`Chimera`: Transparent and High-Performance ISAX Heterogeneous Computing via Binary Rewriting

Jiatai He[1,2], Qinglin Pan[1,2], Ruilin Zhao[1,2], Ji Qi[1,3*], Kaiwen Liang[6,1,4,5], Jiahao Xu[1,2], Zhiyuan Li[2,1,4,5], Yuexiang Wang[1], Jiageng Yu[1], Yanjun Wu[1]

[1]Institute of Software Chinese Academy of Sciences [2]University of Chinese Academy of Sciences [3]Key Laboratory of System Software (Chinese Academy of Sciences) [4]University of Chinese Academy of Sciences, Nanjing [5]Nanjing Institute of Software Technology [6]Hohai University

Abstract

ISAX heterogeneous processors integrate cores that share a common base ISA, with certain cores offering extensions ISAs (e.g., vector extension) to accelerate computation. ISAX balances performance and energy efficiency while facilitating the reuse of existing software ecosystems. RISC-V, which adopts the ISAX architecture, has gained extensive attention in both industry and academia. Binary translation via binary rewriting enables transparent ISAX heterogeneous computing by translating extension instructions when migrating a program to cores without extension support. However, current binary rewriting approaches still struggle to achieve both high performance and correctness.

We propose `Chimera`, an ISAX heterogeneous computing system via binary rewriting that achieves both correctness and high performance. Prior binary rewriting methods ensure correctness by proactive fault checking, incurring unnecessary runtime overhead in normal executions unlikely to encounter faults. `Chimera` introduces a new binary rewriting method that passively triggers fault-handling only when faults actually occur, minimizing runtime overhead. We evaluated `Chimera` and notable ISAX heterogeneous computing systems using real-world workloads. In mixed matrix computational workloads, `Chimera` achieved only 3.2% performance overhead compared to native compilation, and only 5.3% in real-world workloads like OpenBlas. On SPEC CPU2017 benchmarks, our method achieved up to 42.5% performance improvement over existing binary rewriting approaches on average. `Chimera` 's code is released on https://github.com/Eurosys26p57/Chimera.

1 Introduction

As the open-source and modular ISA (Instruction Set Architecture) RISC-V [5, 8, 48] has seen increasing adoption in both industry and academia, ISAX (ISA eXtension) heterogeneity has gained significant popularity. ISAX heterogeneity is based on heterogeneous processors with the overlapping ISA: each core supports the same base ISA and can customize different extension ISAs for computation acceleration.

The overlapping-ISA heterogeneity offers unique strengths over single-ISA and disjoint-ISA heterogeneity. Compared to single-ISA heterogeneity (e.g., ARM big.LITTLE [44]), ISAX heterogeneity allows each core or processor to support instructions optimized for specific workloads, enabling a better performance-energy balance [17, 60–62]. Compared to disjoint-ISA heterogeneity, ISAX heterogeneity simplifies software porting. Porting software to disjoint-ISA heterogeneous processors requires developers to manage complex differences in application binary interfaces (ABIs) [19, 20, 23, 26, 55]. For instance, the notable disjoint-ISA heterogeneous computing system, Popcorn [20], requires implementing a new system infrastructure for communication between OS kernels for different ISAs, as well as specific support for compilation toolchains. In contrast, ISAX heterogeneous processors share a common ABI through their common base ISA [53], eliminating the need for extensive system modifications, and thus enhancing software portability.

Binary translation through static binary rewriting is especially suitable for ISAX heterogeneous computing. By translating *source instructions* into architecture-specific *target instructions* before execution, binary rewriting produces *rewritten binaries*, eliminating the interpretation or JIT overhead that plagues dynamic translators.

Binary rewriting offers both transparency and high performance for ISAX systems. First, extension instructions accelerate computations by batching the operations of base instructions [5, 9] and take only a tiny portion (5~10%) of all instructions in binaries (§2); translating this small subset into equivalent base sequences is therefore lightweight and preserves near-native speed, allowing tasks to migrate seamlessly across heterogeneous cores. Second, the original application binary, ABI, and toolchain remain unchanged, and no source-code modifications are required, avoiding the cumbersome dependency-resolving and re-build overhead inherent in compilation methods [59].

ISAX heterogeneous computing has two requirements on binary rewriting: **correctness** and **high performance**. For correctness, the rewritten binary must preserve the original binary's semantics after any *upgrading* (extension to base) or *downgrading* (base to extension). For high performance, rewritten binaries must run efficiently without extensive performance overhead (defined in §3.2).

While ensuring correctness, existing binary rewriting methods fail to achieve high performance [21, 27, 29, 49]. In

*Ji Qi (qiji@iscas.ac.cn) is the corresponding author.

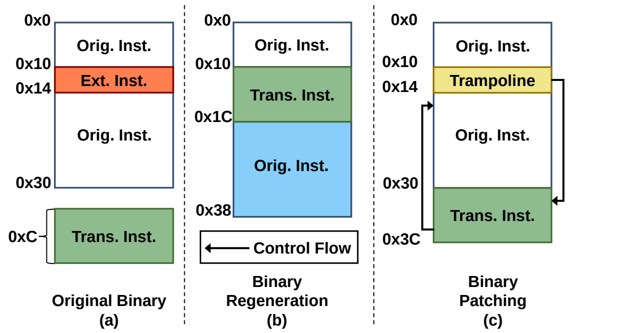

Figure 1. (a) When translating a source instruction to target instructions, binary regeneration (b) overwrites the source with target instructions by shifting subsequent instructions (0x1C~0x38); binary patching (c) replaces the source with a single-inst trampoline pointing target instructions.

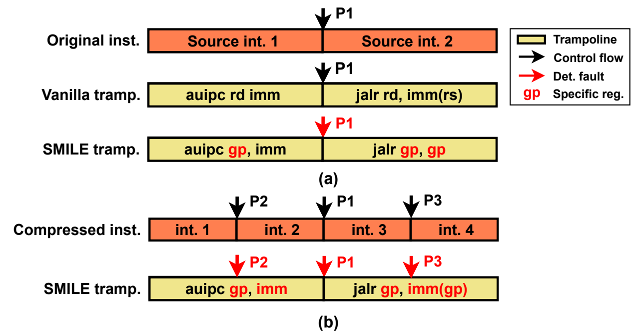

Figure 2. (a) Design of our *SMILE* trampoline. (b) Design of our *SMILE* trampoline for compressed instructions.

Figure 1a, *downgrading* an extension instruction often requires replacing it with multiple translated instructions, requiring shifting subsequent instructions for enough space. However, since binaries' control flows are tightly coupled with fixed instruction addresses at compile time (e.g., many *jump* targets are prearranged), such shifting corrupts the semantics of the original binary (e.g., *jump* to an unintended instruction, causing *erroneous jumps*). The same problem also exists in *upgrading*. To preserve the original binary's semantics, two methods exist: *binary regeneration* and *binary patching*, both incur high runtime overhead.

Binary regeneration [21, 49] ensures correctness through runtime checking, causing performance degradation. The method translates source instructions in place and corrects erroneous *jump* targets caused by instruction shifting. However, statically correcting all jump targets is impossible, as some can only be identified at runtime [32, 64]. For a *jump* with unidentified targets, the method performs runtime checks each time the jump is executed to detect and correct erroneous jump targets, resulting in 30-40% performance degradation [21].

Binary patching [27] replaces a source instruction with a *single-inst* trampoline targeting translated instructions (Figure 1c) without shifting the subsequent instructions, but introduces heavy, trap-based trampolines. Due to ISA encoding constraints, a *jump* instruction has limited bits to encode its target address, leading to a constrained jump range (e.g., ±1MB in RISC-V). In a large binary such as the OpenCV Graph API compiled with the vector extension, about 70% of target instructions cannot be reached by a single jump instruction. RISC architectures do support long-distance trampolines. In RISC-V, for example, the multiple-inst long trampoline uses *auipc* to load the jump target and *jalr* to perform the jump (Figure 2a). However, since the vanilla trampoline contains two instructions, if control flow jumps to the second instruction, the target address not correctly set by the preceding *auipc* will cause an unintended jump and break correctness. Therefore, existing works propose trap-based trampolines that rely on runtime mechanisms to redirect control flow, causing up to 50% performance degradation (§2).

In summary, existing binary rewriting methods rely on runtime mechanisms to ensure correctness, at the cost of high performance overhead. We attribute this to their design philosophy of *proactively trading normal execution performance for security against potential control-flow faults caused by erroneous jumps*. For example, binary regeneration inserts runtime checks before possible erroneous *jump* instructions in normal executions.

We observe that the root cause of this overhead lies in the non-deterministic behavior resulting from erroneous *jumps*, which forces the use of conservative fault-handling strategies. An erroneous jump can lead to unpredictable program behavior after executing several unintended instructions. To reconcile correctness with high performance, our key insight is that: **if all potential erroneous jumps can be made to trigger deterministic faults, then fault handling can perform passively in only erroneous executions, without impacting normal execution performance.**

We proposes `Chimera`, a novel ISAX heterogeneous computing system via binary rewriting. The core of `Chimera` is `CHBP`, a Correct and High-performance Binary Patching method. `CHBP` leverages **special registers** in ABI (e.g., *gp* register in RISC-V [5, 53]) to design a *SMILE* trampoline (Secure Multiple-Instruction Long-distancE trampoline), ensuring that erroneous executions always jump to an invalid address, such as the data segment, to raise deterministic faults.

As shown in Figure 2a, to avoid shifting subsequent instructions, `Chimera` replaces two adjacent instructions with RISC-V's vanilla trampoline to enable long-distance jumps. However, the second trampoline instruction (*jalr*) becomes a potential jump target (P1) to partially execute the trampoline. To address this, our *SMILE* trampoline leverages the ABI-specified special register, the *gp* register in RISC-V, to perform the jump. In normal execution, *auipc* correctly modifies the *gp*'s value to the address of target instructions, allowing *jalr* to jump to the correct location. In erroneous execution, since the ABI ensures the *gp* register points to the data segment, executing only the *jalr* instruction triggers a deterministic segmentation fault.

