Partial Cross-Compilation and Mixed Execution for Accelerating Dynamic Binary Translation

Yuhao Gu
Sun Yat-sen University
Guangzhou, China
guyh9@mail2.sysu.edu.cn

Zhongchun Zheng
Sun Yat-sen University
Guangzhou, China
zhengzhch3@mail2.sysu.edu.cn

Nong Xiao*
Sun Yat-sen University
Guangzhou, China
xiaon6@mail.sysu.edu.cn

Yutong Lu
Sun Yat-sen University
Guangzhou, China
luyutong@mail.sysu.edu.cn

Xianwei Zhang*
Sun Yat-sen University
Guangzhou, China
zhangxw79@mail.sysu.edu.cn

Abstract

With the growing diversity of instruction set architectures (ISAs), cross-ISA program execution has become common. Dynamic binary translation (DBT) is the main solution but suffers from poor performance. Cross-compilation avoids emulation costs but is constrained by an "all-or-nothing" model—programs are either fully cross-compiled or entirely emulated. Complete cross-compilation is often unfeasible due to ISA-specific code or missing dependencies, leaving programs with high emulation overhead.

We propose a hybrid execution system that combines compilation and emulation, featuring a selective function offloading mechanism. This mechanism establishes cross-environment calling channels, offloading eligible functions to the host for native execution to reduce DBT overhead. Key optimizations address offloading costs, enabling efficient hybrid operation. Built on LLVM and QEMU, the system works automatically for both applications and libraries. Evaluations show it achieves up to 13x speedups over existing DBT, with strong practical value.

Keywords: binary translation, emulation, cross-compilation, ABI, function offloading

1 Introduction

With the slowdown of Moore's Law[7, 9], we have witnessed the rapid proliferation of diverse instruction set architectures (ISAs), such as x86, ARM and RISC-V. As a result, the demand for cross-ISA execution has become increasingly prevalent. Emulators and dynamic binary translation (DBT) have emerged as the mainstream solutions, widely adopted to support legacy applications and port applications to new ISAs[17], such as Rosetta 2 for MacOS[1] and Prism for Windows on Arm[12]. However, DBT consistently faces severe performance challenges: since a single guest[1] instruction often requires translation into multiple host instructions, and further runtime translation itself incurs significant overhead, emulated execution can be dozens of times slower than native execution [11, 15, 18].

Meanwhile, most machine code executed at runtime is generated by compiling from high-level languages (e..g. C/C++) rather than written directly in assembly, which implies a high degree of target-independency. By cross-compiling[2] the source code to the host ISA, one can obtain a native executable that completely eliminates the interpretation overhead of DBT. Yet, existing approaches confined to an "all-or-nothing" paradigm: either the entire program can be cross-compiled and executed natively, or cross-compilation fails, forcing the whole program to fall back to DBT emulation. In practice, complete cross-compilation is often infeasible. For instance, the source code may include platform-specific macros or assembly codes, relying on middleware libraries unavailable on the host ISA, or simply lack accessible source code — such as legacy applications and commercial closed-source software. In such scenarios, it becomes unavoidable to endure the heavy overhead of full DBT emulation.

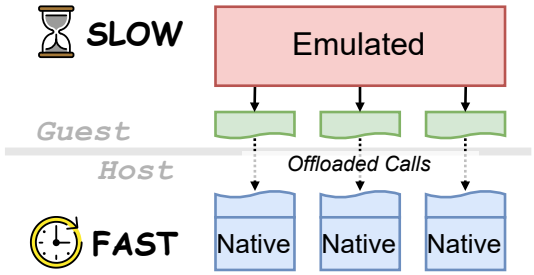

Figure 1. TECH-NAME mechanism enables partial offloading of guest code to the host side for native execution, thereby bypassing the emulation overhead of DBT.

*Corresponding author.
[1]Throughout 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.

