# Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation

Alexis Engelke
Martin Schulz
Technical University of Munich
{engelke,schulzm}@in.tum.de

## Abstract

Dynamic binary instrumentation frameworks are popular tools to enhance programs with additional analysis, debugging, or profiling facilities or to add optimizations or translations without requiring recompilation or access to source code. They analyze the binary code, translate into a—typically low-level—intermediate representation, add the needed instrumentation or transformation and then generate new code on-demand and at run-time. Most tools thereby focus on a fast code rewriting process at the cost of lower quality code, leading to a significant slowdown in the instrumented code. Further, most tools run in the application's address space, making their development cumbersome.

We propose a novel dynamic binary instrumentation framework, *Instrew*, which closes these gaps by (a) leveraging the LLVM compiler infrastructure for high-quality code optimization and generation and (b) enables process isolation between the target code and the instrumenter. Instead of using our own non-portable and low-level intermediate representation, our framework directly lifts the original machine code into LLVM-IR, where instrumentation and behavioral changes may be performed, and from which high quality code can be produced. Results on the SPEC CPU2017 benchmarks show that the rewriting overhead is only 1/5 of the overhead incurred using the state-of-the-art toolchain Valgrind.

***CCS Concepts*** • **Software and its engineering → Runtime environments**; *Just-in-time compilers*; Software testing and debugging;

***Keywords*** Dynamic Binary Instrumentation, Dynamic Binary Translation, LLVM, Client/Server Model, Optimization

*VEE '20, March 17, 2020, Lausanne, Switzerland*

**ACM Reference Format:**
Alexis Engelke and Martin Schulz. 2020. Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3381052.3381319

## 1 Introduction

Tools for rewriting binaries dynamically at run-time are frequently used for program analysis, debugging and portability. During rewriting, such tools may also add additional *instrumentation* code to perform their analysis, code specialization or even code translation. The advantage over adding such transformations at compile-time lies in the fact that instrumentation is possible on the fly and without recompiling the program or its libraries. Therefore, several tools for Dynamic Binary Instrumentation (DBI) exist [11, 12, 25], most notably Valgrind [26] and Pin [24].

Many of these frameworks focus on low overhead during the instrumentation and code generation process. As such, they perform only lightweight optimizations at the cost of lower performance of the newly generated code. Additionally, instrumentation typically happens at the level of basic blocks or superblocks, limiting the scope of possible optimizations even further. As a consequence, it is not unusual for instrumented programs to run several times slower than non-instrumented code.

The lack of optimizations can be avoided by incorporating a high-quality optimizer and machine code generator. In fact, DBILL [25] showed that using the LLVM [23] compiler infrastructure for code generation can achieve a higher performance. However, as the machine code is lifted first to TCG (the IR of QEMU [10]), it faces several limitations: (1) only common integer instructions can be mapped to the LLVM-IR, causing a high overhead for other instructions, including floating-point and SIMD instructions; (2) the code is instrumented with basic block granularity, limiting the scope of possible optimizations; and (3) code for emulating the original behavior and code for managing the emulation are mixed and can only be distinguished using metadata annotations, which are known to cause performance problems [18] and increase complexity for tool writers.

Another limitation of existing binary instrumentation frameworks is that the instrumenting tool is required to use the same address space as the executed binary. While this eases access to the code to be instrumented, as it resides within the same address space, tool developers have to be careful to not corrupt program state, often limiting the use of external libraries. To our knowledge, the only exception is DynInst [11], where the tool using the DynInst API runs in a separate process and modifies the instrumented process using the `ptrace` debugging interface.

In this paper, we describe a novel and comprehensive framework that overcomes these shortcomings by leveraging the optimization and transformation capabilities of LLVM for dynamic binary rewriting. In particular, we present a library to lift x86-64 machine code directly to LLVM-IR covering most x86-64 instructions (include SSE/SSE2, but excluding the rarely used x87 FPU/MMX and AVX) and capable of lifting entire functions for efficient code generation. Our rewriting framework is based on this library and implements a client-server model for rewriting, where the server performs the instrumentation and code generation, while the client is only dispatching and executing the generated code. Leveraging LLVM-IR opens the door to high-quality code instrumentation and optimization, as well as—in the extreme case—binary translation to other platforms by changing the back-end used. Results on the SPEC CPU2017 benchmarks show that our average rewriting overhead is only 72%, i.e., only around 1/5 compared to Valgrind's average overhead of 367%.

