ARMing x86 Games: Accelerating Binary Translation Using Software-Only Validated Flag Speculation

James Yen[1], Jiarui Wang[1], Zhibai Huang[1], Zhixiang Wei[1], Ziyang Zhang[1], Chen Chen[1], Senhao Yu[1], Yun Wang[1], Hao Wang[2], Zhengwei Qi[1]
[1]Shanghai Jiao Tong University
[2]Stevens Institute of Technology

ABSTRACT

As ARM architecture becomes more prevalent in personal computers, users transitioning from x86-based Windows platforms face compatibility issues, particularly with x86 applications like games. Existing solutions, such as QEMU, Box64, and Apple's Rosetta 2, either incur high latency, face performance bottlenecks, or are limited to specific ecosystems. A key challenge remains the efficient translation of x86 status flags, which impacts performance.

We propose a novel optimization method that enhances compatibility and performance by leveraging software-only strategies tailored to ARM hardware features. Using data flow analysis, our approach identifies when ARM's hardware flags can replace x86 flags, reducing reliance on software emulation and lowering translation overhead. This results in improved speed and compatibility for x86 applications on ARM, supporting demanding applications like games across x86 and ARM platforms without specialized hardware. Experimental results show significant performance gains, with computational tasks improving by up to 18%, and graphics rendering (FPS) also increasing by up to 18%. In particular, real-world testing on popular Steam titles demonstrates FPS improvements ranging from about 7% to over 12%.

CCS CONCEPTS

• **Computer systems organization → Reduced instruction set computing**; • **Software and its engineering → Compilers**; **Software performance**.

KEYWORDS

Binary Translation, x86, ARM, EFLAGS

ACM Reference Format:
James Yen[1], Jiarui Wang[1], Zhibai Huang[1], Zhixiang Wei[1], Ziyang Zhang[1], Chen Chen[1], Senhao Yu[1], Yun Wang[1], Hao Wang[2], Zhengwei Qi[1] . 2025. ARMing x86 Games: Accelerating Binary Translation Using Software-Only Validated Flag Speculation. In *The 23rd Annual International Conference on Mobile Systems, Applications and Services (MobiSys '25), June 23–27, 2025, Anaheim, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3711875.3729163

MobiSys '25, June 23–27, 2025, Anaheim, CA, USA

ACM ISBN 979-8-4007-1453-5/2025/06
https://doi.org/10.1145/3711875.3729163

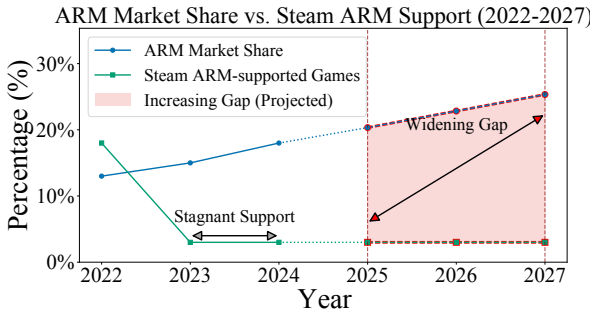


Figure 1: Trends in ARM market share and Steam ARM support (2022–2027).

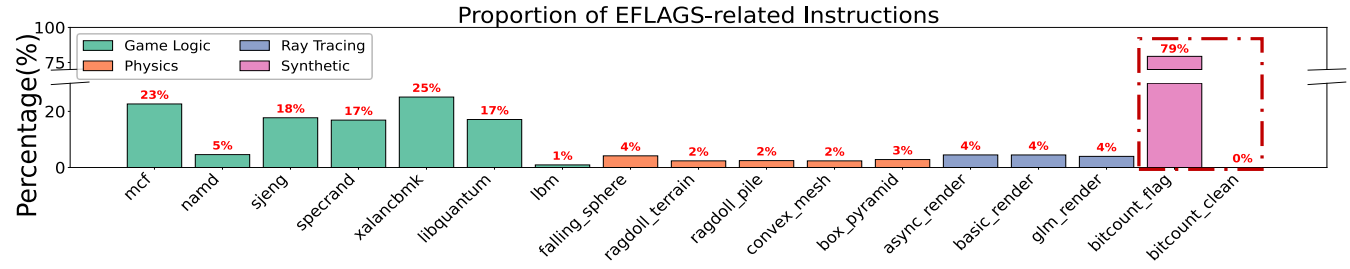
1 INTRODUCTION

The Advanced RISC Machine (ARM) architecture has secured dominance in the smartphone and tablet markets, holding 99% of the smartphone segment and powering nearly all high-end devices. This supremacy is due to ARM's exceptional power efficiency and thermal performance [5, 8], which are vital for battery-operated devices, especially in high-demand applications like gaming where low power usage extends battery life. In 2022, mobile gaming generated 50% of total game revenues, outpacing both console and PC gaming [31], highlighting ARM's crucial role in mobile gaming [15, 17, 36].

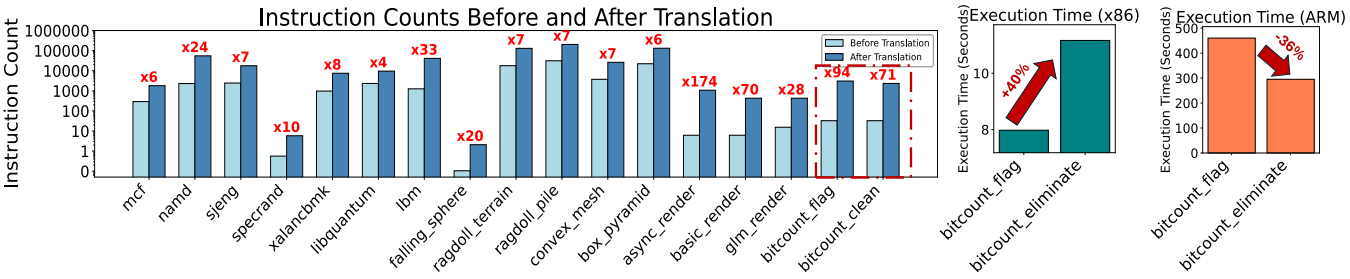
ARM processors use a Reduced Instruction Set Computer (RISC) design, executing simpler instructions than Complex Instruction Set Computer (CISC) processors. This leads to lower energy use and less heat generation [26, 33], allowing longer gameplay without draining the battery [2, 10, 37].

ARM's success in mobile has extended to personal computing. ARM-based processors like Apple's M1 and M2 chips offer strong performance and energy efficiency in desktops and laptops [4, 23]. However, software support, particularly for gaming, has not kept pace with ARM's growth in personal computing. As shown in Figure 1, despite ARM's increasing market share in PCs, native ARM game support has stagnated, indicating a disconnect between hardware potential and software development. While companies such as Nintendo [29] and Steam [14] are working on ARM compatibility, most PC games remain optimized for x86 architectures. This presents a challenge in integrating ARM into markets traditionally dominated by x86 processors [1, 11, 20, 22].

A major barrier is the compatibility with existing software. While new games can target ARM, the vast library of existing games is vital to the ecosystem. Enabling x86 games to run on ARM without



(a) Percentage of EFLAGS Operations in Real-World Examples



(b) Instruction Bloating After Binary Translation **(c) Example of EFLAGS Importance**

Figure 2: Analysis of EFLAGS operations and their implications in binary translation and real-world contexts.

modifications would preserve gaming history and leverage ARM's mobile dominance to attract desktop and gaming users.

To solve this, a method is needed to run x86 applications, especially games, on ARM devices efficiently. Creating separate versions for each architecture is costly and time-consuming, particularly for legacy code. Additionally, maintaining multiple versions can lead to inconsistencies and increased development complexity.

Binary translation is a promising solution, converting x86 instructions to ARM instructions at runtime. This allows legacy games to run on ARM without changes, with dynamic binary translation enhancing performance. Existing tools like QEMU [7] offer broad compatibility by fully emulating x86 on ARM but add significant computational overhead, making them unsuitable for real-time gaming. Box64 [25] improves performance through dynamic translation but still faces translation overhead issues. Apple's Rosetta 2 [3] optimizes translation using hardware-specific features but is limited to Apple devices.

A key bottleneck in binary translation is managing the EFLAGS register in x86. Minimizing the overhead of EFLAGS emulation is a central challenge addressed in this paper.

Key Insight and Methodology. We discovered that not all EFLAGS status flags need emulation, as many are overwritten before use. This allows for optimization. To evaluate this, we analyzed real-world gaming workloads—spanning logic, physics, and rendering—using representative benchmarks like Jolt Physics [19], Intel OSPRay [18], and SPEC CPU, allowing us to capture EFLAGS overhead without analyzing entire games.