CHBP still faces two challenges. First, many ISAX processors, such as RISC-V processors with the compression extension [5, 52], support compressed instructions, introducing extra potential jump targets within our *SMILE* trampoline. An original binary with compressed instructions contains both 2-byte and 4-byte instructions. If an 8-byte *SMILE* trampoline is overwriting four 2-byte instructions, two extra jump targets, P2 and P3, appear in the trampoline instructions (Figure 2b). Since P2 and P3 point to the middle of the trampoline's jump target address, we handle such erroneous executions by carefully arranging *SMILE* trampolines' jump targets, ensuring that any attempt to execute from these targets triggers a deterministic illegal instruction fault.

The second challenge lies in selecting registers for trampolines to jump back from target instructions to original instructions. In RISC architectures, long jumps are register-based and require a dead register (whose value is not used by subsequent instructions [46]), to hold the jump target. Although *gp* can be used to jump to target instructions (§4.2), we need another dead register to jump back after execution. Existing methods use binary register liveness analysis to identify dead registers, which can fail due to the limitations in binary data flow analysis [32, 46] and high register pressure in compute-intensive tasks. To find a dead register in most cases, we reduce the registers used by subsequent instructions by including more subsequent instructions into the translated block.

We evaluated Chimera with a state-of-the-art compilation-based ISAX heterogeneous computing system MELF [59], and evaluated CHBP with state-of-the-art binary rewriting methods, including ARMore [27] and Safer [49]. Our evaluation assesses the performance and correctness of Chimera using several real-world applications widely used in previous works, including Debian [4] packages and SPEC CPU2017 [2]. Our evaluation results show that:

- Chimera achieved high performance in ISAX heterogeneous computing, incurring only 3.2% performance overhead on average compared to MELF.
- CHBP achieved high performance in binary rewriting. The rewritten binaries achieved up to 42.5% higher performance than ARMore and Safer.
- CHBP achieved correctness via the passive fault handling strategy, incurring only 2.1% performance overhead on average.

Our main contribution is CHBP, a novel binary rewriting method for ISAX heterogeneous computing that achieves both correctness and high performance through passive fault handling. Unlike prior works, the passive fault handling avoids performance penalties on normal executions, thereby achieving high performance. The innovations behind CHBP can be extended to other research areas (e.g., binary hardening [28, 30, 58, 64], binary hot-patching [22, 35, 51], and fuzzing [65]), because they also require correctly and efficiently applying binary patches.

Table 1. Comparison of Chimera and related works.

System	Need Source Code	Low Porting Effort	Correctness	High Perf.
SCHEDULING				
FAM [39]	No	Yes	Yes	No
COMPILATION				
MELF [59]	Yes	No	Yes	Yes
BINARY REGENERATION				
Multiverse [21]	No	Yes	Yes	No
Safer [49]	No	Yes	Yes	No
Egalito [64]	No	Yes	No	Yes
SURI [36]	No	Yes	No	Yes
BinRec [13]	No	Yes	No	Yes
BINARY PATCHING				
ARMore [27]	No	Yes	Yes	No
PIFER [50]	No	Yes	Yes	No
Chimera (ours)	No	Yes	Yes	Yes

2 Background

We compared Chimera with related works in Table 1.

2.1 ISAX Heterogeneous Computing

Scheduling-based methods. A binary for ISAX heterogeneous computing system often includes extension instructions not supported by all cores. Scheduling-based methods [39] address this via fault-and-migrate (FAM): when a core encounters an unsupported instruction, it triggers an illegal instruction fault, prompting the scheduler to migrate execution to a compatible core. However, FAM cannot ensure every instruction has a compatible core, limits scheduling flexibility and under-utilizes hardware. Our evaluation shows that FAM introduces about 33.1% overhead in end-to-end latency compared to other methods (§6.1).

Compilation-based methods. Prior works proposed compiler-level support for ISAX heterogeneous computing [17, 39, 60–62]. They compile functions into multiple versions, allowing them to run on heterogeneous cores with high performance [59]. However, they require the source code. As ISAX processor extensions are increasingly diverse, developers may not know all available extensions on target machines. Thus, pre-compiling all possible ISA extensions is impractical, especially for legacy or closed-source applications whose source code is unavailable. Compilation can also be costly, for example, compiling SPEC CPU2017 costs 10 hours on Banana Pi BPI-F3 with a SpacemiT K1 8-core RISC-V 1.6GHz CPU [18], while Chimera just costs 40 minutes. Additionally, compiling source code is often complex, requiring specific toolchains and compatible libraries.

Another available approach is to distribute applications in an intermediate representation (IR) and compile the IR to binary on the target hardware, thereby reducing compilation overhead. However, as RISC-V extensions proliferate, maintaining multiple hardware-vendor-specific IR dialects (e.g., LLVM IR with different custom IR intrinsics [38]) and toolchains imposes substantial overhead on software developers. Since no single IR dialect has achieved a broad consensus [56], users must install a separate toolchain for each vendor's IR dialect, degrading the user experience, especially for non-technical users. Additionally, as most applications are distributed in binaries, providing them as IR is largely incompatible with the existing software ecosystem.

Motivation 1. *Binary rewriting* is suitable for ISAX heterogeneous computing. As extension instructions (or base instructions that can be upgraded to extension instructions) constitute only a small portion of the binary (about 3% in OpenCV [6]), most instructions do not need rewriting. Consequently, the overhead induced by trampolines is negligible (about 3.2% in `Chimera`, see §6.2). Moreover, binary rewriting does not require source code or customized compilation toolchains, making it more practical for ISAX heterogeneous computing with diverse extensions.

2.2 Binary Rewriting

Correctness of binary rewriting. Correctness of binary rewriting means rewritten binaries should have the same semantics as the original ones. However, accurate control flow recovery (§1), especially accurately determining all potential targets of indirect jumps involving pointers and jump tables, is still an unsolved problem [15, 37, 49, 63, 64].

Although many binary regeneration methods [15, 25, 32, 36, 37, 63, 64] use heuristics and binary metadata, like relocation information (Egailtio [64]) or security metadata (SURI [37]), to recover control flow, the correctness issue still remains. The notable study BinRec [13] addresses this using dynamic and incremental regeneration to recover erroneous control flow when faults occur at runtime. However, BinRec is unsuitable for heterogeneous computing as runtime faults can cause unrecoverable side effects. In contrast, `Chimera` only triggers deterministic faults that are side-effect free.

Performance of binary rewriting. For correctness, existing methods rely on runtime mechanisms but induce a significant performance degradation.

For binary regeneration, previous methods ensure correctness by proactively checking and correcting the target of each indirect jump in both erroneous and normal executions, causing significant performance overhead. For instance, Multiverse [21] uses a lookup table to correct all indirect jumps in regenerated binaries at runtime, causing above 30% performance overhead. Safer [49] encodes indirect jump targets that have been statically corrected and checks them at runtime. Encoded targets can jump directly, while only unencoded targets require correction via a table, reducing the frequency of table queries. However, it still struggles with complex binaries, causing around 40% performance overhead in the perlbench benchmark of SPEC CPU2017 (§6.2); moreover, when an encoded jump target is modified by an undetected control flow (e.g., add an offset), Safer can only detect the erroneous target but cannot correct it, and thus cannot guarantee correctness.

For binary patching, to avoid *multiple-inst* long trampolines to corrupt correctness, previous approaches [50] use trap-based trampolines, but also introduce significant performance overhead. ARMore [27] relocates all original instructions to a new code section, where source instructions are translated while others remain unchanged. In the original code section, each instruction is replaced with a single-inst trampoline to maintain the mapping between original and relocated instructions. Indirect jumps in the relocated code still use original addresses as targets, which point to the corresponding trampolines in the original section. These trampolines then redirect control flow to the correct instructions in the relocated section. ARMore achieves a low performance overhead (about 1%) on ARM [1] binaries where a single `jump` instruction reaches up to 128MB. However, it is impractical for the next-generation RISC-V architecture, whose jumping distance of one *jump* instruction is only ±1MB to reduce instruction encoding types for power saving and improving hardware extensibility [5]. For many applications (e.g., Vim [57] and Git [34]), code sections are larger than 1MB, ARMore must use trap-based trampolines to redirect control flows, inducing unacceptable performance overhead (§6.2).

Motivation 2. Compared to prior binary rewriting methods, `Chimera` achieves both high performance and correctness through *passive fault handling*. For high performance, `Chimera` avoids the vast majority of heavy trap-based trampolines (98.97% on average, see §6.2) and proposes a novel *multiple-inst* trampoline, which triggers recoverable deterministic faults in only erroneous executions. `Chimera` adopts a passive fault-handling mechanism to recover these faults, ensuring correctness without degrading the normal execution performance.

3 Overview

3.1 System Setup

The architecture of `Chimera` is shown in Figure 3. `Chimera` aims heterogeneous processor environment (Figure 3a), which consists of heterogeneous ISAX **processors** (processor 1 ~ processor 3) and heterogeneous **cores** within a single processor (processor 2). Given an original binary compiled for a specific ISA (RISC-V in our experiments), `Chimera` first converts it into rewritten binaries by static binary rewriting (§4.1 and §4.2) and then guarantees execution correctness and high performance on heterogeneous cores through runtime mechanisms (§4.3).

3.2 Chimera's Guarantee

An *execution flow* refers to the sequence of instructions executed at runtime, including control transfers such as jumps, branches, and calls. The *semantics* of a binary denotes the intended behavior it exhibits. The "*erroneous executions*" are the execution flows that corrupt the original binary's semantics after rewriting. As discussed in §1, binary regeneration may cause erroneous execution flows, but the limitations in

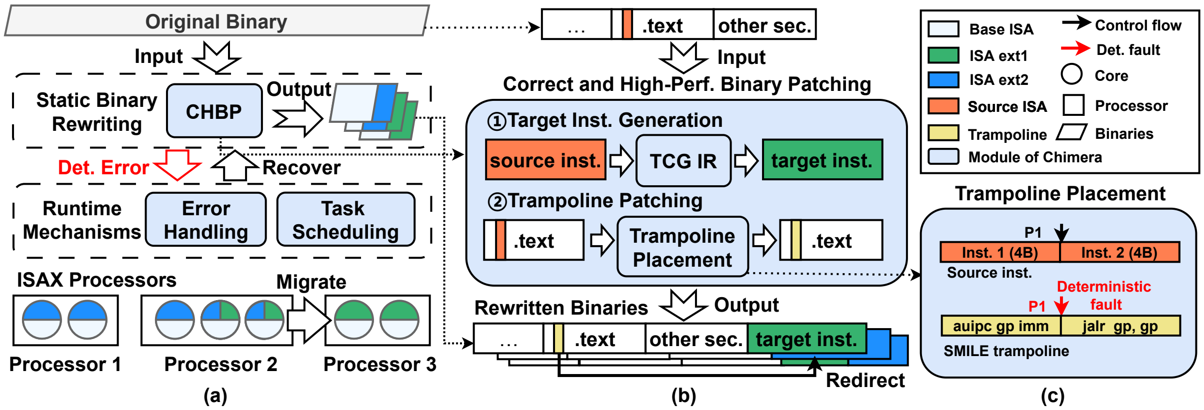

Figure 3. The architecture of Chimera. The *static binary rewriting* prepares the rewritten binaries for ISAX cores. The *runtime mechanisms* schedule tasks among ISAX processors and handles deterministic faults raised by erroneous executions.

binary analysis (§2.2) hinder complete correction. In contrast, "*normal executions*" preserve the original binary's semantics.