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

This paper presents TECH-NAME, a function offloading mechanism that enables cross-compilation and hybrid code execution for different guest and host ISAs to accelerate DBT emulation, as shown in Figure 1. Such offloading require supports from both compile-time and runtime. At compile-time, TECH-NAME extracts eligible guest functions and generates semantically equivalent host functions. At runtime, guest-side calls to these functions are forwarded to the host side for native execution, thereby bypassing the substantial emulation overhead of DBT. TECH-NAME successfully addresses two key challenges in offloading: first, the discrepancy between guest and host ABIs, which is resolved through **calling conversion** implemented by collaborative stubs between the guest and host; second, the interleaving of call chains caused by offloaded host functions calling back to the guest, which is tackled via an elegant **emulation reentrancy** at runtime. Additionally, since switching execution between guest and host modes incurs much higher overhead than direct function calls, we introduce three key **optimization techniques** to mitigate this cost and harvest performance benefits: global reference tables (GRT), fast calling paths (FCP), and partial function outlining (PFO). Beyond applications, TECH-NAME can also be applied to shared libraries, enabling speedups for downstream binaries without modification to them. This benefits closed-source applications, and is particularly suitable for commercial software who uses open-source basic libraries, thus holding strong practical value.

To validate the effect of TECH-NAME, we implemented a prototype tool set, IMPL-NAME, based on the state-of-the-art open-source infrastructures, LLVM[10] and QEMU[6]. IMPL-NAME consists of the compile-time and runtime tools, supporting the acceleration for both emulating x86-64 on AArch64 and vice versa. Our evaluaton spans the LLVM Test Suite[4], NAS Parallel Benchmarks[2], along with multiple dynamicly linked real-world applications. Experimental results demonstrate that, IMPL-NAME can accelerate execution by up to 13.03x on AArch64 and 18.91x on x86-64, with an a geometric mean of 3.03x (AArch64) and 3.18x (x86-64), validating TECH-NAME's effectiveness in accelerating cross-ISA DBT. All implementation and experimental codes have been made publicly available, and we continue to refine the the system for deployment in large-scale real-world applications.

In summary, this paper makes the following contributions:

1. We propose TECH-NAME, a mechanism that combines the compile-time and runtime to offload guest function calls to the host side for native execution, detailing the guest-to-host forwarding and host-to-guest callbacks.
2. We further introduce three optimizations, GRT, FCP and PFO, that mitigate the high overhead of guest-host switching and deliver significant performance improvements.
3. We implement IMPL-NAME, a prototype for TECH-NAME with all the optimizations, and evaluate it using benchmarks and real-world applications. The results show that IMPL-NAME can achieve speedups of up to an order of magnitude, validating its practical effectiveness in accelerating cross-ISA emulation for DBT.

2 Background and Motivation

2.1 Background

2.1.1 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 must be performed dynamically. This leads to Dynamic Binary Translation (DBT), which, despite its flexibility, suffers from substanial performance overhead. This inefficiency arises from 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 efficiency. Second, the translation process itself is computationally expensive, and to ensure real-time translation, DBT typically forgoes deep optimizations during translation, further limiting performance. Consequently, cross-ISA emulation via DBT can be dozens of times slower than the native execution, severely restricting its applicability in real-world scenarios.

2.1.2 Cross-compilation. Given DBT's inherent limitations, cross-compilation is often considered as a superior alternative for migrating applications to new ISAs. Nowadays, most applications are developed in high-level languages like C and C++, whose source code is largely ISA-agnostic. It is thus theoretically possible to recompile these applications directly into native binaries for the target ISA via the native compilation toolchain. In practice, however, many applications are not fully target-independent. Their source codes often embed components tightly coupled to the original ISA, mainly manifesting in the following scenarios:

- Target-specific macros: source code frequently contains ISA-related snippets conditionally compiled via preprocessor macros (e.g., `#ifdef x86-64`). These macros are typically used to leverage ISA-specific features such as register layouts, instruction set extensions. If the new ISA is not explicitly supported, cross-compilation may yield missing functionalities or incorrect behaviors.
- Target-specific assembly code: to pursue performance or perform low-level operations, applications often include ISA-specific assembly code. Such code can only be executed on the original ISA and is typically rejected by cross-compilation toolchains, preventing a successful build.

- Unavailable dependencies: even when the application itself is ISA-agnostic, its direct or indirect dependencies may not be available on the target ISA. This issue is further exacerbated for closed-source commercial libraries that lack support for the new architecture.

For these scenarios, existing compiler toolchains often fall into an "all or nothing" dilemma — either extensive source code modifications must be performed to eliminate ISA dependencies, or cross-compilation must be abandoned which falls back to DBT emulation with its prohibitive overhead.

