, the time overhead for a fast call path (`fcp` 12.43 ns) is almost the same as native function calls. This evidence confirms that guest/host switching is the primary performance bottleneck in `gyhcall`.

3) *Correlation between Switching Reduction and Performance:* To elucidate the pervasive and substantial performance gains achieved by `coM-pgf` and `coM-pgfi`, we conduct a comprehensive quantitative analysis of `g2h` calls and `h2g` callbacks across all evaluated workloads. Specifically, we calculate the ratio of saved execution time to the reduction in `gyhcalls` relative to `coM-pg`, verifying whether these results align with our previously measured guest/host switching overhead. The final results are illustrated in Figure 14.

Notably, the ratio for `lbm` is omitted as the number of `gyhcalls` remains unchanged (resulting in a division by zero); correspondingly, its performance variation in Figure 12 is negligible and within the margin of error. A similar divergence is observed for `nab` under the `coM-pgf` configuration, which we also attribute to experimental noise, as its performance fluctuated only insignificantly in Figure 12.

Remarkably, for all other benchmarks, the resulting values align closely with the slope derived from our linear regression in the micro-benchmark ($\approx 3\mu s$). This high degree of consistency empirically corroborates our earlier intuition that each avoided transition saves several microseconds of execution time. Especially in `coM-pgfi`, the inter-module fast paths for `libc.so` effectively mitigates the callback explosion described as case (B) in Figure 10.

We can thus confidently conclude that `coMulator`, through the integration of our optimization suite, successfully transforms fragmented, "cost-prohibitive" offloading into efficient, localized execution.

D. Comparison to Existing Works

To position `coMulator`, we evaluate it against representative prior works from both binary-level function offloading and emulation-oriented ABI designs: `Risotto` [12] and Microsoft's `Arm64EC` [14].

1) *Binary-Level Function Offloading:* `Risotto` [12] is a recent QEMU-based binary-level offloading framework. It

operates solely at the binary level and relies on manual IDL scripting to declare function signatures. In our evaluation, we extend its official standard math library (`libm.so`) IDL scripts to cover other common standard headers (such as `<string.h>`), aligning its offloading scope with `coMulator's` `libc.so`.

As shown in Figure 15a, offloading only standard libraries yields limited GMEAN speedups over vanilla QEMU (0.10 vs. 0.09 of native performance), because most computation in SPEC workloads is concentrated in application-level code rather than standard library routines. Noticeable improvement is observed mainly in `nab` ($1.25\times$ gain), which is more math-intensive and thus benefits more from library offloading. Conversely, when manually expanding the offloading scope to application-level code (Figure 15b), `Risotto` achieves 0.68 and 0.62 of native performance on `mcf` and `lbm`, approaching `coM-pgfi's` 0.94 and 0.58. `Risotto` exhibits a marginal performance advantage over `coMulator` in `lbm` owing to its lightweight guest/host switching. Unburdened by the architectural necessity to support guest callbacks or handle emulation reentrancy, `Risotto` naturally introduces lower runtime overhead.

However, enabling application-level offloading in `Risotto` incurs a substantial manual engineering overhead. It required extensive codebase analysis to decouple monolithic applications into dynamic libraries, forcing us to **manually rewrite 975 and 150 Lines of Code (LoC)** for `mcf` and `lbm`—a significant fraction of their total sizes (3,484 and 1,042 LoC, respectively). Furthermore, `Risotto's` lack of guest callback support blocks some functions in `mcf` from being offloaded, capping its performance potential. This intractable manual scripting explains why we only report `mcf` and `lbm`; for other massive benchmarks like `gcc` (1,304 KLoC), manual refactoring is practically impossible. On the other hand, `coMulator` requires no modification to the code, including build scripts like `Makefile`, which precisely highlights its practical value.

2) *Emulation-Dedicated ABI:* `Arm64EC` [14] is a proprietary Windows-on-ARM (WoA) ABI specification within the WoA's Prism Emulator, custom-tailored for a singular guest/host pair (`x86_64` on `aarch64`). Like `coMulator`, it can compile legacy code into `x86_64`-compliant `aarch64` "thunks" via a specialized compiler toolchain (MSVC) to enable hybrid guest/host execution.