Our analysis of **QEMU** [7] revealed a high percentage of EFLAGS-related instructions when TCG binary translation is unoptimized, such as 25.11% in **xalancbmk**, 17.72% in **sjeng**, and up to 79.38% in synthetic benchmarks. After translation, instruction counts significantly increased, for example, a 174× rise in **async_render**, highlighting the inefficiency of direct EFLAGS emulation. To isolate this, we created controlled workloads comparing "flag-heavy" versions (relying on EFLAGS-based conditions) with "flag-minimized" versions. On native x86 systems, flag-heavy code benefited from optimized flag handling. However, on ARM—where QEMU optimizations were disabled—the flag-heavy code slowed significantly, taking up to 460.23 seconds versus 295.33 seconds for flag-minimized code, due to costly EFLAGS emulation. The corresponding assembly code is presented in Listings 1 and 2,

To overcome these issues, we present a software-only optimization strategy that reduces unnecessary EFLAGS emulation, enhancing performance without specialized hardware. Our contributions are:

- **Efficient Software-Only Mapping for x86 to ARM Flag Mapping.** We use data flow analysis to map x86 status flags to ARM's native flags when possible, reducing translation overhead while maintaining compatibility with ARM devices.
- **Resource Sharing and Taint Analysis for Syscall Emulation.** Our method enhances syscall emulation by leveraging resource sharing and taint analysis to allow ARM flags to substitute software-emulated x86 flags, improving responsiveness in flag-intensive gaming.
- **Graphics Optimization via ARM's Conditional Execution.** Utilizing ARM's efficient conditional execution, we reduce redundant computations, enhancing gameplay smoothness through optimized instruction handling without relying on hardware-specific features.

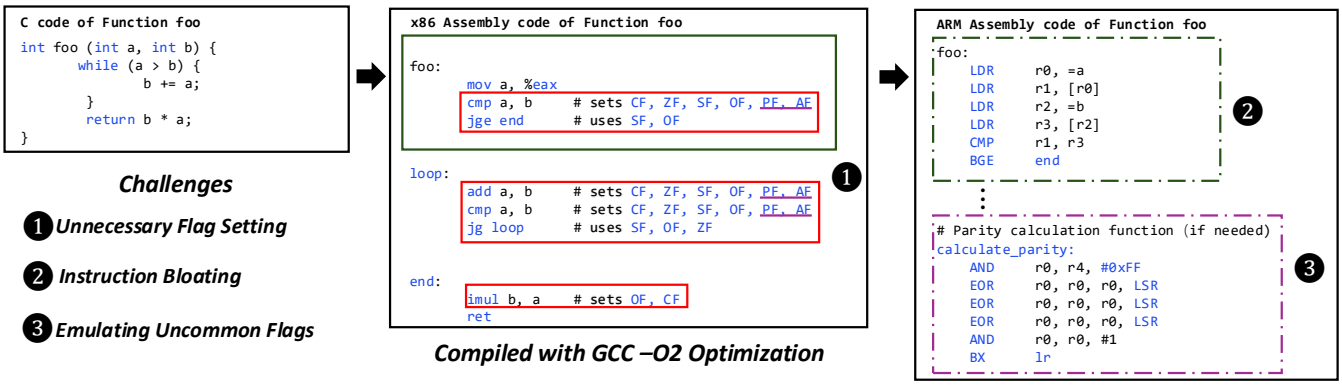


Figure 3: Foo Program Diagram. (Function foo compiled with gcc at the -O2 optimization level.)

2 OBSERVATION AND MOTIVATION

At the core of binary translation lies a fundamental challenge: bridging the gap between source and target architectures. As shown in Figure 3, the x86 assembly for function **foo** uses instructions like **cmp** to update EFLAGS, and subsequent conditional branches—like **jge**—use those updated flags to determine control flow. Such fine-grained flag handling in x86 adds complexity to translation, particularly when targeting ARM. While binary translation provides cross-platform compatibility, several issues arise when translating x86's extensive and implicitly updated flags into ARM's more limited and explicitly controlled flag set. These observations highlight key performance challenges.

2.1 Observation

Observation ❶: Unnecessary flag updates introduce overhead in binary translation. Many x86 instructions update multiple flags even if not all are needed by subsequent instructions. For example, the **cmp** instruction sets CF, ZF, SF, OF, PF, and AF, even though a following instruction may only rely on SF and PF. Simulating all these flags wastes computation. In loops, flags set by an **add** instruction might be immediately overwritten by a **cmp**, making some flag computations pointless. Minimizing these extraneous computations is crucial for performance.

> Challenge 1: How to avoid unnecessary flag setting in x86 instructions during binary translation?

Observation ❷: Divergent flag storage mechanisms cause instruction bloat. ARM and x86 handle flags differently. x86 instructions often update flags implicitly, whereas ARM requires explicit instructions to manipulate individual flags. Directly translating x86's flag logic to ARM often results in longer code sequences and instruction inflation. This expanded code footprint can degrade performance by increasing instruction cache misses and pipeline inefficiencies.

> Challenge 2: How to prevent instruction bloating when translating x86 instructions to ARM?

Observation ❸: Certain x86 flags have no direct ARM equivalents. Flags like the Parity Flag (PF) and the Auxiliary Carry Flag (AF) are unique to x86. ARM offers no direct counterparts, requiring explicit emulation and additional instructions. This increases complexity and introduces performance penalties by augmenting the computational burden needed to replicate these flag behaviors.

> Challenge 3: How to efficiently handle flags that have no ARM equivalents, such as the Parity Flag?

2.2 Implications in Cross-Platform Gaming

The overhead introduced by EFLAGS handling in binary translation significantly impacts cross-platform gaming where latency and performance are paramount. Modern games require high computational efficiency to deliver smooth, responsive experiences. When running x86-based games on ARM platforms through binary translation, the additional instructions required to simulate EFLAGS can lead to increased execution times and reduced frame rates.

Gaming workloads are latency-sensitive. Frame rates below certain thresholds cause noticeable lag and player dissatisfaction. As shown in Figure 2, EFLAGS-related instructions can constitute a substantial portion of the total instruction count in gaming workloads. The overhead from unnecessary flag simulation both inflates the translated code and consumes more processing time.

Figure 2 conceptually illustrates how excessive EFLAGS-related instructions burden the execution pipeline, increasing CPU load and latency. Resource-constrained environments—such as mobile devices or consoles reliant on ARM architectures—face even greater performance challenges due to limited processing power and stringent energy constraints.

These observations—minimizing unnecessary flag setting, avoiding instruction bloat, and efficiently handling non-native flags—are especially critical in gaming contexts. Traditional binary translation approaches that ignore EFLAGS optimizations risk rendering cross-platform gaming impractical, owing to unacceptable performance degradation [13, 35].

This underscores the need for advanced binary translation techniques that intelligently optimize EFLAGS handling. By selectively simulating only essential flags, it is possible to substantially reduce overhead and improve performance. Such innovations can narrow the performance gap, enabling gamers to enjoy high-quality experiences across diverse ARM platforms.

Algorithm 1 Flag Generation with Parity Calculation

```
Register_flags ← 0
Register_temp ← 0
for each flag in flags do
    GenerateFlag(Register_temp, flag)
    BitFieldInsert(Register_flags, Register_temp, flag)
end for
parity ← 0
for each bit in result do
    parity ← parity ⊕ bit
end for
BitFieldInsert(Register_flags, parity, parity_flag)
return Register_flags
```

3 BACKGROUND

Binary Translation for x86. Binary translation is a critical technique that enables the execution of binaries compiled for one Instruction Set Architecture (ISA) on a different ISA. When translating from x86 to ARM, one of the most significant challenges involves accurately replicating the semantics of the x86 EFLAGS register. On x86 processors, the EFLAGS register holds status flags that reflect the outcome of arithmetic and logical operations. These flags include the Sign Flag (SF), Zero Flag (ZF), Carry Flag (CF), Overflow Flag (OF), Parity Flag (PF), and Auxiliary Flag (AF), among others. Correct emulation of these flags is essential since their values influence branching and condition evaluation in x86 code. A single operation in an x86 binary can set multiple flags simultaneously, thus translators must carefully model and propagate these flags to maintain functional equivalence.

In binary translation frameworks, a dedicated and systematic approach is required to handle flag computation. One strategy involves generating the required flags after each arithmetic operation based on the result, and then storing these flags in a structure that mimics x86's EFLAGS. For instance, as illustrated in Algorithm 1, all relevant flags are computed and integrated into a single register, ensuring that conditions reliant on these flags are preserved. Special attention must be paid to the parity flag, as it has no direct counterpart in ARM, necessitating an explicit parity computation in the translator's code.

Flag Mapping in ARM: Unlike x86, the ARM architecture does not maintain a dedicated flags register analogous to EFLAGS (details of this can be seen in Section B). Instead, ARM condition codes are distributed in the program status register (PSR), which contains four primary condition flags: Negative (N), Zero (Z), Carry (C), and Overflow (V). These flags are updated conditionally, depending on the instruction and whether the instruction is suffixed with an update directive (e.g., 'S' in ARM's assembly language). This architectural difference introduces complications when translating x86 binaries that rely on implicit flag updates after nearly every arithmetic operation.