2.2 Opportunities and Challenges

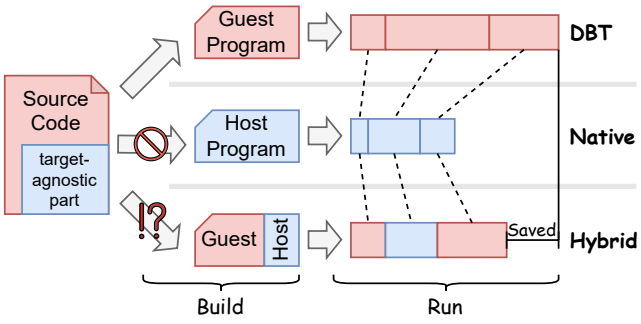

Figure 2. Though the whole source code can't be cross-built to the host target, the target-agnostic part of the source may still be exploited to accelerate DBT.

2.2.1 Coordinated Compilation and Emulation.

The independence inherent in most ISA-specific applications' code bases has not been fully explored. Recent studies [8, 13, 14, 16] have studied bypassing DBT by offloading certain guest function calls to the host semantically equivalent functions, thereby significantly accelerating the emulation speed. For instance, one recent work reported up to a 28x speedup [13]. However, existing offloading methods are largely constrained: they either rely on built-in helper functions [14] or require developers to manually register functions via Interface Definition Language (IDL) [8, 13]. Such approaches demand in-depth knowledge of both the ISA and the target functions, greatly limiting practicality require developers to be very familiar with the ISA and the offloaded functions, making it difficult to be applied to other code bases. As a result, their applicability remains confined to a narrow set of standard or common libraries, such as libc, Sqlite, rather than extending to broader, real-world code bases.

From the DBT perspective, the fundamental difficulty in function offloading lies in the loss of key semantic information at the binary level. Compilation and optimization strip away crucial metadata from the source, such as function signatures, parameter and return value types, all of which are essential for smooth cross-ISA invocation. For example, without determining the types of functions, it is difficult to correctly pass parameters and convert stack frame layout between the guest ISA and the host ISA, and thus impossible to safely offload function calls from the emulation to native execution.

A promising alternative is to shift offloading to a source-level perspective. At the source code level, full type information and structural semantics are preserved, enabling accurate and reliable identification of offloadable functions. Through static analysis, ISA-agnostic code blocks can be cross-compiled to the host ISA, while ISA-specific code is compiled as usual to the original guest ISA. The final program thus retains full functionality on the guest ISA, but with performance-critical portions offloaded to native host execution. This insight motivates the design of **coordinated compilation and emulation** (*Hybrid* in Figure 2) to enable automated and generalizable function offloading, which makes it applicable to diverse code bases without requiring intrusive manual efforts to fully automate such offloading and generally expand its usage to all code bases.

2.2.2 Challenges of Guest-Host Invocations and Callbacks.

Realizing automated and generalizable source-guided offloading is far from straightforward. It introduces two fundamental challenges that must be addressed to ensure correctness, robustness, and performance:

Guest-Host ABI Discrepancies. The functions in source code have the same parameter and return value types regardless the ISAs, laying a foundation for cross-ISA interaction. But directly moving guest functions to the host side is not feasible. The core obstacle lies in the Application Binary Interfaces (ABIs) of different ISAs. The ABI governs low-level function call conventions, including parameter-passing, register usage, and stack frame layout [3, 5]. Without proper calling conversion, offloaded execution risks runtime failures such as incorrect parameter delivery, stack corruption, or misaligned return values. Existing approaches [8, 13, 14] often rely on manual intervention to handle such conversions, which is error-prone and impractical at scale. In addition, functions usually reference external globals (e.g., variables, constants, or other functions). These globals must be properly propagated to the host side to preserve semantic correctness during offloaded execution, adding another layer of complexity to the calling conversion process.

Host-to-Guest Callbacks and Emulation Reentrancy. Offloaded host functions are not always self-contained; they may need to call back guest functions like invoking function pointers passed from the guest. This necessitates establishing a bidirectional calling channel between host-native code and guest-emulated code. Current solutions typically restrict themselves to callback-free scenarios, such as leaf functions that do not invoke others. For complex call chains involving callbacks remain unsolved in general. Handling these scenarios requires carefully managing context switches between

emulated and native execution modes while preserving runtime state consistency. Beyond callbacks, non-local control flows in general-purpose code (e.g. `longjmp`, exceptions) introduce additional complications. These constructs disrupt standard call-return semantics, making cross-ISA invocation particularly error-prone and difficult to generalize.