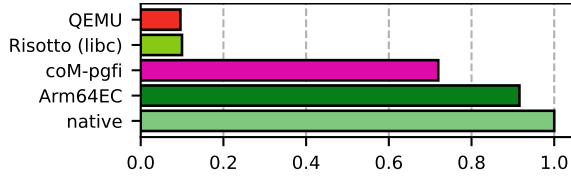
Since `Arm64EC` targets the Windows platform while `coMulator` operates on Linux/QEMU, their performance metrics (Figure 15a) serve as a qualitative baseline comparison rather than a strict apple-to-apple benchmarking. However, since all results are normalized against their respective native execution baselines, they offer meaningful comparative insights. `Arm64EC` exhibits exceptional efficiency, achieving 0.91 of native performance, outperforming `coMulator` under its optimized `coM-pgfi` configuration (0.72). This gap stems from `Arm64EC's` highly specialized, vertically-integrated nature, benefiting from kernel-level integration, a closed-source compiler, and a highly tuned interpreter dedicated to one target pair. As a result, it provides a strong reference point for highly

optimized, platform-specific systems.

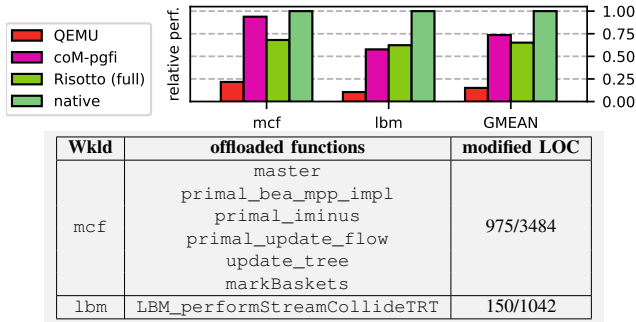
Conversely, coMulator’s core strength lies in its **superior generality** driven by `gyhcall`. coMulator’s target-neutral design enables cross-ISA execution across multiple configurations, naturally scaling to a 3×3 matrix spanning **9** different ISA combinations (Figure 15c) with minimal target-specific engineering. By accepting a modest performance trade-off against a platform-specific industrial monolith, coMulator delivers broad architectural versatility, providing a transparent, extensible, and automated alternative to closed proprietary systems.

E. Real-World Case Studies

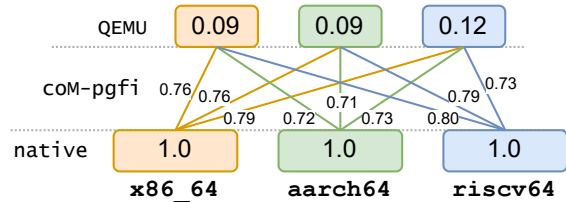
We further evaluate coMulator on relatively realistic and proprietary applications. For better reproducibility, three open-source applications are included to mimic the case where



(a) coMulator approaches 72% of native performance, roughly comparable to 91% achieved by the target-specific Arm64EC. Risotto shows almost no improvements (10% vs QEMU 9%) when just offloading functions in `libc.so`.



(b) When offloading application-level code, Risotto can achieve significant improvement close to coMulator. However this requires a huge amount of manual modification to the benchmark source code. On the other hand, coMulator is fully automatic and requires literally ZERO modifications.



(c) Compared to Arm64EC, coMulator is target-neutral and supports all 9 combinations of guest/host ISAs. The numbers report the performance relative to native execution.

Fig. 15: Comparison of coMulator against SOTA works.

coMulator accelerates open-source libraries for closed-source applications. The results are given in Table IV. In fact, the aforementioned SPEC CPU 2017 is just one such example, which did not support `riscv64` until version 1.1.9 in 2022. Before that, manual setup is required to use it on `riscv64`.

TABLE IV: Real-world cases. The numbers show relative performance and speedup over QEMU.

Scenario	Application	coMulator Perf.
Target-specific code	Data Format	0.68 (6.18×)
	Multi Media	0.76 (6.91×)
Dependency lock-in	imagemagic	0.75 (2.88×)
	ffmpeg	0.59 (4.21×)
	zlib-flate	0.69 (4.31×)

Target-specific code: Legacy applications often suffer from deep coupling with their original ISAs. The two proprietary applications evaluated here all fail to be cross-built on ISAs beyond `x86_64` due to a combination of assembly, macros, and obsolete build configurations. By selectively offloading the ISA-agnostic functional logic while retaining the problematic code in the emulated guest context, coMulator successfully migrates these applications without any manual refactoring.