We categorize faults triggered by erroneous executions into two types: "*non-deterministic faults*" and "*deterministic faults*". Non-deterministic faults may lead to unpredictable program behaviors. For instance, a jump to an incorrect target address can cause the program to execute unintended instructions. Deterministic faults trigger specific faults that immediately halt execution (e.g., illegal instruction faults).

Assertion 1 (Correctness of *Chimera*). *Chimera guarantees the correctness of all rewritten binaries by ensuring that any erroneous execution triggers a deterministic fault, which is detected and recovered at runtime.*

Therefore, the rewritten binaries generated by Chimera are consistent in semantics with the original binary and maintain their correctness.

3.3 Analysis of Chimera's Performance Guarantee

Assertion 2 (High Performance of *Chimera*). *In normal executions, Chimera's fault-handling incurs only the overhead of executing SMILE trampolines.*

Prior binary rewriting methods correct all erroneous executions by checking every control flow transfer (e.g., jumps), inducing high costs in both normal and erroneous executions (e.g., Safer, see §2.2). Chimera captures erroneous executions passively with its *SMILE* trampolines, introducing negligible overhead to normal executions (e.g., additional jump). By combining *SMILE* trampolines with its runtime fault handling mechanisms, Chimera achieves both correctness and high performance for all rewritten binaries.

Chimera's requirements. Chimera is designed for RISC-V ISAX processors, but its principles are general to other RISC architectures with fixed instruction lengths. CHBP's *SMILE* trampoline (§4.2) leverages the illegal instruction encoding and special registers such as *gp* in RISC-V. Chimera needs kernel modifications to implement its runtime mechanisms (§4.3).

• *Illegal instruction.* We use two types of illegal instruction encoding to trigger illegal instruction faults (§4.2). The first is an unsupported instruction prefix. RISC-V reserves a large encoding space for future extension instructions longer than four bytes, whose lower encoding is all "11111". These extension instructions will not be enabled before the space is used up, because longer instructions induce power consumption and cache problems [53]. The second is the standard compressed extension, including 128 reserved instructions unlikely to be exhausted in the future. *SMILE* uses one of them.

• *The gp register.* In the *SMILE* trampoline, the register that holds the target address must be restorable after being overwritten; moreover, if the trampoline is partially executed, the register's original value must point into data segmentation and thus trigger a deterministic fault (§4.2). Most general registers cannot meet these two requirements because their runtime values cannot be determined statically during the rewriting. The *gp* register in RISC-V satisfies this requirement [53].

Under the RISC-V ABI, *gp* is designed to reduce code size by curtailing frequent data accesses from two instructions to one instruction. Specifically, *gp* points to a fixed data-segment address, allowing load/store address to be calculated from *gp*. This ensures that *gp*'s value is calculated at compile time and is read-only, allowing the *SMILE* trampoline to recover *gp*'s value in target instructions (§4.2).

Besides RISC-V, the *SMILE* trampoline can also use "gp-like" register in other ISAs (e.g., MIPS's *$28* register and ARM's *r9* register under certain ABIs). For ISAs without such a register, we can construct the *SMILE* trampoline with a general register storing a data pointer (§4.2), though this may increase the reliance on trap-based trampolines.

• *Kernel modifications.* Because Chimera needs to catch and recover deterministic faults at runtime (e.g., segmentation faults) and support task scheduling across heterogeneous ISAX cores, it requires modifications to the kernel's fault-handling and migration components.

Furthermore, even if Chimera can leverage the properties of ABI to guarantee that *gp* can be restored correctly, a signal delivered while the *gp* is temporarily overwritten by the *SMILE* trampoline still causes the user-space signal handlers to observe the incorrect *gp* value [53]. Therefore, `Chimera` requires kernel modifications to ensure compatibility with the existing signal-handling mechanism (§4.3).

3.4 `Chimera`'s Workflow Overview

`Chimera`'s workflow comprises the static binary rewriting which prepares rewritten binaries for ISAX cores, and runtime mechanisms which correctly and transparently execute tasks on ISAX cores.

`Chimera`'s static binary rewriting. Given an original binary, `Chimera` uses `CHBP` to prepare a rewritten binary for each heterogeneous core by *upgrade* and *downgrade*. Instruction *downgrade* translates unsupported extension instructions into semantically equivalent base instructions, while instruction *upgrade* optimizes base instructions to more efficient extension instructions. Therefore, `CHBP` allows a program to efficiently run on ISAX processors and cores. `CHBP` prepares a rewritten binary in two steps (Figure 3b).

• *Step 1: target instruction generation.* Depending on which ISA extensions the core supports, `CHBP` first scans the original binary to identify all instructions that need binary rewriting (*upgrade* or *downgrade*). We refer to these instructions as *source instructions* and the corresponding translated instructions as *target instructions*. The target instructions preserve the semantics of source instructions and use only supported extensions. `CHBP` statically translates source instructions, generating both target instructions and necessary instructions to use/simulate additional registers (§4.1).

• *Step 2: trampoline patching.* `CHBP` creates a copy of the original binary and patches the copy by replacing source instructions with trampolines targeting the corresponding target instructions (§4.2). During patching, it is challenging to achieve both correctness and high performance (§3.2). `Chimera` tackles this challenge with its novel *SMILE* trampoline.

The *SMILE* trampoline passively triggers deterministic faults on erroneous jumps at runtime without sacrificing the normal execution performance. We design the trampoline in two steps: First, we place RISC-V's vanilla multiple-inst long-distance trampoline by overwriting the source instruction and its adjacent instructions (inst 1 and 2 in Figure 3c), allowing erroneous executions targeting these original instructions to partially execute the trampoline (e.g., jump to P1 and execute only the *jalr* instruction). Second, we ensure that such partial execution triggers a deterministic fault. In Figure 3c, partially executing the *SMILE* trampoline will use the unmodified *gp* register as the jump target. Since *gp* points to the non-executable data segment, this results in a deterministic segmentation fault.

`Chimera`'s runtime mechanisms. `Chimera`'s runtime loads rewritten binaries, transparently schedules program tasks among ISAX cores, and handles deterministic faults triggered in only erroneous executions (§4.3).

Overall, `Chimera` achieves correct and high-performance heterogeneous computing via binary rewriting. Previous studies [21, 27, 49] ensure correctness by introducing substantial traps or proactive fault checks, incurring about 30.7% performance overhead [21]. `Chimera` passively handles runtime faults that actually occur, significantly reducing the invocation frequency of the fault-handling mechanism. This is achieved by our novel *SMILE* trampoline which guarantees that any erroneous execution triggers a deterministic recoverable fault. The heavy fault-handling is restricted to rare erroneous executions, keeping normal executions largely unaffected, and incurring only a 5.3% overhead due to trampoline-based control flow redirection (§2.2).

4 Protocol Design

4.1 Target Instructions Generation

`Chimera` recursively disassembles a binary using IDA Pro [3] to ensure the recognized instructions are correct. However, it does not ensure completeness, meaning some instructions may remain unrecognized. If an unrecognized extension instruction is executed on an unsupported core, it triggers an illegal instruction fault. `Chimera` detects such faults and rewrites the unrecognized instructions at runtime (§4.3). Given source instructions, `Chimera`'s `CHBP` produces corresponding target instructions consisting of (1) computation instructions, and (2) register manipulation instructions. We leverage translation templates in Qemu TCG [7] to produce computation instructions, which may require additional registers beyond those used in source instructions due to two types of register mismatches.

Use extra base registers. A batch processing extension instruction can be translated into a sequence of base instructions requiring additional base registers to store intermediate results. For example, the RVB extension instruction *sh1add a0, a1, a2* (shift left by one and add), can be translated into two base instructions: *slli a3, a2, 1*, which shifts *a2* left by one and saves the result to *a3*, and *add a0, a3, a2*, which adds *a3* and *a2*, saving the result to *a0*. Here, register *a3* temporarily holds the shifted result and overwrites its previous value. For correctness, we insert stack manipulation instructions to save and restore *a3* in the stack around the computation. When multiple registers are involved, their saves/restores are ordered in a first-in, last-out manner to ensure correct restoration.

Simulate unsupported extension registers. Source instructions may use extension-specific registers whose lengths are larger than 64-bit base registers (e.g., the 256-bit vector registers in the RISC-V V extension [9]). These registers maintain the computation context across instructions and must be accurately simulated by `CHBP`. For instance, the output vector register of a *vadd* instruction can serve as input to a following *vadd* instruction. To preserve this context on cores without the extension, we simulate extension registers using a dedicated readable/writable data section in the

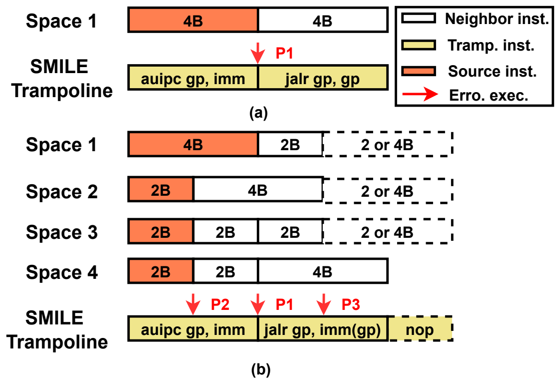

Figure 4. Trampoline placement of `Chimera`. A space contains a source instruction and its adjacent instructions. Our *SMILE* trampoline overwrites instructions in the space. For a space larger than 8 bytes, we insert an extra 2-byte *nop*. All erroneous potential jumps falling within the trampoline trigger a deterministic fault (§4.2) and are handled at runtime (§4.3).

rewritten binary. Each simulated register maps to a reserved memory region. During translation, register accesses are replaced by memory accesses to this region, ensuring consistent behavior across heterogeneous cores.