A combination of code compilation and runtime support is critical to tackle the aforementioned challenges. Moreover, since crossing guest-host boundaries inevitably incurs overhead, optimizations are necessitate to mitigate performance penalties and share key insights into its implementation.

3 Design

3.1 Overview

Figure 3 outlines the general framework of a proposed mechanism. To migrate specific function executions from an emulated environment to a native host environment, coordinated support is required during both compilation and runtime phases.

In the compilation stage, suitable target functions are identified, with their core logic extracted and adapted for native host execution. Since the host's ABI differs from the emulated environment, intermediate bridging stub functions are retained in the emulated side to manage cross-environment function call conversions. Extracted functions may include references to other components in the emulated environment, requiring reverse conversion support for callback scenarios.

At runtime, the stub functions handle call conversion and triggers environment switching via dedicated interface calls, which transfer execution control to the host-side adapted functions. Interleaved call chains from host to emulated executions may arise from cross-environment callbacks, necessitating support for nested environment switching. A complementary interface is introduced to facilitate switching back to the host environment upon callback completion. Critical requirements include non-intrusiveness, ensuring the mechanism does not disrupt normal emulation operations.

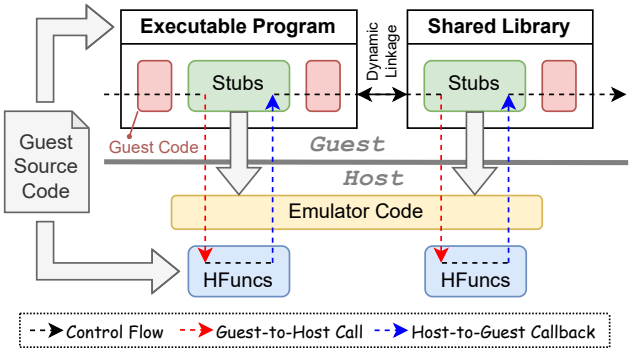

Figure 3. The overview of TECH-NAME mechanism.

This mechanism is also applicable to library-level functions, enabling performance optimization by migrating library function calls. For dynamically linked libraries, optimization can be achieved by replacing the library binaries directly, without modifying or recompiling the target application. Binary compatibility of shared libraries ensures functional correctness after updates, allowing this approach to benefit both open-source and closed-source applications. This feature is particularly useful for scenarios involving commercial software integrated with standard libraries, offering practical applicability.

3.2 Calling Conversion

Calling conversion is jointly handled by stub functions in the emulated environment and offloaded functions derived from original components, instead of relying solely on host-side binary adjustments. The latter would bind adapted functions tightly to specific emulated architectures and internal emulator designs. By allocating part of the conversion task to the emulated side, the proposed mechanism achieves a concise, versatile, and maintainable interface.

In the emulated environment, the original function implementation is replaced with bridging code. When the function is invoked, the stub prepares arguments and other things and send them to the offloaded functions on the host side. This data exchange method allows adapted functions to operate independently of guest or host ISAs, ensuring architecture neutrality and implementation simplicity. A similar approach applies to data exchange for cross-environment callbacks, though a key distinction exists: guest functions called indirectly (e.g., via function pointers) lack built-in unpacking capabilities, requiring additional bridging code in the emulated environment. Adapted functions may reference global elements (including required callback components) that reside in the emulated address space and must be transmitted to the host.

3.3 Emulation Reentrancy

Host-executed functions may call-back emulated components, creating an "emulation reentrancy" issue. This interacts with QEMU's runtime stack layouts. We extend QEMU's built-in switching mechanism to manage cross-environment transitions. To fix this, we build host function contexts on the guest stack and the host stack frames, keeping stack consistency and matching the emulated program's logic. This mechanism supports both guest-to-host calls and host-to-guest callbacks.

Auxiliary functions isolate host and emulated execution, ensuring strong compatibility. Even if the emulated program crashes, QEMU stays stable—host functions have no unintended side effects on QEMU's native operations. LLVM IR resolves underlying interface differences, making it the ideal layer for implementing this mechanism—compiled host code reliably operates on emulated data.