Dependency lock-in: It’s common to see proprietary applications released in binaries that have no versions for emerging target ISAs. We simulate this situation by using pre-compiled distributions of three free open-source applications. Without access to the application’s source code, coMulator achieves a speedup by transparently substituting their underlying shared libraries. This demonstrates a viable path for bootstrapping software ecosystems on emerging architectures, even for proprietary commercial software where source-level modifications are impossible.

These results confirm that coMulator maintains performance above an acceptable threshold (e.g., $> 50\%$ of native) while obviating exhaustive manual porting. This validates its robustness and efficacy in solving complex, real-world migration tasks.

V. RELATED WORK

We position coMulator within the cross-architecture execution landscape across three optimization pathways.

A. Hybrid Binary Translation

Binary translation has been extensively explored to bridge ISA gaps [5]. While QEMU prioritizes translation speed over code quality, various optimizations focus on code caching [8], [24], [25] and function/basic block inlining [26]–[28]. To enhance efficiency, researchers incorporate LLVM [20] into the translation loop via IR conversion [29], profile-guided optimization traces [30], [31], or pure static binary translation [6], [32]–[38]. To balance runtime overhead and binary quality, hybrid translation frameworks [7], [39]–[44] apply offline pre-translation to ahead-of-time (AOT) compile guest binaries into host code.

Unlike these bottom-up, binary-level that operate on stripped executables with limited semantic visibility, coMulator instead adopts a top-down, compile-time assisted paradigm. Operating on target-neutral LLVM IR modules allows coMulator to preserve rich semantic metadata and unlock aggressive, whole-program optimizations for native execution.

B. Native Shared Library Function Offloading

Another pathway intercepts guest library calls and offloads them to host-native execution. Early systems like FX!32 [9] and modern compatibility layers like Wine [45] rely on runtime wrappers to bridge cross-environment APIs. In cross-ISA scenarios, Box86 / Box64 [10] achieves substantial speedups by manually crafting wrappers to redirect standard and graphics library calls (e.g., libc, OpenGL, SDL) to the host. To scale beyond hard-coded libraries, recent research [11]–[13] introduce automated runtime linkers that mix guest programs and host libraries, though they still dictate that developers write explicit interface specifications using a dedicated Interface Definition Language (IDL) to describe function signatures.

Existing offloading techniques suffer from laborious manual efforts (IDL or wrappers) and generally lack support for emulation reentrancy (e.g., guest callbacks). In contrast, coMulator fully automates the offloading pipeline via compiler passes without manual annotations and introduces a context-switching mechanism to safely support nested, bi-directional cross-ISA execution, thus making it possible to offload vast application-level functions.

C. Source-Guided Partial Cross-Compilation

To bootstrap emerging software ecosystems, industry-leading solutions explore shifting cross-ISA coordination closer to the source-code level. Microsoft’s Arm64EC ABI [14] allows compiling legacy code into native ARM64 binaries that comply with `x86_64` calling conventions, serving as native thunks tightly interwoven with emulated code inside Windows’ Prism. Similarly, Apple’s Rosetta 2 [15] deeply integrates specialized compiler toolchains and runtime translation engines to execute mixed-architecture software stacks.

While methodologically aligned, industrial tools like Arm64EC and Rosetta 2 are closed-source, proprietary monoliths tied to specific OS platforms and custom target ABIs that require heavily modified compiler backends. Conversely, coMulator achieves architectural universality by decoupling ISAs at the target-neutral LLVM IR level, allowing both sides to strictly adhere to their native ABIs and enabling rapid bootstrapping across diverse open-source hardware combinations.

VI. CONCLUSION

This paper presents coMulator, a source-guided, callback-capable, and ABI-bridging design that coordinates compilation and emulation. With the help of compilers, coMulator facilitate function offloading for diverse custom library and application codebases. By leveraging `gyhcall`, coMulator enables ABI-neutral bidirectional cross-ISA invocations for

various guest/host combinations. Evaluation on SPEC CPU 2017 and real-world applications demonstrates that optimized coMulator consistently achieves significant speedup. These results validate coMulator as a robust and practical solution for streamlining the porting of applications across heterogeneous architectures.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive and insightful feedback. This work was supported in part by National Key R&D Program of China (Grant No. 2025YFB3003501), and by the National Natural Science Foundation of China (NSFC) under grants 62472462 and 62461146204. Experiments were conducted in part on the YatCC platform (<https://yatcc-ai.com/>), with its computing resources and runtime environments.