To address the lack of a direct parity or auxiliary flag in ARM, the translator must either recompute these flags as needed or adjust the code logic to avoid reliance on them. For instance, as shown in Table 1, some x86 flags can be directly mapped to ARM flags (such as SF to N, CF to C, ZF to Z, and OF to V). However, others, like PF and AF, have no direct equivalence. In these cases, binary translators often resort to explicit computations or additional bookkeeping to ensure correct semantics. Likewise, the difference in how ARM and x86 treat carry and borrow operations, summarized in Table 2, must be taken into account when implementing subtraction and comparison operations to maintain correct program behavior.

Table 1: Correspondence between x86 and ARM Flag

x86 Flag Bit	Corresponding ARM Flag Bit
Sign Flag (SF)	Negative Flag (N)
Carry Flag (CF)	Carry Flag (C)
Zero Flag (ZF)	Zero Flag (Z)
Overflow Flag (OF)	Overflow Flag (V)
Parity Flag (PF)	No Direct Correspondence
Auxiliary Flag (AF)	No Direct Correspondence

Table 2: Carry Flag Behavior in x86 versus ARM

Operation	x86 CF	ARM C
Subtraction		
No borrow ($A \geq B$)	0	1
Borrow occurs ($A < B$)	1	0
Comparison (CMP)		
$A \geq B$	0	1
$A < B$	1	0
Relationship	x86 CF = ¬(ARM C)	

In summary, modeling and translating x86 EFLAGS behavior in ARM environment is a non-trivial task. It involves careful condition code computation, explicit parity calculations, and additional logic to handle the ARM condition flags properly. Such meticulous handling of status flags is crucial for robust and reliable binary translation.

4 DESIGN & IMPLEMENTATION

To address the identified challenges, we propose an innovative binary translation system that combines dynamic recompilation with advanced analysis techniques. This system efficiently translates **x86** binaries to ARM64 architecture without relying on platform-specific hardware optimizations.

A key innovation of our system is its accurate flag emulation. Rather than emulating all flags, the system uses data flow and taint analysis to identify which CPU flags are truly needed at each execution point. This targeted approach ensures that only the necessary flags are emulated, preventing the performance penalties associated with full flag emulation. By mapping compatible flags to ARM's condition flags, we ensure correctness while minimizing overhead.

In addition to flag emulation, the system optimizes performance by focusing on translating only the instructions required for the game's execution, avoiding full-system emulation. This approach reduces unnecessary overhead, improving frame rates and reducing latency. To ensure hardware-independent optimization, the system employs software-based techniques such as speculative execution and dynamic recompilation, which are not tied to specific hardware

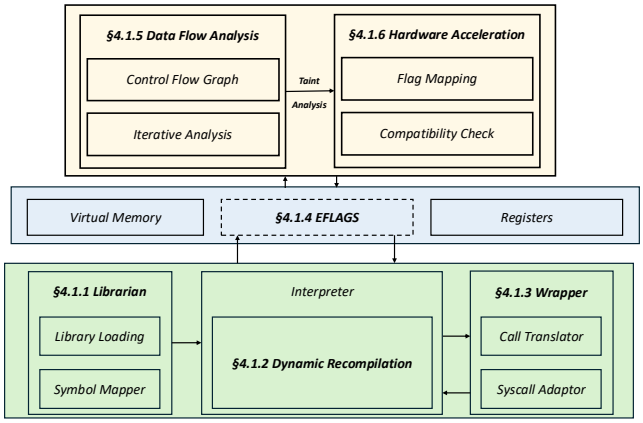


Figure 4: System Design Diagram

features. This enables high-performance, cross-platform gaming on a wide range of ARM64 devices, expanding the potential user base beyond proprietary ecosystems like Apple's.

4.1 System Overview

As illustrated in Figure 4, the architecture of the system consists of several key components, each working in concert to translate, analyze, and optimize **x86** binaries for execution on ARM64 platforms.

4.1.1 ***Library Preparation and Dependency Resolution.*** The Librarian module prepares the execution environment before any code is run. It loads all necessary external libraries, resolves symbols, and ensures that references in the **x86** binary have valid ARM64 counterparts. By creating a stable and compatible foundation, it supports subsequent stages—whether executed via interpretation or dynamic recompilation—and ensures that the final translated code can find and use all resources as intended.

4.1.2 ***High-Performance Execution with Dynamic Recompilation.*** Once the environment is established, the system begins executing the **x86** binary. Here, two complementary methods coexist: Dynamic Recompilation and Interpretation. Dynamic translates entire blocks of **x86** instructions into optimized ARM64 code on-the-fly, reducing runtime overhead and improving overall performance. For complex or infrequently encountered instructions that resist efficient translation, the system gracefully falls back to Interpretation, which executes them instruction-by-instruction.

4.1.3 ***Seamless Cross-Architecture Functionality via Wrappers.*** While execution proceeds, the Wrapper module continuously manages interactions with the host environment. It adapts function calls, arguments, and return values from **x86** conventions to ARM64. Likewise, it translates system calls and resource accesses, ensuring seamless interplay between the translated code and the underlying platform. By doing so in real-time, Wrappers maintain stability and correctness, allowing the translated program to behave as if it were running natively.

4.1.4 ***EFLAGS Optimization for Efficient Conditional Handling.*** Once dynamic recompilation has begun to accelerate code execution, the focus shifts to optimizing critical architectural details. Chief among these is the handling of **x86** EFLAGS, which encode conditions for branches and other operations. At this stage, EFLAGS are efficiently managed to minimize overhead. This involves mapping "clean" flags that directly correspond to ARM's native N, Z, C, and V flags, so no extra instructions are needed for their evaluation. Flags that lack direct ARM equivalents are identified as "dirty" and emulated using a minimal set of instructions to preserve performance. By refining EFLAGS handling here, the system reduces unnecessary computations and creates a more efficient execution pipeline.

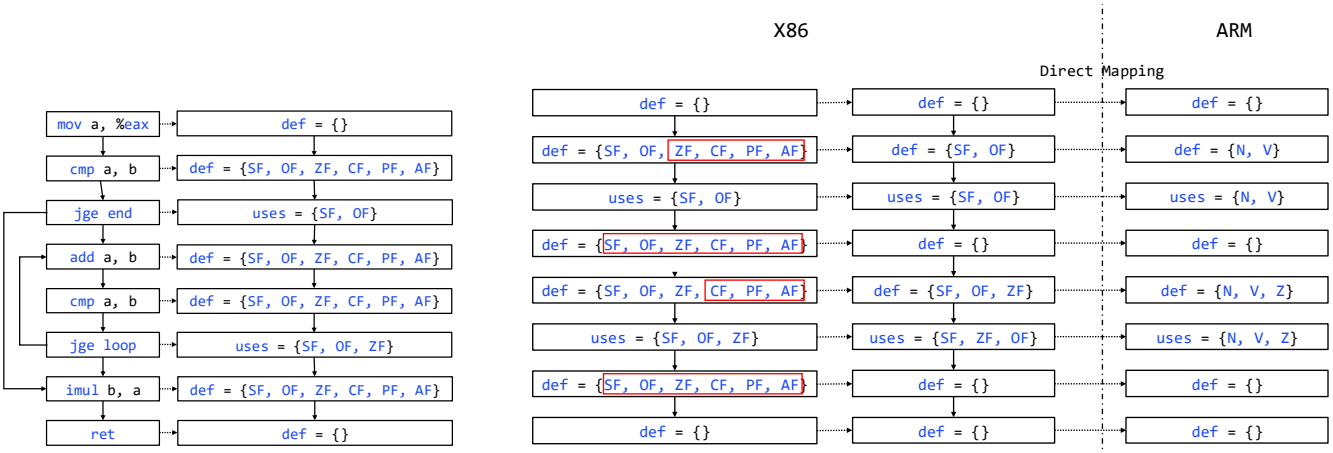
4.1.5 ***Advanced Data and Control Flow Analysis.*** With EFLAGS management streamlined, the system proceeds to a deeper level of analysis. Data Flow Analysis (DFA) examines how data moves through instructions and constructs a Control Flow Graph (CFG). It then applies iterative methods to understand how each instruction influences EFLAGS. Taint Analysis marks each EFLAGS bit as Clean, Dirty, or Unknown, guiding further optimization. Clean flags are already fully optimized, Dirty flags are known to require limited emulation, and Unknown flags undergo additional analysis until their status is clarified. This refined understanding of how conditions flow through the codebase allows for more aggressive optimization strategies.

4.1.6 ***Hardware-Assisted Optimization for Conditional Execution.*** Armed with insights from Data Flow and Taint Analysis, the system taps into hardware capabilities to further accelerate execution. This involves establishing a direct correspondence between **x86** condition checks and ARM's native conditional instructions. By aligning **x86** semantics with ARM's hardware-level features through careful Flag Mapping and Compatibility Checks, the translated code can take full advantage of ARM64's efficient conditional execution. This synergy ensures that the final generated code runs with minimal overhead, leveraging the native instruction set to handle branching and decision-making swiftly.