4.2 Trampoline Patching

`CHBP` leverages the special register *gp* to design the *SMILE* trampoline, ensuring erroneous jumps that partially execute the trampoline to trigger a deterministic fault.

Trampoline construction. `Chimera` extends RISC-V's vanilla trampoline to construct a *SMILE* trampoline that triggers deterministic faults on partial execution.

• *RISC-V's vanilla multiple-inst trampoline.* The vanilla *multiple-inst* trampoline has ±2GB *pc*-relative jumping range. The trampoline contains two 4-byte instructions, an *auipc* and a *jalr*. The first instruction sets the target address based on *pc*, and the second one adds an offset to the target address and then jumps to it. Specifically, "*auipc*, rd_1, *imm*" sets the target address by adding an immediate number to the value of *pc* (i.e., the address of *auipc*). The immediate number's upper 20 bits are *imm* and lower 12 bits are zero. The target address is written in rd_1; "*jalr* rd_2, *imm*(rd_1)" reads the address in rd_1, adds a sign-extended immediate number *imm*, and then jumps to the resulting target address. After the jump, *jalr* writes $pc + 4$ to its rd_2 register as the return address.

• `Chimera`*'s SMILE trampoline.* As shown in Figure 4a, a *SMILE* trampoline differs from the vanilla one in two aspects. First, we replace rd_1 with *gp* to store the target address. This is because the unmodified value stored in *gp* points to the data segment, so erroneous executions executing only *jalr* will jump to the data segment and try to execute an instruction from there. Such behavior is considered unsafe and forbidden by modern operating systems. The OS sets the data segment as non-executable, and any execution attempt

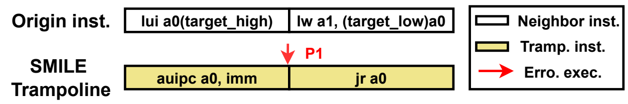

Figure 5. The *SMILE* trampoline using a general register storing a data pointer. Erroneous executions (P1) will trigger deterministic segmentation faults, as the unmodified *a0* still points to the data segment.

from it triggers a segmentation fault. Second, because the unmodified value of *gp* can be determined statically, we use *gp* (in place of rd_2) to hold the return address. Once the jump completes, we restore *gp*'s original value (Figure 6), ensuring that overwriting *gp* doesn't compromise program semantics.

• *SMILE trampoline using general register.* We can also construct the *SMILE* trampoline with a general register storing a data pointer when the ISA doesn't have a register like *gp*. As shown in Figure 5, the original inst. is a general memory-accessing sequence to load data from the static target address *target*. The *lui a0, target_high* sets the higher bits of *target* (*target_high*) in the general register *a0*. The *lw a1, (target_low)a0* calculates the target address *target* by adding the lower bits of *target* (*target_low*) to *a0* and load data from *target* to *a1*. We can replace this sequence with the *SMILE* trampoline in Figure 5. After replacement, normal executions correctly execute the trampoline and restore *a0* by executing the original *lui* in the target instructions; erroneous executions (P1) that partially execute the trampoline (only *jr a0*) trigger deterministic segmentation faults, as the unmodified *a0* still points to the data segment.

To construct such a trampoline, we need to identify the memory-accessing sequence preceding the source instruction in the same basic block, replace it with our *SMILE* trampoline, and copy intervening instructions into target instructions. Since not all source instructions contain such a memory-accessing sequence, we must fall back to a trap-based trampoline for them.

Trampoline placement. To avoid shifting subsequent instructions, `Chimera` places a *SMILE* trampoline by overwriting the source instruction in place. Since an 8-byte *SMILE* trampoline is longer than a 4-byte source instruction, `Chimera` must additionally overwrite the succeeding adjacent instruction of the source instruction (referred to as "neighbor").

For correctness, `Chimera` copies the overwritten neighbor into the target instructions. In Figure 6a, suppose the translated instructions *downgrade* source inst. 1. `Chimera` places a copy of the neighbor (copied inst. 2) after the translated instructions, followed by a trampoline that jumps back to inst. 3 in the rewritten binary. In normal executions, the neighbor executes after the translated instructions. In erroneous executions that jump to P1, `Chimera`'s runtime redirects control flow to the neighbor (§4.3). If the target instructions *upgrade* a sequence of source instructions (e.g., inst. 1 ∼ 3 in

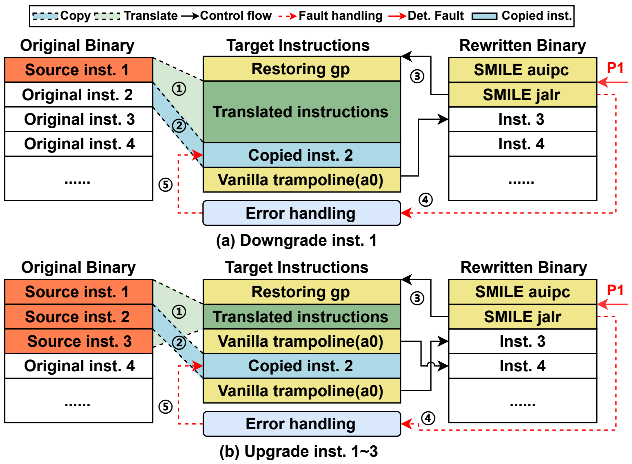

Figure 6. `Chimera` translates source instructions (①) and copies their neighbors (②). After executing the *SMILE* trampoline, a normal execution (③) first restores the value of *gp* that was overwritten by *SMILE* trampoline; then executes translated instructions and finally executes copied instructions (a) or skips copied instructions (b). For an erroneous execution (P1), `Chimera` determines its fault address (④) and redirects it to the copied neighbor (⑤).

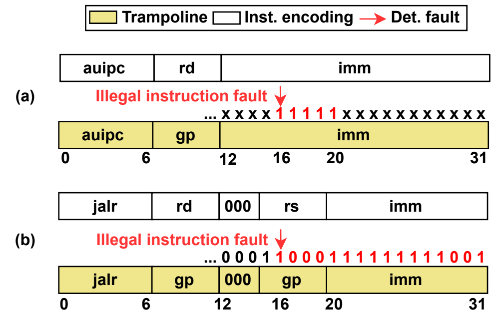

Figure 7. Trampoline placement and encoding.

Figure 6b), `CHBP` also copies inst. 2 into the target instructions, but inserts another trampoline between the translated instructions and the copied neighbor. This ensures that normal executions skip the copied inst. 2, while erroneous executions can still be safely redirected to it.

Additionally, to enhance performance, `CHBP` copies a series of target instructions (along with their intervening instructions), whose source instructions belong to the same basic block, into the first source instruction's target instructions, allowing the series of target instructions to be executed via one trampoline. All original trampolines within the basic block are still preserved to handle external jumps into the block. This optimization is widely used in prior binary-patching works [27, 46].

Challenge 1: instruction compression. Since the compression extension is widely enabled on ISAX processors (Arm and RISC-V), both a source instruction and its neighbor can be 2-byte. Overwriting 2-byte instructions introduces more potential jump targets in the middle of trampoline instructions (e.g., P2 and P3 in Figure 4b). It is challenging to trigger deterministic faults for the extra jump targets.

`Chimera` tackles this challenge by meticulously arranging the target addresses of trampoline instructions. P2 and P3 only appear when the compression extension is enabled. Since they point to bit 16 of both *auipc* and *jalr*, and the processor parse instructions from lower to higher address, we encode the higher 16 bits of the trampoline instructions as a 2-byte illegal instruction, triggering an illegal instruction fault. For *auipc* (Figure 7a), since it determines *SMILE* trampoline's jump range (upper 20 bits in the target address), we only confine bits 16-20 of *auipc* as "11111", without reducing the max jump range of our *SMILE* trampoline compared to the vanilla trampoline. A *SMILE* trampoline obtains a ±2GB jumping range by changing the higher bits 21-32 of *auipc* and thus can jump out of the original code section. Furthermore, *auipc* obtains the jump target by adding its own address with its *imm* field as an offset. By setting the offset of each *SMILE* trampoline, `Chimera` flexibly arranges all target instructions.

For *jalr* (Figure 7b), because the trampoline needs *gp* to jump, which restrics bits 15-19 of *jalr* to "11000", we encode bits 16-31 of *jalr* to a 2-byte illegal instruction starting with "1000". This instruction is reserved by RISC-V (§3.2). The encoding adds an immediate offset to trampoline's target address, and we position target instructions accordingly.

Challenge 2: register selection. The second challenge is the register selection for trampolines to jump back from target instructions to the original instructions. In ISAX architectures (e.g., RISC-V), one trampoline requires an *auipc*, which uses a register to store the target address. This register must be a dead register whose *current value* is not used by any subsequent instructions following control flows [46]. Overwriting a dead register will not corrupt the semantics of subsequent instructions. In Figure 8, each patching requires two long-distance trampolines, the first is a *SMILE* trampoline for jumping to the head of target instructions (*entry position*) and the second can be a vanilla trampoline for jumping back from the tail of target instructions (*exit position*).

Although *SMILE* trampolines can directly use *gp* for *entry position*, we cannot use *gp* for *exit position* because the value of *gp* must be restored before jumping out to the *exit position*. `CHBP` sequentially employs two strategies for selecting a dead register for the returning trampoline. First, `CHBP` uses register liveness analysis to find a dead register for *exit position*. However, the register liveness analysis techniques [46] may fail to find a dead register for *exit position* due to the limitations of binary data flow analysis [32, 46] and high register pressure in compute-intensive tasks. Second, if the register liveness analysis fails, `CHBP` adjusts the target instructions to shift the *exit position* to a subsequent position that confirms a dead register. Specifically, `CHBP` tries finding an instruction having a dead register, by following the control flow after

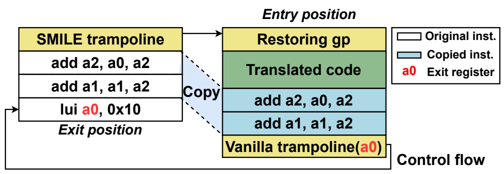

Figure 8. Exit register selection. We copy subsequent instructions to be executed as target instructions (blue instructions) and thus alter the *exit position*, thereby increasing the possibility of finding exit registers (red registers).