REFERENCES

- [1] L. Chris, “The golden age of compiler design in an era of HW/SW co-design,” 2021.
- [2] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.
- [3] “Rosetta 2 on a Mac with Apple silicon,” <https://support.apple.com/en-ca/guide/security/secebb113be1/web>, 2021.
- [4] mattwojo, “How emulation works on Arm,” <https://learn.microsoft.com/en-us/windows/arm/apps-on-arm-x86-emulation>, 2024.
- [5] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, “From Hack to Elaborate Technique—A Survey on Binary Rewriting,” *ACM Comput. Surv.*, vol. 52, no. 3, pp. 49:1–49:37, Jun. 2019.
- [6] X. Zhang, Q. Guo, Y. Chen, T. Chen, and W. Hu, “Hermes: A fast cross-ISA binary translator with post-optimization,” in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2015, pp. 246–256.
- [7] W. Li, X. Luo, Y. Zhang, Q. Meng, and F. Ren, “CrossDBT: An LLVM-Based User-Level Dynamic Binary Translation Emulator,” in *Euro-Par 2022: Parallel Processing*, ser. Lecture Notes in Computer Science, J. Cano and P. Trinder, Eds. Cham: Springer International Publishing, 2022, pp. 3–18.
- [8] W. Wang, P.-C. Yew, A. Zhai, and S. McCamant, “A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT),” in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 591–603.
- [9] R. Hookway, “DIGITAL FX!32 running 32-Bit x86 applications on Alpha NT,” in *Proceedings IEEE COMPCON 97. Digest of Papers*, Feb. 1997, pp. 37–42.
- [10] “Box86 / Box64,” <https://box86.org/>, Apr. 2021.
- [11] J. Tan, J.-m. Pang, and S.-b. Lu, “Using local library function in binary translation,” *Current Trends in Computer Science and Mechanical Automation*, vol. 1, pp. 123–132, 2018.
- [12] R. Gouicem, D. Sprokholt, J. Ruehl, R. C. O. Rocha, T. Spink, S. Chakraborty, and P. Bhatotia, “Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 107–122.
- [13] T. Spink and B. Franke, “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*, ser. LCTES 2024. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 1–11.
- [14] mattwojo, “Arm64EC for Windows 11 apps on Arm,” <https://learn.microsoft.com/en-us/windows/arm/arm64ec>, 2025.
- [15] “Reverse-engineering Rosetta 2 part1: Analyzing AOT files and Rosetta 2 runtime - Project Champollion,” <https://ffri.github.io/ProjectChampollion/part1/>, 2021.