4.2 EFLAGS-Aware Data Flow Analysis

Our approach advances beyond the methodologies presented by Salgado et al. [27] and Ottoni et al. (Harmonia) [24]. Unlike Salgado's reliance on hardware triggers and Harmonia's optimization limited to predefined code regions, our technique employs an interprocedural, per-flag liveness analysis integrated into the control-flow graph, enabling the elimination of dead flags even mid-block. Additionally, we utilize dynamic taint tracking to identify and skip updates of untainted flags, surpassing Harmonia's static data-flow analysis and significantly reducing unnecessary flag emulation.

Our method leverages runtime taint information to selectively emulate only essential flags dynamically, improving efficiency beyond static approaches. Unlike Harmonia, which statically applies optimizations, we dynamically adjust based on actual runtime conditions. Moreover, we leverage native ARM hardware capabilities for just-in-time emulation of critical flags, resulting in reduced overhead and increased performance, especially beneficial in scenarios where precompiled static optimizations are insufficient or unavailable. Though data flow and conditional execution are standard compiler techniques, our contribution is applying them at



(a) Initial Data Flow Graph State **(b) Final Data Flow Graph State**

Figure 5: Comparison of Initial and Final Data Flow Graph States

Algorithm 2 Efficient Flag Setting Based on Analysis

```
for flag in flags_active do
    GenerateFlag(Register_temp, flag)
    BitFieldInsert(Register_flags, Register_temp, flag)
end for
```

Algorithm 3 Data Flow Analysis Iteration Method

Require: Data flow graph G, initial state $Init$, termination nodes $TerminateG$

```
function ANALYZE(G, Init)
    for t in TerminateG do
        ITERATE(t, Init)
    end for
end function
```

Require: Node v, current state in, flag sets def_b, use_b, result out_b, predecessors $pred_b$

```
function ITERATE(v, in)
    out ← (in − def_b) ∪ use_b ∪ out_b
    if out ≠ out_b then
        out_b ← out
        for pred in pred_b do
            ITERATE(pred, out)
        end for
    end if
end function
```

runtime in an x86-to-ARM translator for scenarios like gaming, where offline compilers do not dynamically compile EFLAGS to ARM condition flags. We selectively emulate only essential flags, leveraging ARM hardware in a just-in-time setting.

Specifically, our EFLAGS-aware data flow analysis identifies the minimal set of required flags by performing backward, interprocedural analysis across the entire control flow graph, whereas Salgado and Harmonia rely on conservative or limited-region analyses. Furthermore, our direct mapping strategy to ARM's status registers exploits native hardware support extensively, improving performance by directly aligning x86 condition flags to ARM's hardware-managed indicators, unlike previous methods that depend heavily on software-based emulation.

This optimization employs a data flow analysis framework to accurately determine when and where each EFLAG is needed. By performing a backward analysis of the control flow, the method identifies the specific status flags that each instruction truly depends on, eliminating redundant computations. As illustrated in Figure 5a, an initially broad network of flag dependencies is progressively refined to a minimal set of essential flags that impact subsequent operations. Shifting from a comprehensive to a selective approach significantly improves EFLAGS management efficiency.

At the core of this method is the propagation of flag usage from their points of use back to their setting points. Conditional operations and other flag-dependent instructions guide this propagation, ensuring that only flags influencing future decisions remain active. During the analysis of each control flow node, its role in defining or using specific flags is combined with information from related nodes. Through iterative refinement, unnecessary flag tracking is eliminated, resulting in a stable and concise set of critical flags at each relevant program point.

This streamlined flag dependency enhances the translation process by allowing the generation to focus solely on required status indicators instead of updating all flags after every arithmetic instruction. For instance, if only the Zero and Sign flags affect subsequent decisions, maintaining the Carry or Overflow flags becomes unnecessary. Consequently, the translated output is more efficient, particularly in performance-critical sections, reducing overhead and improving responsiveness.

An additional improvement involves tracking the necessity of flags, distinguishing between redundant and required flags at specific execution points. By aligning these states with comprehensive data flow results, instructions are introduced only when meaningful. As macro-level constructs incorporate these insights, unnecessary flag operations are avoided, ensuring the final outcome maintains architectural fidelity while optimizing operational efficiency.

Once the data flow equations converge on a minimal set of essential flags, these insights are integrated with ARM's architectural features. Without selective equations, translations might simulate all x86 flags on ARM, including unused ones. However, data flow analysis removes such inefficiencies. The revised instructions, guided by Algorithm 2 and Algorithm 1, interact directly with ARM's native status registers, leveraging hardware support for common flags like Zero, Sign, Carry, and Overflow.

Furthermore, these equations not only reduce the flag set but also optimize specific calculations, such as the parity condition. Instead of computing parity after every instruction, the equations determine it only when necessary by treating computation results as bit sets and reducing them to a single parity value through relevant bit iterations. This incremental approach ensures that the parity flag is calculated based on prior analysis rather than as a routine operation.

This transition from broad flag-generation equations to a selective, context-aware set represents a significant improvement. By reusing results from earlier data flow equations that identify needed flags downstream, the new equations avoid unnecessary computations. They no longer generate every possible flag state mechanically but instead integrate results intelligently, inserting only the minimal and precisely selected flags. Consequently, the final equations provide a more efficient and semantically accurate representation of the program's state, ensuring operations like parity calculation occur only when necessary, thereby enhancing both clarity and performance of the translation process.

4.3 Direct Mapping to ARM's Status Registers

Focusing on essential x86 flags enables a direct mapping to ARM's status registers, reducing the overhead typically associated with flag emulation. As depicted in Figure 5b, ARM's native status registers correspond directly to specific x86 flags. The x86 Sign, Zero, Carry, and Overflow flags align naturally with ARM's built-in condition indicators used in comparisons and arithmetic operations. This alignment allows ARM's hardware to automatically update and access these conditions, eliminating the need for separate software-based flag computations and storage. Instead, our approach leverages the target environment's hardware support to manage flag operations efficiently.

To achieve this, we utilize a data flow analysis technique to identify and isolate only the flags genuinely required by the x86 code. This selective approach minimizes complexity and overhead by ignoring unnecessary flags. By directly mapping these essential flags to ARM's status registers, such as the APSR (Application Program Status Register), the translator effectively uses ARM's native flag-handling mechanisms, which automatically update condition flags based on instruction results.

This direct mapping significantly optimizes performance. ARM instructions inherently maintain certain flags, allowing the translator to rely on the processor's built-in status updates for arithmetic or logical instructions. This approach removes the need for manual flag calculations and adjustments after each instruction, thereby reducing both latency and code size. For example, when determining if one value is larger or equal to another, the translator enables an ARM instruction to set the processor's internal flags instead

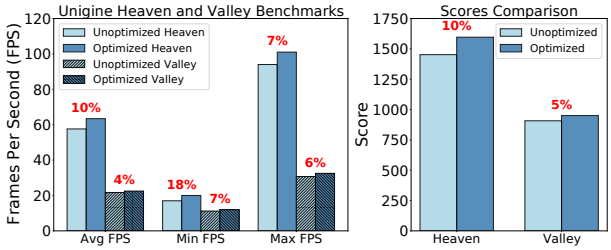


Figure 6: Performance Metrics for Unigine Heaven 4.0 and Unigine Valley 1.0 Benchmarks

of explicitly computing carry and borrow conditions in software. Subsequent comparisons or branches then operate directly on these hardware-managed conditions, minimizing the number of required instructions and simplifying data flow.

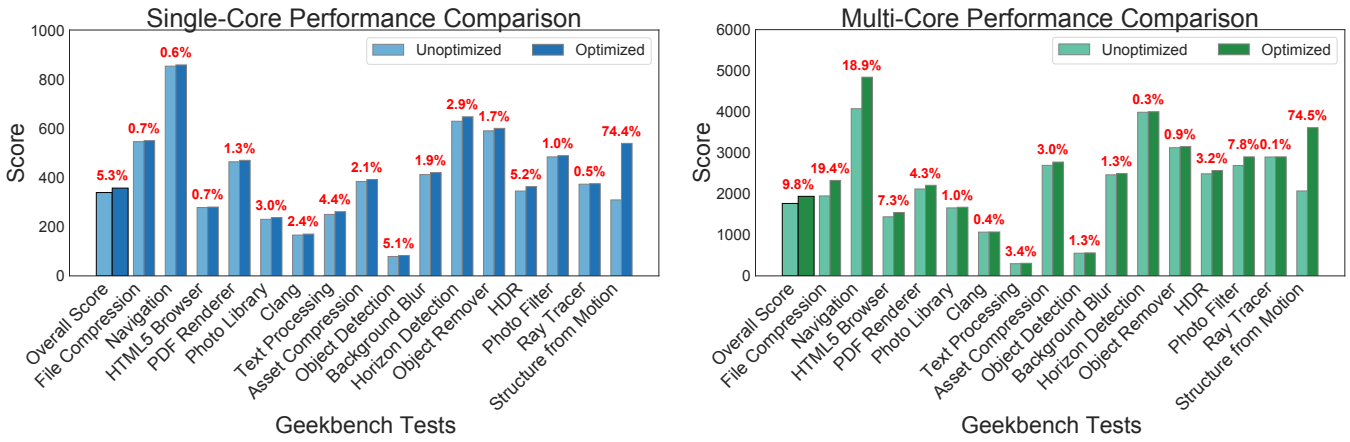
To further enhance optimization, the translator employs instruction scheduling techniques that maximize the use of ARM's conditional execution features. By arranging instructions to take advantage of ARM's conditional branches and predicated instructions, the translation process reduces the number of branching operations and improves instruction-level parallelism. This results in more efficient utilization of the ARM pipeline and overall better performance.