Our main contributions are the following:

- An x86-64-to-LLVM lifter that supports function-level lifting and is suitable for generation of efficient code that is able to reach a performance close to the original code.
- A retargetable rewriting and instrumentation framework based on LLVM, capable of generating high quality code for high run-time performance.
- A client-server approach for dynamic binary instrumentation allowing the (expensive) rewriting and code generation to be done on different machines and the result to be cached for further use.

The rest of this paper is structured as follows: in Section 2 we outline use cases of dynamic binary instrumentation and briefly describe the general structure of binary instrumentation frameworks. In Section 3 we detail our approach to lift x86-64 machine code to LLVM-IR and in Section 4 we describe the architecture of our rewriting framework as well as some further details regarding the effective usage of LLVM for dynamic binary rewriting. In Sections 5 and 6, we will evaluate the performance of our framework compared to state-of-the-art tools. Finally, in Section 7 we cover related work and in Section 8 we conclude with a summary of our findings.

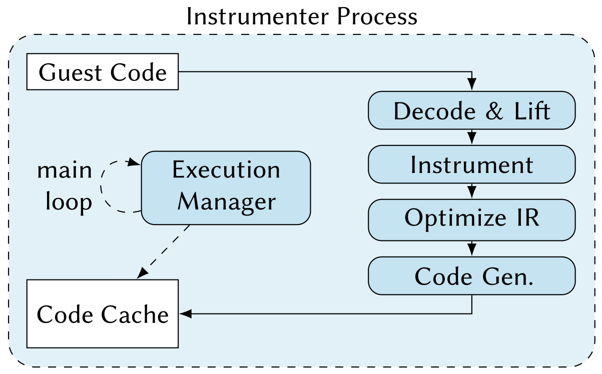


**Figure 1.** Architecture of a typical Dynamic Binary Instrumenter.

## 2 Background

The technique of dynamic binary instrumentation is widely used and has many applications, especially in debugging and performance analysis. For example, a memory checker, like Valgrind Memcheck [30] or Dr. Memory [13], can enhance the already compiled code with bounds checks and therefore detect invalid memory accesses at run-time. Another use case is the addition of tracing functionality to a program, enabling the extraction of information about program elements such as function calls or memory allocations. This information can then be used for performance analysis, e.g., for the detection of performance critical parts of a program or for the analysis of cache usage [33].

While such functionality could also be added at compile-time, which would allow for more optimizations and a higher performance of the instrumented program, this would require *all* source code to be re-compiled with these instrumentations, including all libraries. This not only takes a significant amount of time, but also is sometimes impossible, e.g., if the source of a library is not available.

The general approach of systems for dynamic binary instrumentation is the following (cf. Figure 1): in the beginning, the instrumenter (or host) loads the binary to be instrumented (target or guest) into memory, initializes its state and sets the guest instruction pointer to the entry address of the guest binary, before starting the host code's main execution loop. The host, starting from the current instruction pointer, decodes the guest code and lifts it into some kind of intermediate representation from where the program code can be modified and enhanced with the requested instrumentation. This results in a modified guest program—still represented in an IR—from where it generates new machine code, which is then stored in a separate location. The rewritten code is then executed and at the end sets the guest instruction pointer to the address of the next instruction, at which the main execution loop continues translating.

To reduce redundant decoding and instrumentation, rewriters typically deploy a *code cache*. If the execution loop encounters an address that is already translated and still is in cache, the code previously stored in cache is reused. As a further optimization, the translated code can be patched to directly branch to the following translated code block without going through the main loop of the host. This optimization is known as *chaining*. However, the target address must be known for chaining, for which reason this optimization cannot be applied to indirect branches and function returns.

## 3 Lifting x86-64 to LLVM-IR

Dynamic binary rewriting and instrumentation typically relies on lifting machine code to an Intermediate Representation (IR). This increases flexibility and simplifies the implementation of code transformations. Two prominent examples for this are VEX, the IR of the Valgrind instrumentation framework [26], and TCG, the IR used by QEMU [10, 28]. To reduce the overhead of both lifting code and generating new code from it, these IRs are typically low-level and incomplete, i.e., cannot represent the whole instruction set of an architecture. Instead, they rely on external helper functions for complex instructions not covered in the IR. Obviously, this limits optimization possibilities and has an impact on the quality and performance of the generated code.