- [16] C. Chen, S. Sugita, Y. Nada, H. Irie, S. Sakai, and R. Shioya, "Biotite: A High-Performance Static Binary Translator using Source-Level Information," in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC '25. New York, NY, USA: Association for Computing Machinery, Feb. 2025, pp. 167–179.
- [17] W. Wang, P.-C. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba, "Enabling Cross-ISA Offloading for COTS Binaries," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 319–331.
- [18] "Abi-aa/aapcs64 at main · ARM-software/abi-aa," <https://github.com/ARM-software/abi-aa>, 2025.
- [19] "x86 psABIs / x86-64 psABI · GitLab," <https://gitlab.com/x86-psABIs/x86-64-ABI>, Mar. 2025.
- [20] C. Latner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, 2004. CGO 2004., Mar. 2004, pp. 75–86.
- [21] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, Apr. 2005, p. 41.
- [22] "IPDL: Inter-Thread and Inter-Process Message Passing — Firefox Source Docs documentation," <https://firefox-source-docs.mozilla.org/ipc/ipdl.html>.
- [23] S. Narayan and C. Disselkoen, "Retrofitting Fine Grain Isolation in the Firefox Renderer."
- [24] D. Bruening and V. Kiriansky, "Process-shared and persistent code caches," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: Association for Computing Machinery, Mar. 2008, pp. 61–70.
- [25] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith, "Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications," in *International Symposium on Code Generation and Optimization (CGO'07)*, Mar. 2007, pp. 74–88.
- [26] J.-K. Lai and W. Yang, "Hyperchaining Optimizations for an LLVM-Based Binary Translator on x86-64 and RISC-V Platforms," in *50th International Conference on Parallel Processing Workshop*, ser. ICPP Workshops '21. New York, NY, USA: Association for Computing Machinery, Sep. 2021, pp. 1–9.
- [27] W. Wang, "Helper function inlining in dynamic binary translation," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 107–118.
- [28] Y.-P. You, T.-C. Lin, and W. Yang, "Translating AArch64 Floating-Point Instruction Set to the x86-64 Platform," in *Workshop Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP Workshops '19. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 1–7.
- [29] V. Chipounov and G. Candea, "Dynamically Translating x86 to LLVM using QEMU."
- [30] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: Association for Computing Machinery, Mar. 2012, pp. 104–113.
- [31] C.-C. Hsu, P. Liu, C.-M. Wang, J.-J. Wu, D.-Y. Hong, P.-C. Yew, and W.-C. Hsu, "LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends," in *2011 International Conference on Parallel Processing*, Sep. 2011, pp. 226–234.
- [32] A. Di Federico, M. Payer, and G. Agosta, "Rev.ng: A unified binary analysis framework to recover CFGs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017. New York, NY, USA: Association for Computing Machinery, Feb. 2017, pp. 131–141.
- [33] Lifting Bits, "Lifting-bits/mcsema: Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode," <https://github.com/lifting-bits/mcsema>, 2025.
- [34] —, "Lifting-bits/remill: Library for lifting machine code to LLVM bitcode," <https://github.com/lifting-bits/remill?tab=readme-ov-file>, 2025.
- [35] B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. Yang, "LLBT: An LLVM-based static binary translator," in *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '12. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 51–60.
- [36] S. B. Yadavalli and A. Smith, "Raising binaries to LLVM IR with MCTOLL (WIP paper)," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2019. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 213–218.
- [37] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing binary translation of dynamically generated code," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2015, pp. 68–78.
- [38] W. Wang, J. Wu, X. Gong, T. Li, and P.-C. Yew, "Improving Dynamically-Generated Code Performance on Dynamic Binary Translators," *SIGPLAN Not.*, vol. 53, no. 3, pp. 17–30, Mar. 2018.
- [39] H. Guan, E. Zhu, K. Chen, R. Ma, Y. He, H. Deng, and H. Yang, "A Dynamic-Static Combined Code Layout Reorganization Approach for Dynamic Binary Translation," *J. Softw.*, vol. 6, no. 12, pp. 2341–2349, 2011.
- [40] H. Guan, E. Zhu, H. Wang, R. Ma, Y. Yang, and B. Wang, "SINOF: A dynamic-static combined framework for dynamic binary translation," *Journal of Systems Architecture*, vol. 58, no. 8, pp. 305–317, Sep. 2012.
- [41] I.-C. Liu, I.-W. Wu, and J. J.-J. Shann, "Instruction Emulation and OS Supports of a Hybrid Binary Translator for x86 Instruction Set Architecture," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, Aug. 2015, pp. 1070–1077.
- [42] B.-Y. Shen, J.-Y. You, W. Yang, and W.-C. Hsu, "An LLVM-based hybrid binary translation system," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, Jun. 2012, pp. 229–236.
- [43] L. Sun, Y. Wu, L. Li, C. Zhang, and J. Tang, "A Dynamic and Static Binary Translation Method Based on Branch Prediction," *Electronics*, vol. 12, no. 14, p. 3025, Jan. 2023.
- [44] Z. Yang, X. Chen, L. Wang, W. Guo, D. Zhao, C. Yang, and F. Zhang, "MFHBT: Hybrid Binary Translation System with Multi-stage Feedback Powered by LLVM," in *Advanced Parallel Processing Technologies*, C. Li, Z. Li, L. Shen, F. Wu, and X. Gong, Eds. Singapore: Springer Nature, 2024, pp. 310–325.
- [45] "WineHQ - Run Windows applications on Linux, BSD, Solaris and macOS," <https://www.winehq.org/>, 2025.