However, not all flags translate directly between architectures, necessitating careful adjustments in specific scenarios. For example, the carry flag behaves differently during subtraction: in x86, it indicates a borrow, whereas in ARM, it signifies the absence of a borrow. Therefore, the translator must reinterpret or invert the carry flag's meaning to accurately map x86 code that relies on it to ARM's flag semantics. This reinterpretation is essential for maintaining functional equivalence in code paths dependent on the exact behavior of the x86 carry flag. To address such discrepancies, the translator incorporates a flag translation layer that dynamically adjusts flag interpretations based on the operation context, ensuring accurate and reliable flag behavior across different instruction types.

Adopting this direct mapping strategy ensures that the translation process remains faithful to the original x86 code's semantics while tightly integrating with ARM's native hardware features. Consequently, the final environment not only preserves the required x86 flag behavior but also achieves enhanced performance and efficiency by leveraging ARM's internal status management capabilities. This integration ensures that translated applications maintain their intended behavior and performance characteristics, providing a robust and efficient execution environment on ARM architectures.

5 EVALUATION

We have integrated our binary translation system into Box64 by restructuring its condition code handling mechanism to directly map x86 EFLAGS onto ARM conditional flags during dynamic recompilation. Our approach adheres closely to established methodologies, carefully optimizing the translation to minimize overhead for the evaluated applications. Specifically, we enhanced code generation macros to unify and streamline the translation of x86 conditional



(a) Geekbench 6 Single-Core Performance

(b) Geekbench 6 Multi-Core Performance

Figure 7: Geekbench 6 Computational Performance Improvements

jumps, directly leveraging ARM64's inherent condition codes. Additionally, we implemented flag taint tracking to enhance speculative execution safety and maintain consistent state information across conditional execution paths. Further refinements involved precise management of barriers, register allocation strategies, and scratch register usage, effectively controlling the conditions under which flags are accessed and modified. Instruction bloat was minimized by selectively mapping only essential x86 flags to native ARM condition codes, explicitly excluding unused flags to reduce computational overhead. Consequently, our integrated solution substantially decreases dependency on software-based condition code emulation, resulting in improved overall system performance.

5.1 Experimental Set-Up

Testbed. All experiments were conducted on an NVIDIA Jetson Xavier AGX platform, equipped with an NVIDIA GeForce GTX 1060 GPU (6 GB HBM, 6144 MiB), an ARMv8 processor comprising four sockets, each with two cores, totaling eight cores, and 30 GiB of host memory. The system operates Ubuntu 20.04.6 LTS (Focal Fossa), representing an ARMv8.2 architecture typical of mobile system-on-chip (SoC) configurations commonly employed in smartphones, tablets, and IoT devices. This ensures that performance improvements observed in our study are broadly applicable within ARM-based ecosystems [32].

Workloads and Methodology. To ensure rigorous quantitative analysis, standardized aggregated metrics from multiple benchmarks were utilized. We evaluated our system performance using three distinct workload categories, each targeting specific platform attributes. First, we employed Geekbench, a synthetic benchmarking suite encompassing a variety of CPU and GPU tasks. Second, the Heaven Benchmark was utilized to evaluate graphics-intensive capabilities comprehensively. Lastly, we incorporated benchmarks derived from real-world Steam games to assess comprehensive end-to-end performance under practical usage conditions. Multiple experimental iterations were conducted to verify result consistency and thoroughly examine the performance implications of binary translation.

Table 3: Overhead Analysis (CPU and GPU Benchmarks)

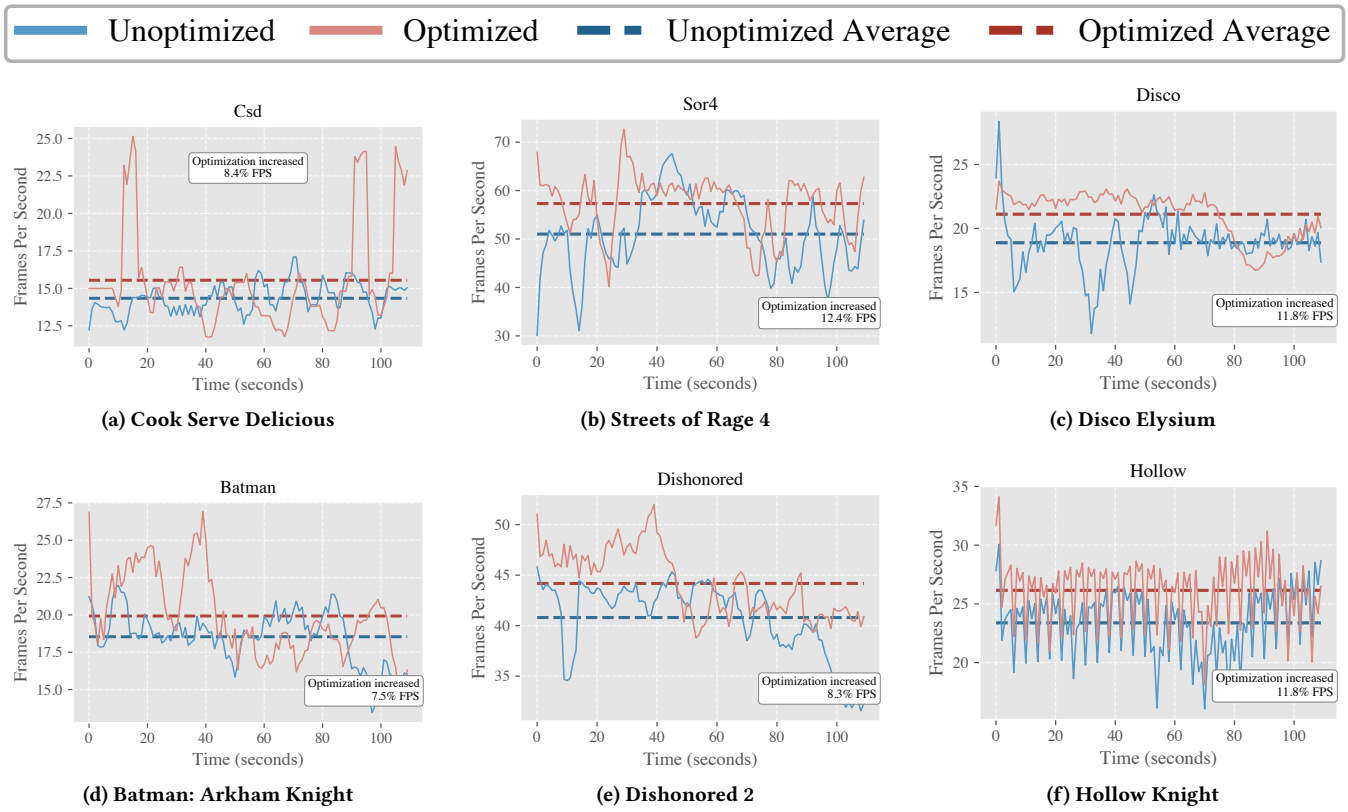
Metric	Geekbench	Heaven	Valley
CPU overhead (%)	0.12%	0.08%	0.24
Memory overhead (%)	0.84%	4.01%	3.17%

5.2 System Computational Performance

Experiment Set-up. We evaluate the system's computational performance using Geekbench 6, focusing on single-core and multi-core scenarios across a range of workloads such as text processing, compression, complex navigation, and computational photography. Experiments are conducted under both unoptimized and optimized configurations, allowing us to isolate the impact of improved memory handling, GPU acceleration, and our integrated binary translation optimizations. The latter includes re-architecting condition code handling within Box64's dynamic recompilation framework. By carefully mapping x86 **EFLAGS** onto ARM condition flags, we reduce software-based condition emulation overhead, streamline branching, and increase the efficiency of just-in-time (JIT) code generation. These modifications, combined with taint tracking of flags and enhanced register allocation strategies, ensure that condition-dependent execution paths are more directly and efficiently expressed on the ARM architecture.