We, therefore, pursue a different direction and target a higher-level and target architecture-independent IR. Instead of developing a new one, though, we leverage LLVM-IR [23], which not only fulfills our criteria, but also features a high-quality code optimizer and machine code generator. LLVM-IR is used by several compilers, including commercial ones (e.g., Arm Compiler for Linux [8]), and is becoming the defacto standard for IRs offering a rich eco-system of optimization passes.

A similar approach, the use of LLVM-IR, was used by DBILL [25]. However, DBILL first lifts machine code to TCG, its own IR, and from there to LLVM-IR. Instead, we generate LLVM-IR code *directly from the machine instructions*. This approach has several advantages:

- We can lift all instructions and have no restriction to instructions caused by the intermediate IR, like TCG. This especially applies to vector instructions, which are used in performance critical code parts.
- We can lift entire functions and do not have a limitation to basic blocks/superblocks. This enables a more comprehensive analysis of used data types, non-local optimizations and leads to less jumps between different code blocks.
- We can maintain a well-defined relation between the original registers/instructions and the corresponding values/instructions in the LLVM-IR and do not lose this information through the intermediate IR. This information is especially helpful for instrumentation use cases.

In order to use LLVM-IR in this context, our lifter must be able to derive "good" LLVM-IR code, i.e., code that is structured so that it can be handled well by the existing optimization/code generation infrastructure, as this is critical for *run-time performance*. Moreover, the IR code must also be designed in such a way that the optimization process itself does not add too much overhead. Furthermore, we require the LLVM-IR code generated by our lifter to be independent of the target architecture (disregarding pointer size), implying that x86-specific intrinsic functions or inline assembly cannot be used. This enables *retargetability*—the x86-64 machine code can then dynamically be retargeted and executed on other 64-bit architectures as well.

We implement our lifter, following these requirements, as a separate library, which we call *Rellume* and which is freely available and open-source[1], licensed under LGPLv2.1+.

### 3.1 Rellume Design

The overall lifting process consists of three main stages:

1. *Decoding:* first, we decode all instructions that belong to one block of compiled code, eventually also following jump targets, and split them into basic blocks.
2. *Instruction Lifting:* second, we create a skeleton LLVM-IR function and generate the corresponding LLVM-IR for each instruction.
3. *Branch Fixup:* finally, we add branches between the basic blocks to the LLVM-IR and fill the so-called PHI nodes, which merge incoming values from different basic block [5].

After these three stages, we can then perform any optimizations or other transformations leveraging the ecosystem of LLVM.

Our lifter works in a semi-lazy fashion: we first generate the corresponding LLVM-IR code for each instruction independently and then perform optimizations across instructions or basic block boundaries via LLVM optimization passes. As this may lead to a large number of unused or trivial instructions, we generate some parts of the LLVM-IR code only on demand. This improves overall performance as well as reduces analysis time and eliminates handling of dead code.

The library allows to create LLVM-IR code for a given set of instructions, with all branches fixed up. Code generation can be controlled through a configuration API, which also allows to to override the implementation of some instructions with custom implementations. This can be used for complex instructions, like `cpuid`, or the operating system dependent `syscall` instruction.

[1]https://github.com/aengelke/rellume, accessed 2020-02-17

### 3.2 LLVM-IR Functions

In LLVM-IR, all code must be enclosed in functions, which can take parameters and have a return value [5]. A function has a single entry point, but—apart from this—it can contain an arbitrary (but well defined) control flow defined on the level of multiple basic blocks.

The functions created by our lifter take a pointer to the *CPU state* as a single parameter and do not return any values. The CPU state structure contains all required information required to describe the state of the CPU needed to drive the host's main loop. This includes all general-purpose registers, the status flags and the SSE vector registers, as well as the instruction pointer `rip` and the segment register base offsets for the registers `fs`/`gs`.

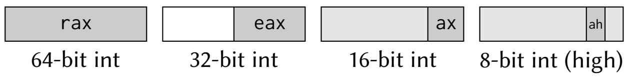
At the beginning of a function, the general-purpose, vector and flags registers are loaded into SSA registers in the LLVM-IR. Likewise, before the function returns, the new values of these registers are written back to memory if they have been modified. This avoids frequent memory accesses for register use within the actual program logic, reducing the workload for the optimizer. As a consequence, the CPU state is mainly used to pass state *between* different functions.