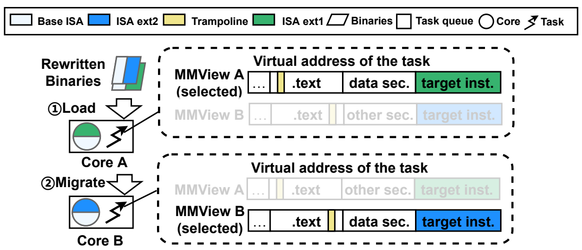

Figure 9. MMViews of Chimera. The rewritten binary corresponding to each core is loaded into a distinct MMView. When a task is scheduled to core A (①), it executes within the MMView corresponding to core A (MMView A). When the task migrates from core A to core B (②), it switches to the MMView corresponding to core B (MMView B) and continues execution.

the current *exit position*. Then, CHBP shifts the *exit position* to the found instruction, copying instructions between the old and new *exit position* into target instructions. If the found instruction lies in a different basic block than the current *exit position* (for example, due to intervening branches or jumps), Chimera merges the intervening basic blocks into the target instructions so that control flow and semantics are preserved.

For example, in Figure 8, shifting the *exit position* to the *lui* instruction helps select *a0* as a dead register. Two extra instructions are copied into target instructions. After executing target instructions, the program jumps to the new *exit position* and overwrites the value of *a0*. While this strategy finds dead registers in most cases (98.97% on average, see §6.2), it may fail under high register pressure. CHBP then falls back to a trap-based trampoline, following prior work [46].

4.3 Runtime Mechanism

Task scheduling. Unlike the conventional process model [40], Chimera loads a program as a process with multiple address spaces, each of which is instantiated from a rewritten binary corresponding to an ISAX core. All address spaces of a process share the same data segmentation. Chimera achieves this process model using MMViews, similar to related works [54, 59].

An address space of a process consists of a list of *Virtual Memory Allocations (VMAs)* [42] and a page directory that maps those VMAs to physical page frames. VMA list and page directory are stored together in a *Memory map (MM)* [41]. MMViews enable a process to maintain multiple address spaces by creating several *MM*s for that process [54]; each of such *MM* is called a MMView. The kernel chooses which MMView to activate at runtime.

As shown in Figure 9, when a task is loaded on core A (①), Chimera loads each ISAX core's rewritten binary into different MMViews; in each MMView, VMAs containing code segmentation points to the physical page frames holding its rewritten binary's code, while VMAs containing data segmentation share a common set of data physical page frames across all MMViews. Then, the MMView corresponding to the current core (MMView A for core A) is selected for execution, whereas other MMViews (e.g., MMView B) remain resident but inactive (the semi-transparent part). When the task migrates to core B (②), it switches to MMView B, which implements the target instructions supported by core B, while MMView A is retained in memory without execution.

Additionally, although rewritten binaries (§3) have the same semantics, the target instructions pointed to by the same *pc* value might not be semantically equivalent; If a migration occurs and the *pc* value is within the target instructions, Chimera delays the migration by inserting a probe [16] at the *exit position* of the target instructions (in Figure 8). Chimera migrates the task once the probe is triggered.

Through the mechanism described above, Chimera can integrate with existing schedulers (e.g., heterogeneous-ware CFS schedulers [39]).

Runtime fault handling. Deterministic faults in Chimera arise from either unrecognized extension instructions or partially executed *SMILE* trampolines. For each fault, Chimera determines its address and root cause, then handles the fault according to the fault-handling table of the rewritten binary.

- *Fault-handling table construction.* The fault-handling table maps potential fault addresses (e.g., P1 in Figure 6a) to their corresponding redirection targets (e.g., Copied inst. 2). When placing a *SMILE* trampoline, CHBP copies neighboring instructions to new locations and overwrites the originals. CHBP records the original address, such as P1, P2, and P3 in Figure 4b, as keys in the table, and maps each key to the new address of the copied instruction (e.g., ② in Figure 6a).
- *Determine the fault address.* When a deterministic fault occurs, Chimera determines its address based on its execution context and fault type. For an illegal instruction fault, the fault address is directly available in the *pc* register. For a segmentation fault, the erroneous execution must have partially executed the latter instruction (a *jalr*) of a *SMILE* trampoline, pointing the *pc* to the data segment. Chimera then determines the fault address using the return address stored in *gp* (§4.2): before jumping, the *jalr* stores the address of its next instruction into the *gp* register. The fault address is then computed by subtracting 4 from this return address.
- *Redirection/Rewriting.* Once identifying the fault address, Chimera further discriminates its root cause with the fault-

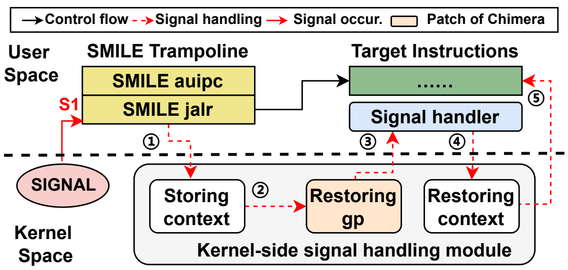

Figure 10. Adapt Chimera's signal handling for compatibility with user-space handlers. Chimera restores the program's *gp* value while the kernel is handling a signal, ensuring that a user-space registered signal handler observes the correct *gp* value during execution.

handling table. If a fault address is an existing key, it points to a neighbor instruction overwritten by a *SMILE* trampoline (§4.2). Chimera then gets the value as the redirection target and then resumes execution (⑤ in Figure 6a). Otherwise, it points to an unrecognized extension instruction (§4.1). Chimera rewrites the instruction and resumes execution.

Signal handler. Chimera modifies the kernel's signal handling module to prevent user-defined handlers from overriding its fault handling mechanism. It gives priority to segmentation faults and illegal-instruction faults (*SIGSEGV* and *SIGILL* [43]) produced by CHBP: when such a signal occurs, the kernel first checks whether it was generated by CHBP; if so, the signal is routed to Chimera's fault handling mechanism, otherwise the kernel falls back to its standard handling (i.e., delivering the signal to the user-defined handler).

Moreover, if a signal is delivered at the moment the process is executing the *SMILE* trampoline and has overwritten the *gp* register (e.g., S1 in Figure 10), the user-space signal handler may observe an incorrect *gp* value and thus fail to execute the expected semantics. To maintain compatibility with existing signal-handling mechanisms, Chimera restores *gp* to its original value ("Restoring gp" in Figure 10) after the kernel stores context for signal delivery. This guarantees that the signal handler observes the correct *gp* and executes correctly.

5 Analysis

5.1 Analysis of Chimera's Correctness Guarantee

Claim 1. *In Chimera, any erroneous execution that occurs in the rewritten binary is guaranteed to trigger a deterministic fault, which can be always detected at runtime.*

Reasoning. After trampoline placement, all non-trampoline instructions remain identical to those in the original binary, ensuring correct execution for control flows that do not encounter trampolines. Control flows executing trampolines fall into two categories: (1) Completely executing the trampoline and jumping to the corresponding target instructions can preserve correctness without triggering faults. (2) Partially executing a trampoline due to an unexpected indirect jump will lead to an erroneous execution. In Chimera's *SMILE* trampolines (§4.2), all partially executed trampolines are designed to induce immediate and deterministic faults (either an illegal instruction fault or a segmentation fault) without executing any unintended instructions. Therefore, any erroneous executions in Chimera trigger deterministic faults. □

Claim 2. *Chimera's runtime fault handling mechanism can correctly recover execution from any deterministic fault.*

Reasoning. When a fault's address and its execution context are correctly identified, this fault can be solved if the fault handling mechanism can recover the execution context to the correct address. Chimera achieves this by maintaining a per-rewritten-binary fault-handling table for correcting erroneous execution contexts. When overwriting a neighbor instruction (whose address is i), CHBP must have copied i to a new address i' in target instructions (§4.2). Since a potential erroneous jump can originally target i, CHBP records i and the new i' as a key-value pair into a table (§4.3), which then acts as a runtime read-only data structure. Chimera corrects any erroneous executions according to this table. □

Assertion 1 (Correctness of *Chimera*)**.** *Chimera guarantees the correctness of all rewritten binaries by ensuring that any erroneous execution triggers a deterministic fault, which is detected and recovered at runtime.*

Reasoning. By Lemma 1, any erroneous execution triggers a deterministic fault. By Lemma 2, any deterministic fault can be recovered by Chimera's runtime fault handling mechanism. Therefore, Chimera guarantees that each rewritten binary maintains the same semantics as the original binary. □

5.2 Analysis of Chimera's Performance Guarantee

Assertion 2 (High Performance of *Chimera*)**.** *In normal executions, Chimera's fault-handling incurs only the overhead of executing SMILE trampolines.*

Reasoning. Normal executions divide into: (1) execution flows without executing trampolines, and (2) execution flows that execute complete trampolines. The first type incurs no overhead. The second type incurs only the overhead of executing our *SMILE* trampoline instead of the expensive *trap-based trampolines* used in prior binary patching methods, with no additional cost for *proactive fault checking* (high invoking frequency, see §6.2) for all indirect jumps as existing binary regeneration methods (§2.2). □

6 Evaluation

Setup. We deployed Chimera on two devices: a Banana Pi BPI-F3 board [18] for general tests and a SOPHGO SG2042 board [47] for scalability tests (only §6.4). The Banana Pi BPI-F3 board contains a SpacemiT K1 8-core RISC-V 1.6GHz CPU [11], supporting RV64GCV ISA with RVV v1.0 and 256-bit vector registers. The board has 16GB RAM and a

128GB SD card. SOPHGO SG2042 has a 64-core 2.0GHz CPU, equipped with 128GB DDR4 DRAM and a 128GB SSD.