A. Abstract

This artifact reproduces the main experimental results of the paper: **Figure 12**. For better reproducibility, the artifact is self-contained: it ships pre-built guest / host SPEC CPU2017 binaries with down-scaled, short-running inputs, and coMulator’s runtime components (including the modified QEMU and libc offloading components). Additional automatic scripts for running and visualization are also included. The experiment compares six coM configurations—baseline QEMU, coM, coM-p, coM-pg, coM-pgf, and coM-pgfi—against native execution across ten SPEC CPU2017 benchmarks.

Figure 12 of the paper uses “emulating x86-64 on AArch64” as the typical experimental setup. Since AArch64 machines are far less prevalent than x86-64 machines, we additionally provide an “emulating AArch64 on x86-64” experiment suite, which can be evaluated optionally; it yields results close to those reported in the paper and also demonstrates the neutrality and generality of coMulator.

B. Artifact check-list (meta-information)

- **Program:** source code available at <https://github.com/yhgu2000/artifact-micro2026-comulator>
- **Binary:** available at <https://doi.org/10.5281/zenodo.21530602>
- **Run-time environment:** Ubuntu 24.04
- **Hardware:** AArch64 or x86-64 host, ≥ 4 GB RAM
- **Metrics:** relative performance to native execution
- **Output:** Figure 12 (PDF) in the paper
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 3-6 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** custom non-commercial license
- **Workflow automation framework used?:** custom python scripts
- **Archived (provide DOI)?:** <https://doi.org/10.5281/zenodo.21530602>

C. Description

1) *How to access:* The experiment suite is available on Zenodo at <https://doi.org/10.5281/zenodo.21530602>.

The source code is not required for reproduction but is publicly available at <https://github.com/yhgu2000/artifact-micro2026-comulator> and will be continuously maintained.

2) *Hardware dependencies:* To reproduce the exact results of the x86_64-on-aarch64 emulation described in the paper, the evaluator needs an AArch64 machine with at least 4 GB of RAM and 64 GB of total system disk space. If this requirement cannot be met, we provide an alternative setup that runs the aarch64-on-x86_64 emulation on an x86-64 machine, which yields qualitatively similar results.

3) *Software dependencies:*

- The artifact ships pre-built binaries to maximize reproducibility; they can run directly on Ubuntu 24.04 (or inside an Ubuntu 24.04 container).

- Python 3 is required to execute the evaluation scripts (`apt install python3`).
- To regenerate the figures, install `matplotlib`, `numpy`, and `pandas` via `pip`.
- Optionally, to build the coMulator toolchain from source, `cmake`, `llvm-18`, and further dependencies are needed; please follow the instructions in the source repository’s README.

4) *Data sets:* The artifact includes substantially reduced-scale input cases for quick testing. As SPEC CPU2017 is a commercial software, to evaluate with the full inputs, the evaluators should use their own SPEC CPU2017 license.

D. Installation

Taking the x86_64-on-aarch64 setup as an example, download the `x86_64-on-aarch64.tgz` archive from the DOI above:

```
tar -xvf x86_64-on-aarch64.tgz
```

Then install the Python dependencies:

```
apt install python3
pip install matplotlib numpy pandas
```

Optionally, to evaluate the complete coMulator toolset, please follow the instructions in the source repository’s README (requires `cmake`, `llvm-18`, etc.). Note that this is not needed to reproduce the reported results.

E. Experiment workflow

Taking the x86_64-on-aarch64 setup as an example:

```
cd x86_64-on-aarch64
# run all emulations
./run # -> result.json
# run the native version
./run-native # -> result-native.json
# visualize the results
./plot # -> result.pdf (Figure 12)
```

F. Evaluation and expected results

The generated `result.pdf` is the paper’s Figure 12. The figure reports a geometric-mean (GMEAN) column, which indicates the overall efficiency of emulation. The relative performance may vary depending on the specific experimental environment, but it should be close to the values reported in the paper.

G. Experiment customization

To use full SPEC inputs, replace the reduced-scale files under `input/<benchmark>/` with the official SPEC CPU2017 inputs (requires a valid SPEC license) and modify the `run` script. However, please note that running such emulation may take weeks.