The results of our benchmarking are presented in Figure 7, showing single-core and multi-core performance improvements across a variety of tasks. Enhanced flag translation, improved speculation safety, and more precise management of barriers all play roles in strengthening overall throughput.

Overall Performance. As illustrated in Figure 7(a), our optimized configuration raises single-core scores from 339 to 357, reflecting meaningful gains. This uplift can be traced to lower translation overhead for conditional instructions and more efficient memory pipelines, enabling smoother execution of tasks like PDF rendering, text processing, and photo filtering. The streamlined condition code handling effectively eliminates unnecessary software guardrails, reducing latency on critical loops and conditional jumps.



(a) Cook Serve Delicious **(b) Streets of Rage 4** **(c) Disco Elysium**

(d) Batman: Arkham Knight **(e) Dishonored 2** **(f) Hollow Knight**

Figure 8: FPS Performance Comparison Across Selected Games

Likewise, multi-core performance, shown in Figure 7(b), rises from 1765 to 1938. Here, the combination of improved concurrency management, enhanced memory allocation strategies, and refined binary translation heuristics synergizes to boost parallel workloads such as object detection and asset compression. Fine-grained scratch register usage and better scheduling further diminish contention among threads, thus capitalizing on ARM's parallel execution capabilities.

Breakdown. A detailed view in Figure 7(a) reveals that refinements in memory handling predominantly benefit data-intensive tasks, ensuring faster data retrieval and smoother memory flows. Similarly, GPU acceleration amplifies performance gains in highly parallelizable workloads, such as ray tracing or image processing, accelerating their execution in the multi-core scenarios of Figure 7(b). The careful integration of binary translation optimizations reduces overhead in handling condition flags and complex control-flow patterns, directly improving code generation quality and execution speed. These insights inform targeted future enhancements, such as more intelligent memory allocators, adaptive scheduling algorithms, and sophisticated JIT compilers that can further leverage platform-specific architectural features.

Overhead and Scalability. Table 3 highlights the efficiency of our framework for **Geekbench**, a CPU-intensive workload, where CPU overhead remains negligible at **0.12%** and memory overhead is tightly constrained to **0.84%**. This demonstrates that our JIT translation layer maintains near-native resource utilization even under dynamic code generation demands. By optimizing register allocation, reducing redundant condition flag updates, and isolating taint tracking to critical code regions, the system avoids scalability bottlenecks for purely computational tasks. These design choices ensure that performance gains from speculative execution and parallelization are preserved, making the approach viable for latency-sensitive and high-throughput CPU workloads.

5.3 Graphics Rendering Performance

Experiment Set-up. To assess graphics performance, we employ the Unigine Heaven 4.0 and Unigine Valley 1.0 benchmarks, which subject the system to demanding 3D rendering tasks involving complex geometry, dynamic lighting, and high-resolution textures. We compare baseline and optimized configurations to quantify the improvements due to advanced GPU instruction translation, streamlined texture loading, and more efficient shading pipelines. As with the CPU-bound tests, our binary translation layer benefits GPU-driven workloads by efficiently mapping high-level rendering calls to ARM-friendly instruction sequences and ensuring that conditional logic within shaders and GPU kernels executes more fluidly.

Overall Performance. The optimized configuration leads to consistently better FPS across both benchmarks, as shown in Figure 6. For Unigine Heaven 4.0, the average FPS jumps from 57.6 to 63.4 (+10% increase), and the minimum FPS improves from 17.0 to 20.0,

reducing stutters and improving responsiveness during scene transitions. Maximum FPS also rises, indicating more efficient handling of peak workloads and memory transfers. In Unigine Valley 1.0, average FPS grows from 21.7 to 22.5, enhancing fluidity in particularly complex environments. Corresponding benchmark scores climb from 1452 to 1596 in Heaven and from 907 to 950 in Valley, reflecting these systemic enhancements.

Breakdown. Closer inspection reveals that improved GPU-based memory optimizations ensure more stable frame times, leading to higher minimum FPS and smoother rendering during the most challenging frames. Our binary translation improvements also help by selectively inlining or simplifying control-flow constructs within GPU code, better aligning shader logic with ARM execution patterns. This synergy decreases translation latency and reduces branching overhead in the GPU pipeline, enabling both higher maximum FPS and steadier average frame rates. Together, these findings guide future optimizations focused on further refining memory access patterns, adjusting register allocation, and integrating more adaptive shading algorithms to sustain high performance across a broad range of graphical workloads.

Overhead and Scalability. For GPU-centric benchmarks like **Heaven** and **Valley**, Table 3 reveals CPU overheads of **0.08%** and **0.24%**, respectively, with memory overheads of **4.01%** and **3.17%**—consistent with the expanded state management required for rendering pipelines. The framework adapts to GPU workloads by prioritizing translation of frequently executed shader blocks and streamlining synchronization between CPU-side translation threads and GPU command queues. This ensures that memory bandwidth remains available for texture/geometry data, while runtime translation latency stays decoupled from frame rendering deadlines. The results validate that our system scales to accommodate GPU workloads without disrupting real-time rendering performance.

5.4 Real-World Gaming Performance

Experiment Set-up. We next measure real-world responsiveness and visual smoothness by running a curated selection of Steam games that represent diverse genres and rendering engines. We track FPS and frame-time consistency to determine whether our integrated optimizations, including improved binary translation and memory handling, translate into tangible benefits for end-users.

Overall Performance. Across various titles, our optimizations consistently elevate average FPS, delivering a more fluid experience during high-intensity action sequences. Minimum FPS also rises, alleviating micro-stutters and reducing frame-time spikes when scenes become visually dense or involve rapid camera movement. The maximum FPS improvements indicate that the system can handle short bursts of demanding effects more gracefully. As a result, gameplay feels both more responsive and visually coherent, contributing to a noticeably smoother player experience.

Breakdown. Memory-specific enhancements underpin these improvements, ensuring swift asset streaming and seamless memory access that maintain stable frame delivery. The binary translation refinements—especially the direct mapping of x86 condition flags to ARM condition codes and the careful tuning of JIT heuristics—improve peak performance by allowing the runtime system to adapt quickly to complex in-game scenarios. By reducing overhead from condition code emulation, the engine maintains higher sustained throughput, capitalizing on parallel hardware resources more effectively. In sum, these insights point toward future directions such as further refining CPU-GPU synchronization, exploring more advanced speculation mechanisms, and integrating adaptive shading strategies to continue advancing gameplay performance and fluidity.

Overhead and Scalability. While the GPU often represents the primary bottleneck in most AAA games due to heavy rendering demands, CPU-side optimizations remain essential for managing large or complex codebases effectively. By minimizing CPU overhead—illustrated in Table 3—our approach helps streamline the execution of critical game logic, scripting, and physics calculations. This balanced design ensures that, even under graphics-intensive workloads, the CPU can efficiently handle task scheduling, resource management, and code translation without impeding overall gameplay performance. Consequently, the system can better accommodate large-scale projects where efficient CPU usage and scalability across multiple cores are vital for stable frame rates and smooth user experiences.

6 DISCUSSION

Our evaluation demonstrates the effectiveness of the proposed software-only method in enhancing binary translation performance between x86 and ARM architectures, especially in gaming applications. While our study emphasizes gaming scenarios due to their frequent control-flow changes and rapid EFLAGS updates, our techniques have broader applicability in other domains characterized by similarly dynamic control-flow behavior, such as real-time systems or interactive applications requiring rapid state updates. We observed substantial performance improvements in computational benchmarks, graphical rendering, and real-world gaming scenarios. These gains result from efficient EFLAGS management, targeted flag emulation, and the direct mapping of compatible x86 flags to ARM's native flags. In interactive workloads, even modest performance gains can significantly reduce frame-time variability in the critical 30–60 FPS range, mitigating stutters and enhancing user satisfaction. Although comprehensive user studies are beyond this paper's scope, maintaining a stable frame rate is a key factor in ensuring smooth real-time experiences. Notably, frame rates exceeding 60 FPS often yield diminishing perceptible returns, yet preserving consistent frame delivery remains beneficial for overall responsiveness and user comfort.

A key advantage of this approach is its ability to bridge architectural differences without relying on proprietary hardware, ensuring broad applicability across various ARM-based platforms and promoting widespread adoption. In contrast to Apple's Rosetta 2, which leverages Apple-specific hardware instructions for significant speedups on Apple Silicon, our translator is entirely software-based. Rosetta 2's heavy reliance on proprietary hardware features limits its portability to other ARM system-on-chips (SoCs). By employing data flow–driven optimizations and ARM's built-in conditional execution rather than Apple-specific instructions, our approach offers broader compatibility across ARMv8-compliant processors, including Snapdragon and Exynos. This vendor-neutral

design ensures that optimizations remain accessible to a wide range of devices without sacrificing performance benefits derived from selective flag mapping. Additionally, by focusing on data flow analysis and taint tracking, the method reduces redundant computations, optimizing both translation latency and execution efficiency. This not only boosts performance but also lowers energy consumption, aligning with ARM's focus on power efficiency.