Our heterogeneous computing evaluation involved two types of cores: base cores supporting the RV64GC ISA, and extension cores supporting the RV64GCV ISA, which additionally included RISC-V Vector (RVV) extensions. We chose them because the RV64GC ISA is the most widely supported ISA, and the RVV extension is the most used optional RISC-V extension, offering significant performance acceleration [10–12]. As the cores in each board are homogeneous, to simulate ISAX heterogeneous processor, we disabled the vector extension on four cores (base cores) on Banana Pi by clearing the bit in the RISC-V CSR *misa*, while retaining it on the other four cores (extension cores). Similarly, for SG2042, we disabled the vector extension on 32 cores and kept it on the others. All experiments without specific mention of SG2042 were conducted by default on Banana Pi.

Baselines. We compared `Chimera` with two kinds of baselines: (1) heterogeneous computing baselines, and (2) binary rewriting baselines, since the performance of rewritten binaries generated by `CHBP` determines the overall performance of `Chimera`. We compared the heterogeneous computing performance of `Chimera` with the following baselines: (1) Fault and migrate (FAM [39]). Binaries with extension instructions could only run on extension cores. This baseline revealed the performance of a heterogeneous computing system without flexible scheduling (§2.1). (2) The SOTA compilation-based heterogeneous computing system MELF [59]. It compiled the source code into two versions of binaries, a base ISA and an extension ISA version, enabling them to run on both base and extension cores. As `Chimera` rewrote the binaries without the source code, MELF revealed the ideal performance of `Chimera`. (3) We adapted Safer [49], which is the binary rewriting method with the best performance, to our environment to compare the performance of `Chimera` with that of SOTA binary rewriting methods(§2.2).

We compared the performance of rewritten binaries generated by `CHBP` with the following baselines: (1) native compilation, representing the performance of binary rewriting under ideal conditions; (2) the SOTA binary patching method, ARMore [27], because ARMore does not support jumping over 1MB, so we use trap-based trampolines in such cases; (3) the SOTA binary regeneration method, Safer [49]; (4) a strawman binary patching method that utilized trap-based trampolines to jump over 1MB. We used this method to evaluate the performance improvements by replacing trap-based trampolines with the *SMILE* trampolines.

Workloads. We evaluated the performance and correctness of `Chimera` and baselines under three representative workloads: (1) A widely used heterogeneous workload suite [33, 59], consisting of matrix and integer tasks with varying proportions (e.g., 40% matrix and 60% integer, see §6.1). (2) SPEC CPU2017 [2], a standard benchmark commonly used in prior binary rewriting studies [27, 46, 49]. (3) Real-world applications, such as Vim [57] and OpenBLAS [24].

We focus on the following questions:

§6.1: How efficient is `Chimera` in heterogeneous computing?
§6.2: How efficient is `CHBP`'s binary rewriting methods?
§6.3: Can `CHBP` guarantee correctness on real applications?
§6.4: Can `Chimera` achieve high performance on real applications?

6.1 Heterogeneous Computing Performance

We evaluated the heterogeneous computing performance of `Chimera` and the baselines on Banana Pi. Our workload comprised two types of tasks: (1) Base tasks, comprised of Fibonacci sequence calculations, which could not be accelerated by RVV extension, had the same performance across all cores. (2) Extension tasks, comprised of matrix multiplication, which could be accelerated by RVV extension, had different performance across different cores. The workload had 1000 mixed tasks. The computation times of different tasks were in the ratio 2:2:2:1 for: (1) a base task on a base core, (2) a base task on an extension core, (3) an extension task on a base core, and (4) an extension task on an extension core. By varying the proportion of tasks with RVV extension instructions (from 0% to 100%), we systematically assessed the acceleration capabilities of both `Chimera` and the baselines under different loads.

We compiled this workload into both base (use only RV64GC instructions) and extension versions (with RVV extension instructions enabled). These versions were the input for `Chimera` and baselines, allowing us to evaluate the performance of *downgrading* and *upgrading* (§3.4).

We used work-stealing, a widely used load-balancing policy across multiple cores [45], as our scheduling policy. It comprised two thread pools: a base core thread pool and an extension core thread pool, each with four workers. Workers with empty task queues could steal tasks from others in the same pool. Tasks with extension instructions were first allocated to the extension core pool, while tasks without extension instructions were allocated to the base core pool. If all workers in a thread pool were idle, they steal tasks from the other pool and execute the corresponding rewritten binary. In the FAM baseline, base core workers migrated the stolen task back to the extension core workers when meeting an unsupported extension instruction.

Result. The accumulated CPU time and end-to-end latency for execution are shown in Figure 11.

For MELF, Safer, and `Chimera`, because extension tasks executed faster, the latency in both *downgrading* and *upgrading* decreased as the proportion of extension tasks increased.

For FAM, in *downgrading*, the latency initially decreased because extension tasks ran faster. However, as the proportion of extension tasks increased and extension instructions could only be executed on the extension core, the base core would idle after completing its base tasks; this resulted in suboptimal utilization of computational resources, and conse-

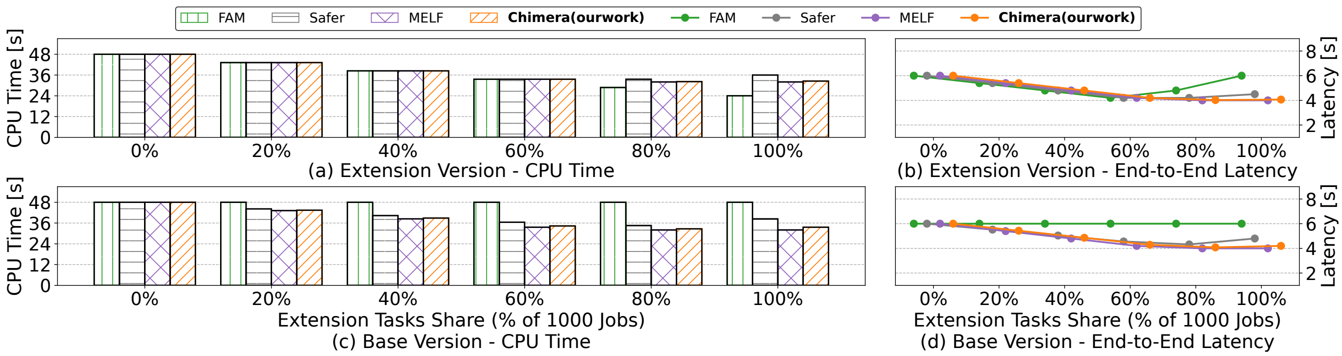

Figure 11. The CPU time and end-to-end latency of `Chimera` and baselines on an 8-core ISAX heterogeneous processor. For MELF, Safer, and `Chimera`, latency decreased in both *downgrading* (Figure 11b) and *upgrading* (Figure 11d) as the proportion of faster extension tasks increased. For FAM, in *downgrading* (Figure 11b), latency first decreased but then increased because base cores will idle as the proportion of extension tasks increased; in *upgrading* (Figure 11d), FAM provided no vector acceleration (an extension task without vector acceleration had the same execution time as a base task), so latency stayed essentially unchanged.

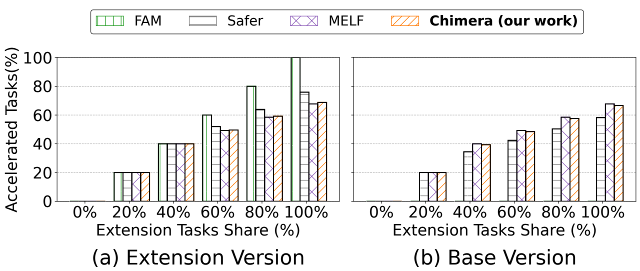

Figure 12. The proportion of extension tasks accelerated by vector extension.

quently, latency gradually rose. In *upgrading*, FAM could not accelerate tasks using the vector extension (an extension task without acceleration has the same execution time as a base task), so overall latency remained essentially unchanged.

Compared to MELF, the performance overhead of `Chimera` in accumulated CPU time and end-to-end latency was about 3.2% in *downgrading* and 5.3% in *upgrading*. These results demonstrated that `Chimera` achieved almost the same performance as compilation-based methods.

Compared to Safer, `Chimera` reduced computation time by 10.1% and end-to-end latency by 12.5% on average in *downgrading* and *upgrading*. Unlike Safer's proactive fault checks on all indirect jumps, `Chimera`'s passive fault handling avoids overhead in normal executions.

MELF, Safer, and `Chimera` achieved up to 33.1% lower end-to-end latency with up to 50.0% longer CPU time than FAM, because these systems can fully utilize idle base cores, which results in more CPU time.

Breakdown. Figure 12 shows the proportion of extension tasks accelerated by vector extension. The results indicate that in heterogeneous systems such as MELF and `Chimera`, approximately 30–40% of tasks containing extension instructions can be offloaded to base cores when extension tasks comprise 100% of the workload, explaining why heterogeneous computing systems have lower end-to-end latency.

Overall, `Chimera` achieves high heterogeneous computing performance compared to the native-compilation-based methods, demonstrating the efficiency of `Chimera` in ISAX heterogeneous computing.

6.2 Binary Rewriting Efficiency

We compared `Chimera`'s binary rewriting performance with the baselines on SPEC CPU2017 [2]. We compiled SPEC CPU2017 with the *-O3* optimization flag, RVV auto-vectorization enabled [14], and compressed instruction support. We selected benchmarks with code sections larger than 1MB (§2.2), as smaller code sections (within ±1 MB) fall within the jump range of single-instruction trampolines and do not require long-distance trampolines (§1).

For fairness, we adopted the commonly used empty patching method in binary rewriting evaluations [27, 46, 49]. Specifically, `CHBP` and baselines directly rewrote source instructions (RVV extension instructions) into target instructions which replicated these extension instructions. This method ensured that the performance overhead only originated from binary rewriting. Then we compared the performance of the rewritten binary by `CHBP` and baselines.

Result. Figure 13 shows the performance degradation of `CHBP` and baselines compared to the original binary. The results showed that compared to the original binary, `CHBP` achieved a 5.3% performance degradation on average, while Safer experienced a 15.6% degradation on average. The worst performance degradation of `CHBP` was 9.6%, but the worst performance degradation of Safer was 42.5%. This is because `CHBP` does not affect normal executions, whereas Safer impacts normal execution by requiring checks on all indirect jumps (§2.2) and lacks support for transparent migration.

ARMore incurred a 171.5% performance degradation as all its trampolines are trap-based. Compared to strawman binary patching, `CHBP` improved performance by 60.2%, highlighting the importance of using the *SMILE* trampoline.