3.4 Optimizations

Cross-environment calls are costly—only long functions benefit. To optimize for short functions, we introduce three key methods.

Global Reference Table (GRT): Basic design incurs unnecessary construction of same data for each cross-side function calls. GRT pre-stores them in global constants to eliminates those costs.

Fast Calling Path (FCP): FCP lets offloaded functions call each others directly without switching to the guest emulation. It can effectively reduce cross-boundary overhead.

Partial Function Outlining (PFO): PFO expands offloadable functions, making originally un-offloadable functions offloadable. Hence we support offloading code using variadic calls and other cases. For context-sensitive code, its complement is split instead of themselves.

4 Evaluation

4.1 Experimental Methodology

4.1.1 Testbed. We evaluate IMPL-NAME, the `TECH-NAME` prototype, on both x86-64 and AArch64 machines. The detailed configurations are as listed in Table 1.

Table 1. Configurations of testbed.

	x86-64	AArch64
CPU	AMD Ryzen 9 5950X @ 3.4GHz	Phytium FT-2000+/64 @ 2.2GHz
RAM	32GB	128GB
OS	Ubuntu 24.04	
LLVM/Clang	18.1	
QEMU	9.1.50 (modified)	

4.1.2 Workloads. Our evaluation assesses monolithic benchmark programs, where IMPL-NAME is applied directly to their source codes for offloading functions. Table 2 summarizes the full list of workloads, which include three customized micro-benchmarks, four randomly chosen C programs from the LLVM test suite [4], and the complete set of NAS parallel benchmarks [2].

4.1.3 Metrics. We evaluate our system using two primary metrics: 1) performance, and 2) program size. The following schemes are compared:

- `native` : programs built and executed natively.
- `qemu` : original guest programs emulated with the vanilla QEMU on the host machine.
- `TECH` : IMPL-NAME with baseline implementation of `TECH-NAME`, i.e., without any enabled optimizations.
- `TECH-g` : IMPL-NAME using `TECH-NAME` enabled with Global Reference Table (GRT) optimization.

Table 2. Evaluated workloads (Wkld).

Source	Wkld	Source	Name	Wkld
Customized	matpowsum	NAS Parallel Bench	BT	npbbt
	cjson		CG	npbcg
	lua		EP	npbep
LLVM test-suite	obsequi		FT	npbft
	oggenc		LU	npblu
	sgefa		MG	npbmg
	viterbi		SP	npbsp
			IS	npbis

- `TECH-gf` : IMPL-NAME using `TECH-NAME` with both GRT and Fast Calling Path (FCP) optimizations.
- `TECH-gfp` : IMPL-NAME using `TECH-NAME` with all optimizations: GRT, FCP and Partial Function Outlining (PFO).

4.2 Performance Improvement

Figure 4 presents the measured speedup (left y-axis) and wall time (right y-axis) for all workloads on the AArch64 machine. It can be seen that the `qemu` emulation is significantly slower than `native` execution with a geometric mean slowdown of 13.23x. When applying IMPL-NAME along with all its optimizations (i.e., `TECH-gfp`), the emulation performance is improved by a geometric mean of 3.03x (up to 13.03x).

A closer examination of the data in the figure reveals several interesting observations. First, the baseline version of IMPL-NAME (`TECH`) achieves a geometric mean speedup of 1.29x, while the sole GRT optimization (`TECH-g`) does not lead to a significant change (still around 1.29x). This indicates that the primary overhead of cross-guest-host invocations does not lie in the arguments and return value conversions, but rather in the internal works of QEMU, including system call handling, context switching, and others. Consequently, reducing the number of guest-host boundary crossings (`TECH-gf`) can significantly mitigate overhead, which is corroborated by the optimization effect of FCP (1.54x).

Second, it can be observed that most significant speedup comes from the final PFO optimization. This indicates that there were indeed many hot functions containing problematic instructions, which prevent them from being offloaded to the host side. Our in-depth investigation of these applications reveals that calls to variadic functions are the primary cause. Source codes often include safety check statements using functions like `printf`, which hinder the offloading of the entire function but are usually not triggered at runtime. The PFO optimization effectively addresses this case.