However, some limitations need further investigation. While FPS and computational scores showed significant improvements, certain benchmarks only saw modest gains, indicating areas for potential optimization. Addressing complex control-flow patterns and refining GPU-specific translations could yield additional performance benefits. Additionally, despite some gaming workloads being predominantly GPU-bound, our optimizations on the CPU side remain important, particularly in managing extensive or intricate codebases that influence overall system performance. Furthermore, the inversion of carry flags and explicit parity calculations introduce minor overheads that might be reduced through advanced speculative execution techniques.

Future research could explore hybrid approaches that integrate machine learning to dynamically predict and optimize flag dependencies. Additionally, deeper integration with ARM hardware features, such as pointer authentication or dedicated registers, could complement our translator to achieve greater optimization. However, our design ethos emphasizes the value of maintaining a vendor-neutral, purely software-based approach to ensure compatibility across diverse ARM implementations without proprietary dependencies. Leveraging hardware-specific features where available, without sacrificing generalizability, could extend the system's benefits to other areas like AI and machine learning applications, which are increasingly relevant on ARM platforms.

7 RELATED WORKS

Optimizing the EFLAGS register is crucial for effective binary translation across architectures. Research in this domain falls into software-only, hardware-only, and hybrid approaches. The central challenge lies in bridging the semantic gap between the source architecture's condition code mechanism and the target architecture's potentially simpler or differently structured condition handling.

7.1 Software-Only Approaches

Software-only methods employ compiler techniques and runtime analysis without modifying hardware. Bansal and Aiken [6] introduced a peephole superoptimization technique to optimize small instruction sequences, managing EFLAGS efficiently and eliminating redundant instructions to boost performance. Salgado et al. [27] used dataflow analysis to reduce the overhead of condition code emulation in dynamic binary translation. Chen et al. [9] focused on dynamic analysis within a whole-system emulator to lessen EFLAGS management overhead. Wang et al. [30] developed a pattern translation method for necessary flag computations, selecting appropriate instruction groups based on flag pattern semantics to minimize native code generation and enhance performance. Xie et al. [34] proposed a peephole optimization approach using live variable analysis and instruction fusion through pattern matching, significantly improving performance.

7.2 Hardware-Only Approaches

Hardware-only solutions improve EFLAGS handling by incorporating architectural enhancements, reducing the need for software emulation. Li et al. [21] presented a hardware-assisted method that offloads specific tasks to dedicated components, enhancing EFLAGS efficiency. Hu et al. [16] proposed a cost-effective hardware-assisted translation system for efficient EFLAGS management. The IA-32 Execution Layer [12] explored architectural features to translate IA-32 instructions, including EFLAGS, to Itanium-based systems.

7.3 Combined Software and Hardware Approaches

Hybrid approaches combine software and hardware strategies to optimize EFLAGS in binary translation. The Loongson binary translation system [28] integrates software and hardware innovations for efficient condition code handling across architectures. Hu et al. [16] extended their hardware-assisted system with co-designed software components to improve EFLAGS optimization. Li et al. [21] combined hardware support with software strategies for effective condition bit mapping. Salgado et al. [27] advocated a hybrid approach by evaluating both software and hardware techniques for condition code emulation. Similarly, Chen et al. [9] merged software optimizations with hardware features to comprehensively manage EFLAGS in binary translation.

Ottoni et al. [24] introduced Harmonia, an ARM-to-IA dynamic binary translator integrating software and hardware optimizations. Harmonia identifies two key challenges for effective binary translation: register mapping and condition-code handling. They propose optimizations including region-based register mapping and redundant-compare elimination to minimize memory accesses and condition-code emulation overheads, complemented by targeted hardware ISA extensions for IA architecture to further reduce translation overhead.

8 CONCLUSION

This paper introduced a software-only strategy to bridge the x86-to-ARM gap for gaming applications, focusing on efficient EFLAGS emulation within binary translation. Our method leverages data flow analysis and taint tracking to isolate necessary flag computations and directly maps compatible flags to ARM's native hardware, thereby avoiding substantial emulation overhead without requiring proprietary hardware. The effectiveness of this technique was confirmed through rigorous evaluation: computational benchmarks showed performance gains reaching up to 18%, while graphics benchmarks saw similar FPS improvements of up to 18%. Most importantly, this translated to a smoother end-user experience, with real-world Steam games exhibiting FPS increases between 7% and 12%. These results demonstrate a clear path towards enhanced cross-platform gameplay and broader ARM adoption in the PC gaming space.

ACKNOWLEDGMENTS

We thank our shepherd and reviewers for their valuable feedback. This work was supported in part by the National Natural Science Foundation of China (No. 62141218, 62232012), and the Shanghai Key Laboratory of Scalable Computing and Systems.

A EXAMPLE PROGRAMS COMPARING EFLAGS USAGE

The following are two assembly programs that perform the same task of counting set bits in an array of 64-bit integers. The only difference is that one program utilizes the **EFLAGS** register, while the other avoids using it.

Listing 1: Bit Counting Using EFLAGS

```
SECTION .data
    array_size equ 100000000
    array times array_size dq 0xFFFFFFFFFFFFFFFF
    total_bits dq 0
    newline db 10
SECTION .bss
SECTION .text
    global _start
_start:
    mov     rcx, array_size
    lea     rsi, [array]
    xor     rax, rax
    xor     rbx, rbx
count_bits_loop:
    mov     rdx, [rsi]
    mov     r8, 64
bit_loop:
    shr     rdx, 1
    jc      bit_is_set
    jmp     check_next_bit
bit_is_set:
    inc     rbx
check_next_bit:
    dec     r8
    jnz     bit_loop
    add     rsi, 8
    dec     rcx
    jnz     count_bits_loop
    mov     rax, rbx
    call    uint_to_ascii
    mov     rdx, rsi
    mov     rsi, rbx
    mov     rax, 1
    mov     rdi, 1
    syscall
    mov     rax, 60
    xor     rdi, rdi
    syscall
uint_to_ascii:
    ret
```

The primary distinction between Listing 1 and Listing 2 lies in how they detect and count set bits. Listing 1 explicitly relies on **EFLAGS** operations: after shifting a bit out of `rdx` via `shr`, it checks the **Carry Flag (CF)** using `jc bit_is_set` to increment the counter, introducing a branch dependency on EFLAGS. In contrast, Listing 2 avoids EFLAGS entirely by isolating the least significant bit with `and rax, 1`, directly adding its value to `rbx` without branching. Loop control is handled by the `loop` instruction, which decrements `rcx` and jumps without referencing EFLAGS, thereby reducing branch mispredictions and pipeline stalls to improve performance on modern CPUs.

Listing 2: Bit Counting Without Using EFLAGS

```
SECTION .data
    array_size equ 100000000
    array times array_size dq 0xFFFFFFFFFFFFFFFF
    newline db 10
SECTION .text
    global _start
_start:
    mov     rcx, array_size
    lea     rsi, [array]
    xor     rbx, rbx
count_bits_loop:
    push    rcx
    mov     rcx, 64
    mov     rdx, [rsi]
bit_loop:
    mov     rax, rdx
    and     rax, 1
    add     rbx, rax
    shr     rdx, 1
    loop    bit_loop

    pop     rcx
    add     rsi, 8
    loop    count_bits_loop

    mov     rax, 60
    xor     rdi, rdi
    syscall
```

B COMPARISON OF X86 AND ARM PROCESSOR STATUS FLAGS

Table 4: Detailed Description of Flag Bits in the x86

Flag Bit	Description
CF (Carry Flag)	Indicates unsigned overflow in arithmetic operations.
ZF (Zero Flag)	Set if the arithmetic result is zero.
SF (Sign Flag)	Reflects the sign of the result's most significant bit.
OF (Overflow Flag)	Set if signed overflow occurs in arithmetic.
PF (Parity Flag)	Set if the least significant byte has even parity.
AF (Auxiliary Carry Flag)	Set if a carry/borrow occurs between nibble bits.

Table 5: Detailed Description of Condition Flags in ARM

Flag Bit	Description
N (Negative Flag)	Set if the arithmetic result is negative (most significant bit set).
Z (Zero Flag)	Set if the arithmetic result is zero.
C (Carry Flag)	Indicates unsigned overflow or borrow in arithmetic operations.
V (Overflow Flag)	Indicates signed overflow occurred during arithmetic operations.

REFERENCES

[1] Alibaba Cloud. [n. d.]. *Alibaba Cloud Unveils New Server Chips to Optimize Cloud Computing Services.* https://www.alibabacloud.com/blog/alibaba-cloud-unveils-new-server-chips-to-optimize-cloud-computing-services_598159 Alibaba Cloud blog.