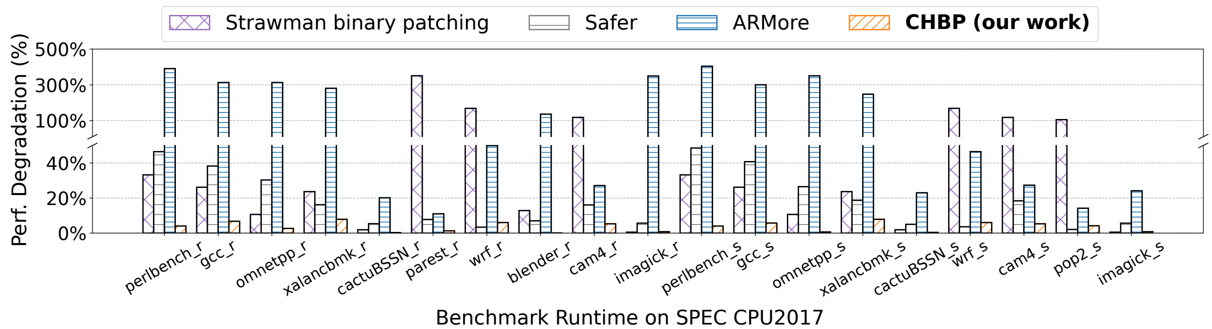

Figure 13. The performance comparison between our work and the SOTA on SPEC CPU2017. The results demonstrated that CHBP had the best performance among all baselines, showing the effectiveness of our passive fault handling mechanism.

Table 2. The count of correctness guarantee mechanism triggering of CHBP and baselines. CHBP triggered the correctness guarantee mechanism the least.

	Fault Handling Trigger Count (10^9)			
	CHBP	Safer	ARMore	Strawman
Real-world Application				
Git	1.4×10^{-7}	0.23	0.23	0.011
Vim	6.9×10^{-7}	0.18	0.18	1.9×10^{-4}
GIMP	2.7×10^{-6}	0.44	0.32	0.44
CMake	9.7×10^{-6}	4.12	4.12	1.74
CTest	7.4×10^{-6}	3.98	3.98	2.16
Python	4.5×10^{-6}	0.82	0.82	0.021
Libopenblas	2.4×10^{-6}	4.1	4.1	1.2
SPEC CPU2017				
cactuBSSN_r	2.5×10^{-7}	6.0×10^{-3}	6.0×10^{-3}	3.0×10^{-4}
cactuBSSN_s	2.7×10^{-7}	5.3×10^{-3}	5.3×10^{-3}	2.0×10^{-4}
cam4_r	1.3×10^{-5}	1.02	1.07	10.66
cam4_s	4.5×10^{-4}	4.51	4.57	40.21
gcc_r	4.2×10^{-4}	16.87	16.87	0.77
gcc_s	7.3×10^{-4}	35.55	35.57	1.124
xalancbmk_r	9.1×10^{-4}	13.12	13.15	0.92
xalancbmk_s	9.2×10^{-4}	13.12	13.15	0.88
imagick_r	3.3×10^{-4}	16.07	16.10	0.57
imagick_s	1.4×10^{-4}	5.34	5.51	0.36
omnetpp_r	3.9×10^{-4}	23.29	23.29	1.26
omnetpp_s	3.9×10^{-4}	23.29	23.34	1.34
perlbench_r	1.7×10^{-3}	65.66	65.56	6.74
perlbench_s	1.7×10^{-3}	65.23	64.56	6.74
pop2_s	7.0×10^{-5}	2.10	2.17	20.16
wrf_r	1.5×10^{-5}	1.12	1.11	5.11
wrf_s	8.4×10^{-4}	6.31	6.21	30.35
blender_r	3.2×10^{-5}	3.87	3.90	0.124

Breakdown. Table 2 shows the number of times additional runtime mechanisms were triggered on benchmarks. For our CHBP, it referred to the number of deterministic faults handling. For strawman binary patching and ARMore, it referred to the number of traps. For Safer, it referred to the number of pointer checks and corrections.

Chimera triggers the fewest fault-handling events—only 0.005% on average of those in the baselines. This is because prior works proactively trigger fault handling on all indirect jumps, even in normal executions, resulting in frequent runtime fault-handling events. In contrast, Chimera adopts a passive strategy that invokes fault handling only in erroneous executions, which are rare in rewritten binaries. This leads to fewer fault-handling events and better performance. Overall, CHBP achieves better performance than existing binary rewriting methods, especially in complex binaries that are pervasive in heterogeneous computing.

Table 3. The code section size, the percentage of extension instructions in the binary, the number of exit trampolines, and cases where the dead register could not be found (**our method (§4.2)/traditional register liveness analysis** [31, 46]) of Chimera's CHBP in real applications and SPEC CPU2017.

	Code Size (MB)	Ext. Inst.	Trampoline Num.	Dead Reg. Not Found
Real-world Application				
Git	3.11	2.7%	3270	21/993
Vim	2.91	2.31%	2915	30/1308
CMake	7.60	3.32%	28128	78/9213
CTest	8.50	3.30%	30990	20/1129
Python	2.31	1.77%	4311	54/1482
Libopenblas	6.72	0.59%	3305	15/628
SPEC CPU2017				
cactuBSSN_r	3.49	3.24%	13281	112/6024
cactuBSSN_s	3.49	3.24%	13293	112/6024
cam4_r	4.29	3.37%	17086	301/7846
cam4_s	4.47	3.27%	17449	401/7846
gcc_r	6.88	0.44%	5482	89/2080
gcc_s	6.88	0.44%	5482	89/2080
xalancbmk_r	2.91	1.36%	8798	107/3923
xalancbmk_s	2.91	1.36%	8798	107/3923
imagick_r	1.41	1.63%	2055	70/860
imagick_s	1.46	1.47%	2136	65/867
omnetpp_r	1.14	0.95%	2688	23/860
omnetpp_s	1.14	0.95%	2688	21/867
perlbench_r	1.52	0.58%	1521	12/583
perlbench_s	1.52	0.58%	1521	12/583
pop2_s	3.57	3.71%	15560	132/7722
wrf_r	16.79	3.21%	41408	103/11121
wrf_s	16.78	3.20%	41468	112/11098
blender_r	7.31	1.51%	15085	154/5395

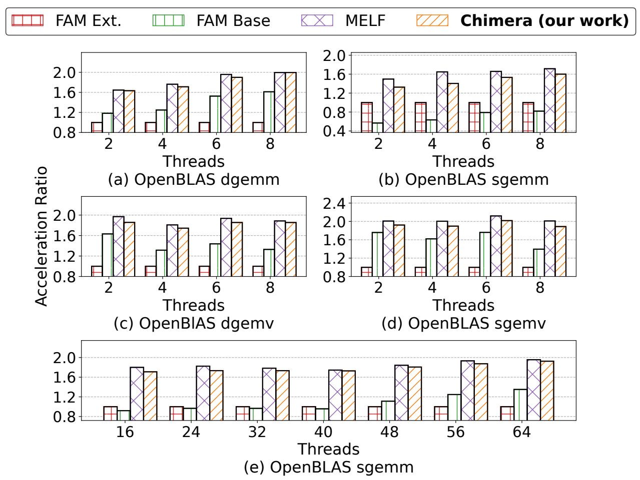

Figure 14. Comparison of performance results for real-world applications. Figure 14(a-d) is a comparison of OpenBLAS. Figure 14e is the scalability evaluation of Chimera and baselines. FAM Ext. is the baseline running extension workloads on FAM and FAM Base runs base workloads. All acceleration ratios are relative to FAM Ext.

6.3 Correctness of Chimera

We evaluated the correctness of Chimera using SPEC CPU and real-world applications (Real-world Application in Table 2 and 3) with large code sections compiled with the RVV extension. These binaries were translated to the base ISA and executed with their respective test suites.

Results. The results show that our method correctly translated all binary files and passed all test suites, proving that our work can guarantee correctness.

Breakdown. In Table 3, extension instructions are a small part of the workload, so most instructions retain their original performance after binary rewriting. This supports our binary patching method. We also evaluated CHBP's ability to identify dead registers. The results show that traditional register liveness analysis [46] failed to find dead registers in about 35.9% of cases. In contrast, our *exit position* shifting method (§4.3) efficiently reduced this failure rate to just 1.1% (98.9% trampoline can find a dead register). Overall, the results show that Chimera could guarantee the correctness of binary rewriting.

6.4 Real-World Applications

To evaluate Chimera's performance in real-world applications, we used openBLAS, a widely utilized library that can be accelerated with vector extensions. We selected four representative matrix computations: dgemm, sgemm, dgemv, and sgemv, and used Chimera to rewrite the vector versions of these workloads. These workloads will be confined to a fixed set of cores based on the number of threads; for example, an 8-thread workload will be limited to 4 base cores and 4 extension cores. To further evaluate scalability, we evaluated sgemm on SG2042 under varying thread counts, as its performance is sensitive to multi-threading.

Result. As shown in Figure 14, FAM Ext. is the baseline running extension workloads on FAM and FAM Base runs base workloads. The two baselines could also represent, respectively, the performance on ISAX hardware of (1) a binary that contains extension instructions which are executed solely on the extension cores and (2) a binary containing only base instructions. The results show that the performance gap between Chimera and MELF was only 5.4%, and the acceleration ratio compared to FAM Base was 32.1%. This indicates the efficiency of Chimera in real-world applications.

Additionally, because FAM Ext. could use only the extension cores for vector computations, the base cores would idle; as the number of threads increases, this also led to contention for the extension cores. Consequently, in most cases, FAM Ext. performs worse than FAM Base (dgemm, dgemv and sgemv). As the number of threads increases, matrix-to-matrix workloads (dgemm, sgemm) experienced a decrease in overall speedup due to the growing thread synchronization overhead. This was particularly evident in the scalability experiments, where the speedup dropped by 60.2% when increasing from 16 to 64 threads. Conversely, matrix-to-vector workloads (dgemv, sgemv) benefited from effective parallelization, resulting in a stable and increasing overall speedup. Additionally, as the thread count rises, synchronization between threads becomes the primary performance bottleneck, which narrows the performance gap between our method and MELF in multi-threaded scenarios.