Third, negative optimizations do exist: in the two workloads `cjson` and `lua`, IMPL-NAME results in worse performance even with all optimizations being enabled. This is because the two applications contain a large number of short functions that call each other. Offloading them not only fails

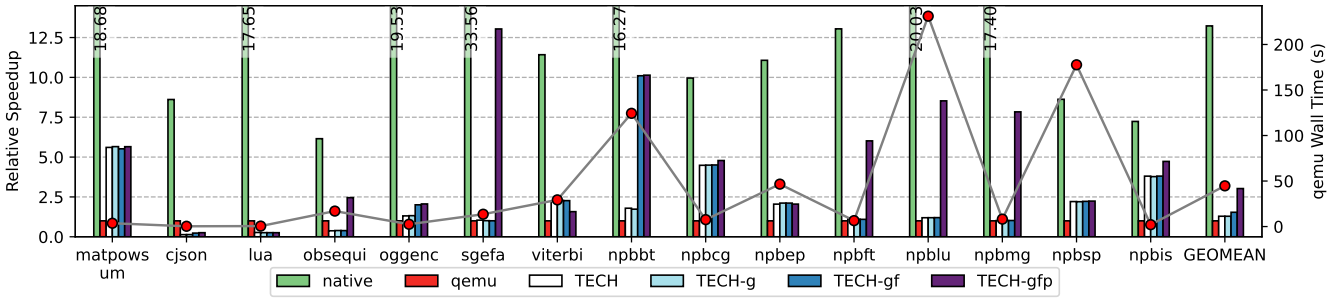

Figure 4. Performance of the x86-64 emulation on AArch64. The bars (left y-axis) show speedup relative to original QEMU, with the folding line (right y-axis) showing its wall time.

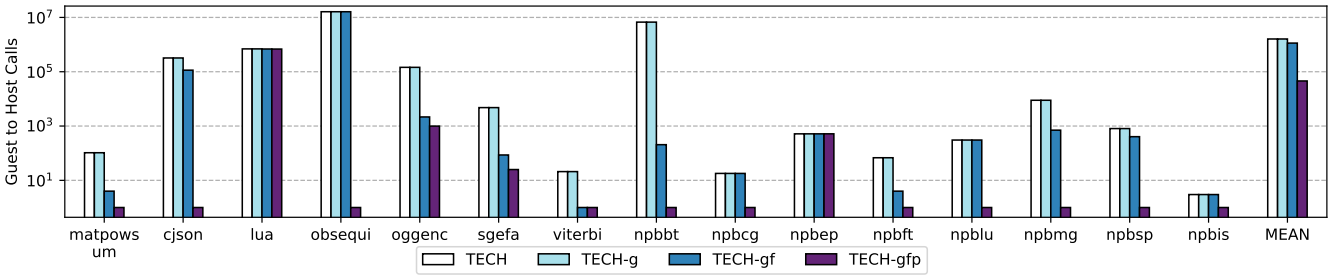

Figure 5. Number of guest-to-host calls during the execution of all the workloads.

to bring much speedup but also causes frequent crossings of the host-guest boundary. This insight conveys that the offloading is not a guaranteed gain; we must wisely select which function to offload. Our current prototype adopts a very simple strategy of filtering out functions whose number of basic blocks and instructions exceeding a certain threshold. More sophisticated strategies are possible, such as better cost models and profiling, which we leave for future work. However, it must be noted that TECH-NAME offloading is incremental and allows degradation to pure emulation in the worst-case scenario. Therefore, the practical performance of IMPL-NAME can definitely be no worse than that of the original qemu .

4.3 Function Statistics

4.3.1 TECH-NAME Invocation Count. Figure 5 gives the count of guest-to-host calls during the execution of all workloads in Figure 4. It's quite obvious that there is a strong positive correlation between these two sets of statistics. This insight confirms the conjecture that the overhead of TECH-NAME mainly stems from crossing the host-guest boundary.

Just as expected, the GRT optimization poses no effect to the invocation count, whereas the FCP optimization significantly reduces the invocation number. For instance, the count drops from 6,713,003 to 206 in the *npbbt* workload. However, greater effect comes from its conjunction with the PFO optimization, where the number decreases by more than an order of magnitude. Especially, 11 workloads triggers only one single TECH-NAME invocation after PFO optimization.