[2] Bhojan Anand, Karthik Thirugnanam, Jeena Sebastian, Pravein G. Kannan, Akhihebbal L. Ananda, Mun Choon Chan, and Rajesh Krishna Balan. 2011. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (Bethesda, Maryland, USA) *(MobiSys '11)*. Association for Computing Machinery, New York, NY, USA, 57–70. https://doi.org/10.1145/1999995.2000002

[3] Apple Developer. 2020. *About the Rosetta Translation Environment.* https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment Accessed via Apple Developer Documentation.

[4] Apple Inc. 2020. *Apple Announces Mac Transition to Apple Silicon.* https://www.apple.com/newsroom/2020/06/apple-announces-mac-transition-to-apple-silicon/ Apple Newsroom announcement.

[5] ARM Ltd. 2023. *ARM CPUs Powering the Smartphone Market.* https://www.arm.com/markets/consumer-technologies/smartphones Accessed from ARM's Consumer Technologies section.

[6] S. Bansal and A. Aiken. 2008. Binary Translation Using Peephole Superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 177–192.

[7] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[8] Beth Kindig. 2024. *Arm Stock: AI Chip Favorite Is Overpriced.* https://www.forbes.com/sites/bethkindig/2024/03/21/arm-stock-ai-chip-favorite-is-overpriced/ Forbes.

[9] Yu Chen and Others. 2006. Dynamic binary translation and optimization in a whole-system emulator. In *Proceedings of the International Conference on Parallel Processing Workshops*. 73–80.

[10] Eduardo Cuervo, Alec Wolman, Landon P. Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (Florence, Italy) *(MobiSys '15)*. Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/2742647.2742657

[11] Fujitsu. 2023. *Specifications of Fugaku.* https://www.fujitsu.com/global/about/innovation/fugaku/specifications/ Fugaku specifications.

[12] D. Goldenberg, M. Ben-Yehuda, and A. Shulman. 2003. IA-32 Execution Layer: A Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 191–201.

[13] Yicheng Gu, Yun Wang, Yunfan Sun, Yuxin Xiang, Yufan Jiang, Xuyan Hu, Zhengwei Qi, and Haibing Guan. 2024. gVulkan: scalable GPU pooling for pixel-grained rendering in ray tracing. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC'24)*. USENIX Association, USA, Article 70, 15 pages.

[14] Christopher Harper. 2024. *Expanded Steam Gaming Compatibility Likely Coming to Arm Chips with Hundreds of Windows Games — Valve Testing ARM64 Proton Compatibility Layer.* https://www.tomshardware.com/video-games/pc-gaming/steam-likely-coming-to-arm-chips-with-support-for-hundreds-of-windows-games-valve-testing-arm64-proton-compatibility-layer Tom's Hardware.

[15] Songtao He, Yunxin Liu, and Hucheng Zhou. 2015. Optimizing Smartphone Power Consumption through Dynamic Resolution Scaling. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (Paris, France) *(MobiCom '15)*. Association for Computing Machinery, New York, NY, USA, 27–39. https://doi.org/10.1145/2789168.2790117

[16] W. Hu, W. Wang, R. Wu, H. Wang, L. Zeng, C. Xu, X. Gao, and F. Zhang. 2009. Efficient Binary Translation System with Low Hardware Cost. In *Proceedings of the IEEE International Conference on Computer Design*. 305–312.

[17] Chanyou Hwang, Saumay Pushp, Changyoung Koh, Jungpil Yoon, Yunxin Liu, Seungpyo Choi, and Junehwa Song. 2017. RAVEN: Perception-aware Optimization of Power Consumption for Mobile Games. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah, USA) *(MobiCom '17)*. Association for Computing Machinery, New York, NY, USA, 422–434. https://doi.org/10.1145/3117811.3117841

[18] Intel Corporation. 2024. *OSPRay: Open Source Ray Tracing Engine for Visualization.* https://github.com/RenderKit/ospray Accessed from GitHub repository.

[19] Jorrit Rouwe. 2024. *Jolt Physics: Open Source Physics Engine.* https://github.com/jrouwe/JoltPhysics?tab=readme-ov-file Accessed from GitHub repository.

[20] O. Khan. 2023. *Microsoft Azure Delivers Purpose-Built Cloud Infrastructure in the Era of AI.* https://azure.microsoft.com/en-us/blog/microsoft-azure-delivers-purpose-built-cloud-infrastructure-in-the-era-of-ai/ Microsoft Azure Blog.

[21] C. Li, Z. Liu, Y. Shang, L. He, and X. Yan. 2023. A Hardware Non-Invasive Mapping Method for Condition Bits in Binary Translation. *Electronics* 12, 14 (2023), 3014.

[22] T. P. Morgan. 2023. *The Interesting Years Ahead for Servers.* https://www.nextplatform.com/2023/01/04/the-interesting-years-ahead-for-servers/ The Next Platform.

[23] Prince Onyeanuna. 2024. *Overview of the Apple M1 chip architecture.* https://everythingdevops.dev/overview-of-the-apple-m1-chip-architecture/#:~:text=The%20Apple%20M1%20chip%20is,a%20Chip%20(SoC)%20architecture. Accessed online.

[24] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. 2011. Harmonia: A transparent, efficient, and harmonious dynamic binary translator targeting the Intel® architecture. 26. https://doi.org/10.1145/2016604.2016635

[25] ptitSeb. 2021. *Box64 - Linux Userspace* x86_64 *Emulator with a twist, targeted at ARM64 Linux devices.* https://github.com/ptitSeb/box64 GitHub project.

[26] Tahmid Noor Rahman, Nusaiba Khan, and Zarif Ishmam Zaman. 2024. Redefining Computing: Rise of ARM from consumer to Cloud for energy efficiency. *World Journal of Advanced Research and Reviews* 21, 1 (Jan. 2024), 817–835. https://doi.org/10.30574/wjarr.2024.21.1.0017

[27] F. Salgado, T. Gomes, S. Pinto, J. Cabral, and A. Tavares. 2017. Condition Codes Evaluation on Dynamic Binary Translation for Embedded Platforms. *IEEE Embedded Systems Letters* 9, 3 (2017), 89–92.

[28] Researcher SomeAuthor. 2020. Binary Translation at Loongson Lab: Combining Software and Hardware Approaches. *Loongson Research Journal* (2020).

[29] Matt Tate. 2024. *Nintendo Switch Models Compared: Switch vs Lite vs OLED.* https://www.stuff.tv/features/which-nintendo-switch-to-buy/ Stuff.

[30] W. Wang, C. Wu, T. Bai, Z. Wang, X. Yuan, and H. Cui. 2014. A Pattern Translation Method for Flags in Binary Translation. *Journal of Computer Research and Development* 51, 10 (2014), 2336–2347.

[31] Tom Wijman. 2023. *Newzoo's video games market size estimates and forecasts for 2022.* https://newzoo.com/resources/blog/the-latest-games-market-size-estimates-and-forecasts Accessed online.

[32] Chenggang Wu, Yongbiao Chen, Zhengwei Qi, and Haibing Guan. 2022. DSPR: Secure decentralized storage with proof-of-replication for edge devices. *Journal of Systems Architecture* 125 (2022), 102441. https://doi.org/10.1016/j.sysarc.2022.102441

[33] Taoran Xiang, Lunkai Zhang, Shuqian An, Xiaochun Ye, Mingzhe Zhang, Yanhuan Liu, Mingyu Yan, Da Wang, Hao Zhang, Wenming Li, Ninghui Sun, and Dongrui Fan. 2021. RISC-NN: Use RISC, NOT CISC as Neural Network Hardware Infrastructure. arXiv:2103.12393 [cs.AR] https://arxiv.org/abs/2103.12393

[34] W. Xie, Q. Luo, X. Tian, J. Huang, and F. Qi. 2024. Performance Improvements via Peephole Optimization in Dynamic Binary Translation. *Electronics* 13, 9 (2024), 1608.

[35] Yixiao Xu, Yubo Li, Wanzhao Xu, Yicheng Gu, Yun Wang, Jiangyuan Ma, and Zhengwei Qi. 2025. gFlow: Distributed Real-Time Reverse Remote Rendering System Model. In *MultiMedia Modeling: 31st International Conference on Multimedia Modeling, MMM 2025, Nara, Japan, January 8–10, 2025, Proceedings, Part II* (Nara, Japan). Springer-Verlag, Berlin, Heidelberg, 3–16. https://doi.org/10.1007/978-981-96-2061-6_1

[36] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. 2019. Mobile Gaming on Personal Computers with Direct Android Emulation. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 15 pages. https://doi.org/10.1145/3300061.3300122

[37] Nairan Zhang, Youngki Lee, Meera Radhakrishnan, and Rajesh Krishna Balan. 2015. GameOn: p2p Gaming On Public Transport. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (Florence, Italy) *(MobiSys '15)*. Association for Computing Machinery, New York, NY, USA, 105–119. https://doi.org/10.1145/2742647.2742660