Overall, Chimera achieves similar performance to compilation-based heterogeneous computing systems on real-world applications. The small performance gap arises mainly from the lower quality of instructions produced by binary translation compared to native compilation and the trampolines, which can be addressed by integrating more advanced translation tools. These results demonstrate the high performance potential of Chimera.

7 Conclusion

We present Chimera, a novel ISAX heterogeneous computing system achieving transparency, high performance, and correctness via binary rewriting. Chimera passively confines error handling to rare erroneous executions, enabling seamless task scheduling across heterogeneous cores with negligible overhead. Evaluation on real-world applications (e.g., SPEC CPU2017, OpenBLAS) shows that Chimera retains program correctness while delivering near-native performance.

Acknowledgments

We would like to thank our shepherd Wolfgang Schröder-Preikschat and the anonymous reviewers for their insightful comments. This work is supported by National Key R&D Program of China (No.2023YFB4503902).

References

[1] 2013. Documentation 2013; Arm Developer — developer.arm.com. https://developer.arm.com/documentation/ddi0210/latest/CACBCAAE. Accessed: 2025-03-25.

[2] 2017. SPEC CPU 2017 — spec.org. https://www.spec.org/cpu2017/. Accessed: 2025-03-25.

[3] 2023. IDA Pro. https://hex-rays.com/ida-pro. Accessed: 2023-10-21.

[4] 2024. Debian – The Universal Operating System — debian.org. https://www.debian.org/. Accessed: 2025-03-25.

[5] 2024. GitHub - riscv/riscv-isa-manual: RISC-V Instruction Set Manual — github.com. https://github.com/riscv/riscv-isa-manual/. Accessed: 2025-03-24.

[6] 2024. Home — opencv.org. https://opencv.org/. Accessed: 2025-05-06.

[7] 2024. QEMU — qemu.org. https://www.qemu.org/. Accessed: 2025-05-06.

[8] 2024. RISC-V International — riscv.org. https://riscv.org/. Accessed: 2025-03-24.

[9] 2024. riscv-v-spec/v-spec.adoc at master · riscvarchive/riscv-v-spec — github.com. https://github.com/riscvarchive/riscv-v-spec/blob/master/v-spec.adoc. Accessed: 2025-03-25.

[10] 2024. SiFive - Leading the RISC-V Revolution — sifive.com. https://www.sifive.com/. Accessed: 2025-05-06.

[11] 2024. SpacemiT - RISC-V SoC - SPACEMIT — spacemit.com. https://www.spacemit.com/en/. Accessed: 2025-05-06.

[12] 2024. XuanTie — xrvm.com. https://www.xrvm.com/product/xuantie/. Accessed: 2025-05-06.

[13] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. 2020. BinRec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[14] AMD. 2023. RISC-V Vector Extension. https://rocm.docs.amd.com/projects/llvm-project/en/latest/LLVM/llvm/html/RISCV/RISCVVectorExtension.html. Accessed: 2023-10-05.

[15] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An {In-Depth} analysis of disassembly on {Full-Scale} x86/x64 binaries. In *25th USENIX security symposium (USENIX security 16)*. 583–600.

[16] The Linux Kernel Authors. 2023. *Uprobe Tracer*. https://docs.kernel.org/trace/uprobetracer.html Accessed: 2025-05-09.

[17] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M Nguyen, Yaosheng Fu, Florian Zaruba, et al. 2020. BYOC: a" bring your own core" framework for heterogeneous-ISA research. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 699–714.

[18] Banana Pi. 2023. Banana Pi: An Open Source Hardware Development Platform. https://banana-pi.org/. Accessed: 2023-10-20.

[19] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the boundaries in heterogeneous-ISA datacenters. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 645–659.

[20] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[21] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.

[22] Bryan Buck and Jeffrey K Hollingsworth. 2000. An API for runtime code patching. *The International Journal of High Performance Computing Applications* 14, 4 (2000), 317–329.

[23] Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2020. Flick: Fast and lightweight isa-crossing call for heterogeneous-ISA environments. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 187–198.

[24] OpenBLAS Contributors. 2023. OpenBLAS: An optimized BLAS library. http://www.openmathlib.org/OpenBLAS/. Accessed: 2023-10-13.

[25] Chinmay Deshpande, Fabian Parzefall, Felicitas Hetzelt, and Michael Franz. 2024. Polynima: Practical hybrid recompilation for multithreaded binaries. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1126–1141.

[26] Matthew DeVuyst, Ashish Venkat, and Dean M Tullsen. 2012. Execution migration in a heterogeneous-ISA chip multiprocessor. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 261–272.

[27] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. 2023. {ARMore}: Pushing Love Back Into Binaries. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6311–6328.

[28] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[29] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 151–163.

[30] Gregory J Duck, Yuntong Zhang, and Roland HC Yap. 2022. Hardening binaries against more memory errors. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 117–131.

[31] Dyninst Project. 2025. Dyninst: Binary Instrumentation and Analysis Framework. https://github.com/dyninst/dyninst. GitHub repository, accessed 2025-09-16.

[32] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. 1075–1092.

[33] Giorgis Georgakoudis, Dimitrios S Nikolopoulos, Hans Vandierendonck, and Spyros Lalis. 2014. Fast dynamic binary rewriting for flexible thread migration on shared-isa heterogeneous mpsocs. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. IEEE, 156–163.

[34] Git Development Team. 2023. Git - Distributed Version Control System. https://git-scm.com/. Online; accessed: 2023-05-06.

[35] Haegeon Jeong, Jeanseong Baik, and Kyungtae Kang. 2017. Functional level hot-patching platform for executable and linkable format binaries. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 489–494.

[36] Hyungseok Kim, Soomin Kim, and Sang Kil Cha. 2025. Towards Sound Reassembly of Modern x86-64 Binaries. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1317–1333.

[37] Hyungseok Kim, Soomin Kim, Junoh Lee, Kangkook Jee, and Sang Kil Cha. 2023. Reassembly is Hard: A Reflection on Challenges and Strategies. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1469–1486.

[38] Akash Kothari, Abdul Rafae Noor, Muchen Xu, Hassam Uddin, Dhruv Baronia, Stefanos Baziotis, Vikram Adve, Charith Mendis, and Sudipta Sengupta. 2024. Hydride: A Retargetable and Extensible Synthesis-based Compiler for Modern Hardware Architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 514–529.

[39] Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[40] Linux Kernel Development Community. 2023. *Linux Kernel Sched-*

uler Documentation. https://docs.kernel.org/scheduler/index.html Accessed: 2023-10-07.

[41] Linux Kernel Development Community. 2024. *Process Address Space*. https://docs.kernel.org/mm/process_addrs.html Accessed: 2024-03-09.

[42] Linux Kernel Labs. 2024. *Memory Mapping*. https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html Accessed: 2024-03-09.

[43] Linux man-pages project. 2024. *signal(2) - overview of signals*. https://man7.org/linux/man-pages/man2/signal.2.html Accessed: 2024-03-09.

[44] Arm Ltd. 2024. big.LITTLE: Balancing Power Efficiency and Performance — arm.com. https://www.arm.com/technologies/big-little. Accessed: 2025-03-25.

[45] Sarah Mcclure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient scheduling policies for {Microsecond-Scale} tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1–18.

[46] Xiaozhu Meng and Weijie Liu. 2021. Incremental CFG patching for binary rewriting. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1020–1033.

[47] Milk-V. 2023. Pioneer Getting Started Guide. https://milkv.io/docs/pioneer/getting-started/processor. Accessed: 2023-10-20.

[48] Julian Oppermann, Brindusa Mihaela Damian-Kosterhon, Florian Meisel, Tammo Mürmann, Eyck Jentzsch, and Andreas Koch. 2024. Longnail: High-Level Synthesis of Portable Custom Instruction Set Extensions for RISC-V Processors from Descriptions in the Open-Source CoreDSL Language. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 591–606.

[49] Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R Sekar. 2023. {SAFER}: Efficient and {Error-Tolerant} Binary Instrumentation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1451–1468.

[50] Shipei Qu, Xiaolin Zhang, Chi Zhang, and Dawu Gu. 2024. Trapped by Your WORDs:(Ab) using Processor Exception for Generic Binary Instrumentation on Bare-metal Embedded Devices. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[51] Ashwin Ramaswamy, Sergey Bratus, Sean W Smith, and Michael E Locasto. 2010. Katana: A hot patching framework for elf executables. In *2010 International Conference on Availability, Reliability and Security*. IEEE, 507–512.

[52] RISC-V International. 2023. *RISC-V Technical Specifications*. https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications#Profiles

[53] riscv-non-isa. 2024. GitHub - riscv-non-isa/riscv-elf-psabi-doc: A RISC-V ELF psABI Document. https://github.com/riscv-non-isa/riscv-elf-psabi-doc. Accessed: 2025-03-24.

[54] Florian Rommel, Lennart Glauer, Christian Dietrich, and Daniel Lohmann. 2019. Wait-free code patching of multi-threaded processes. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. 23–29.

[55] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2028. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[56] The LLVM Project. [n. d.]. *Extending LLVM*. https://releases.llvm.org/3.8.0/docs/ExtendingLLVM.html Release 3.8.0.

[57] The Vim Project. 2023. Vim - Official Website. https://www.vim.org/. Online; accessed: 2023-05-06.

[58] Linan Tian, Yangyang Shi, Liwei Chen, Yanqi Yang, and Gang Shi. 2022. Gadgets splicing: dynamic binary transformation for precise rewriting. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 155–167.

[59] Dominik Töllner, Christian Dietrich, Illia Ostapyshyn, Florian Rommel, and Daniel Lohmann. 2023. {MELF}: Multivariant Executables for a Heterogeneous World. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 257–273.

[60] Luca Valente, Yvan Tortorella, Mattia Sinigaglia, Giuseppe Tagliavini, Alessandro Capotondi, Luca Benini, and Davide Rossi. 2023. HULK-V: A heterogeneous ultra-low-power Linux capable RISC-V SoC. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[61] Ashish Venkat, Harsha Basavaraj, and Dean M Tullsen. 2019. Composite-ISA cores: Enabling multi-ISA heterogeneity using a single ISA. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 42–55.

[62] Ashish Venkat and Dean M Tullsen. 2014. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 121–132.

[63] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 1–37.

[64] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–147.

[65] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 659–676.