One abnormal point is *cjson*. After PFO optimization, its invocation count drops to one, but the performance remains to be inferior to vanilla qemu , though it does show improvement compared to the baseline TECH (1.85x speedup). After investigation, we attribute this to the callbacks to *libc* functions which is not offloaded by IMPL-NAME. Since most *cjson* functions are very light with little computation, the time saved by native execution is far less than that spent on callbacks. This finding inspires us to preferably apply TECH-NAME to the lower half of the entire software stack in a bottom-up manner to eliminate host-to-guest callbacks as much as possible.

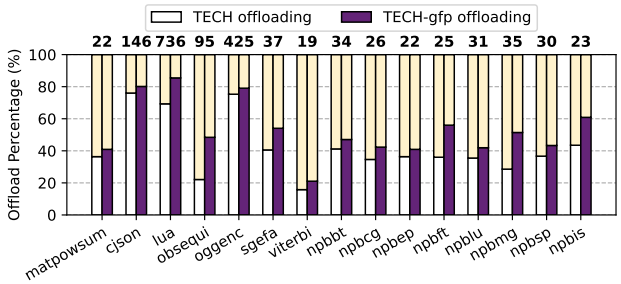

Figure 6. Function offloading coverage results. The values above the bars are the total function numbers.

4.3.2 Function Offloading Coverage. The function coverage results show how many functions are offloaded by IMPL-NAME. The more function offloaded, the higher chance that host-to-guest callbacks may be eliminated with by the FCP optimization. Figure 6 clearly shows that PFO increases the function coverage. For instance, the number of offloaded functions increased from 21 to 46 in the *obsequi* workload, which reduces TECH-NAME invocations from 16,206,473 to just 1, yielding a 6.34× speedup compared to the baseline TECH .

There is no strong correlation between function coverage and performance improvement. This is because the additionally offloaded functions may not be hot or even used. A typical example is the *lua* workload, where a significant improvement in function coverage doesn't bring any changes to its performance results. This inspires us to explore the combination of profiling methods to selectively offload hot functions in the future.

4.4 Additional Studies

4.4.1 Emulate x86-64 on AArch64. The universal design of TECH-NAME makes it easy to be extended to other guest/host ISA combinations. So, without much efforts, we additionally echo the performance of emulating AArch64 on the x86-64 machines, with the results shown in Figure 7. The performance trend on x86-64 is consistent with that on AArch64 with a slightly better geometric mean(3.18x) and maximum (18.91x). This result fully demonstrates the stability and reliability of TECH-NAME, and also reflects its low sensitivity to specific ISA combinations.

4.4.2 Effects on Shared Library. As mentioned before, TECH-NAME is not limited to applications but can also be extended to the libraries. To this end, we supplement acceleration experiments targeting two open-source libraries: *libpng* and `zlib`, along with four applications — *apng2gif*, *optipng*, *imagemagick*, and *zlib-flate* — to test the actual effect of offloading library functions. It is worth noting that all applications are pre-built binaries directly downloaded from the Ubuntu APT repository and dynamically linked. So the acceleration is performed solely by replacing the corresponding library files, without modification to the applications themselves. The relevant experimental results are shown in Table 3.

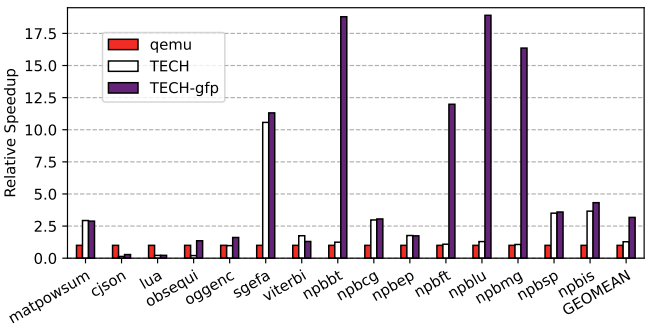

Figure 7. Relative performance of the AArch64 emulation on x86-64.

It is evident that accelerating the *zlib* library significantly performs better than that of *libpng*. Moreover, the acceleration effects of different libraries are additive: in the scenario where *imagemagick* converts compressed PNG images, accelerating both *libpng* and *zlib* altogether achieves an overall performance improvement of 3.96x, which is higher than the effects of accelerating either library alone (3.87x and 1.20x). These results indicate that TECH-NAME can still improve the overall performance of applications by optimizing the dynamic libraries they depend on, even when the application source code is unavailable. What's more, library-level acceleration is universal, i.e., once a basic library is optimized, all applications relying on it can benefit. This significantly expands the scope of application and influence of TECH-NAME.

5 Conclusion

Conventional DBT incurs substantial emulation overhead, while existing cross-compilation techniques often adhere to an "all-or-nothing" paradigm, rendering them impractical for legacy or closed-source software with target-specific elements. To address the challenges, we propose TECH-NAME, a function offloading mechanism that coordinates compile-time and runtime supports. TECH-NAME extracts eligible guest functions, generates their equivalent host versions, and forwards calls to them, thus bypassing expensive DBT emulation. The design resolves ABI discrepancies via calling conversion and ensures correct execution across interleaved chains through emulation reentrancy. Furthermore, our optimizations (GRT, FCP, PFO) effectively reduce guest-host switching costs. We implement the ideas in IMPL-NAME, a prototype based on LLVM and QEMU. Evaluation results demonstrate substantial performance improvements, achieving up to 13.03x speedup on AArch64 and 18.91x on x86-64, with geometric means of 3.03x and 3.18x, respectively. These results validate the effectiveness and robustness of TECH-NAME in accelerating DBT. We are working on applying TECH-NAME to real-world application workloads in large scale currently. Looking forward, we plan to explore deeper compiler-emulator co-optimizations, and more adaptive offloading strategies guided by workload characteristics.

Table 3. Shared library acceleration

Offloaded Lib.	Downstream App.	Speedup
libpng	apng2gif	1.04x
	optipng	1.05x
	imagemagick	1.20x
zlib	imagemagick	3.87x
	zlib-flate	16.48x
libpng+zlib	imagemagick	3.96x

References

[1] 2021. Rosetta 2 on a Mac with Apple Silicon. https://support.apple.com/en-ca/guide/security/secebb113be1/web.

[2] 2024. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html.

[3] 2025. Abi-Aa/Aapcs64 at Main · ARM-software/Abi-Aa. https://github.com/ARM-software/abi-aa/tree/main/aapcs64.

[4] 2025. Llvm/Llvm-Test-Suite. LLVM.

[5] 2025. x86 psABIs / x86-64 psABI · GitLab. https://gitlab.com/x86-psABIs/x86-64-ABI.

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

[7] Lattner Chris. 2021. The Golden Age of Compiler Design in an Era of HW/SW Co-Design.

[8] Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo C. O. Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. 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 (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 107–122. doi:10.1145/3567955.3567962

[9] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. Commun. ACM 62, 2 (Jan. 2019), 48–60. doi:10.1145/3282307

[10] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In International Symposium on Code Generation and Optimization, 2004. CGO 2004. 75–86. doi:10.1109/CGO.2004.1281665

[11] Wei Li, Xiaohui Luo, Yiran Zhang, Qingkai Meng, and Fengyuan Ren. 2022. CrossDBT: An LLVM-Based User-Level Dynamic Binary Translation Emulator. In Euro-Par 2022: Parallel Processing (Lecture Notes in Computer Science), José Cano and Phil Trinder (Eds.). Springer International Publishing, Cham, 3–18. doi:10.1007/978-3-031-12597-3_1

[12] mattwojo. 2024. How Emulation Works on Arm. https://learn.microsoft.com/en-us/windows/arm/apps-on-arm-x86-emulation.

[13] 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 2024). Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3652032.3657565

[14] Jie Tan, Jian-min Pang, and Shuai-bing Lu. 2018. Using Local Library Function in Binary Translation. Current Trends in Computer Science and Mechanical Automation 1 (2018), 123–132.

[15] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2016. A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT). In 2016 USENIX Annual Technical Conference (USENIX ATC 16). 591–603.

[16] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17). Association for Computing Machinery, New York, NY, USA, 319–331. doi:10.1145/3081333.3081337

[17] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From Hack to Elaborate Technique—A Survey on Binary Rewriting. ACM Comput. Surv. 52, 3 (June 2019), 49:1–49:37. doi:10.1145/3316415

[18] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. 2015. Hermes: A Fast Cross-ISA Binary Translator with Post-Optimization. In 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 246–256. doi:10.1109/CGO.2015.7054204