When lifting some x86-64 machine code to LLVM-IR, we start at a specified address, which we then use as entry into the function. Starting from the first instruction, the lifter generates the corresponding LLVM-IR and follows the control flow of the machine code, also creating appropriate branch instructions in the LLVM-IR where needed. When an indirect jump, call or return instruction is encountered, it is not possible to continue further. In such cases, the lifting process stops and the lifted function sets the virtual `rip` register in the CPU state to the address of the next instruction before returning. A separate lifting process will then start once the address is actually known.

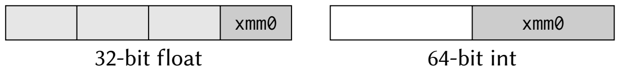
### 3.3 Basic Blocks

A basic block is a sequence of instructions that does not modify control flow and ends with a branch to another basic block or a return instruction. Basic blocks in the x86-64 machine code are detected during decoding. In the same phase, we also ensure that each instruction part of exactly one basic block. This reduces code size and avoids duplicate lifting and optimization work, improving overall performance.

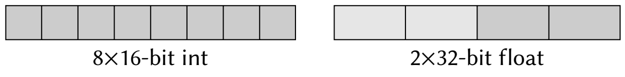
Per the design of the LLVM-IR, all code within a function must be split into basic blocks [5]. While in many cases one *architectural basic block* on x86-64 maps to a single basic block in the LLVM-IR, this is not always the case; e.g., the repeated string instructions have a different control flow logic. As an LLVM-IR basic block is not allowed to contain control flow instructions in the middle, lifting such instructions requires additional LLVM-IR basic blocks.



**(a)** Examples for single element facets (dark gray) of general-purpose registers. For a write-back of 32-bit facets, the upper half of the register is zeroed (white), while for 8-bit or 16-bit facets the untouched part (light gray) has to be preserved via bit masking.



**(b)** Examples for single element facets (dark gray) of SSE vector registers. On write-back, some instructions require keeping the untouched part (e.g., `addss`), while others clear to zero (e.g., `movq`).



**(c)** Examples for vector facets of SSE vector registers. Element type and count depend on the instruction, just as the handling of unused parts on write-back.

**Figure 2.** Registers can be accessed either as a single element or as a vector. The actual instruction defines the facet in which the register is accessed and may also specify whether untouched parts of the registers are preserved or zeroed.

### 3.4 Registers

In our lifter, we model the following x86-64 registers: 16×64-bit general-purpose registers, the 64-bit instruction pointer, 16×128-bit SSE vector registers, and the flags register. We currently do not support x87 FPU, MMX, and AVX registers, but this can be added later on.

Every register can be accessed in different views on the raw binary representation of the register contents, which we call *facets*. Facets can differ in their data type, e.g. `<2 x i64>` vs. `<4 x i32>`, but also in the part of the register they represent. For example, the general-purpose register `rax` can also be accessed as a 16-bit register (`ax`) or a high-byte register (`ah`).

Every register has a *native facet*, which is the bitwise representation of the register contents. In the LLVM-IR, this facet is represented using an integer of the appropriate length, e.g., an `i64` for a general-purpose register or an `i128` for an SSE vector register. The native facet must always be specified and all other facets can be deduced from it, although the values of other facets are cached after their computation.

A general-purpose register has additional integer facets for each accessible subregister (e.g., for `rax`, these are `eax`, `ax`, `al` and `ah`), cf. Figure 2a. Further, general-purpose registers have a special *pointer facet*, which uses a pointer type in the LLVM-IR. While integer and pointer arithmetic are identical in machine code, LLVM has a strong distinction between these types. It is beneficial to use the `getelementptr` instruction in

LLVM-IR for memory accesses and specific instructions [6]. This improves the quality of the alias analysis, which leads to better machine code.

Vector registers have a variety of other facets: scalar integer and floating-point facets with different sizes (cf. Figure 2b), as well as vector facets of different element counts for the scalar data types (cf. Figure 2c).

Register values are propagated between basic blocks using so-called PHI nodes, where the values of all preceeding blocks are specified [5]. There may be several PHI nodes for a single register, depending on the required facets. Whenever a specific facet is required from the predecessors, a new PHI node for this register–facet combination will be created.

***Read Access*** When an instruction reads a register, it accesses the register in a specific facet. For a general-purpose register, the facet is usually implied from the register itself (except for pointer facets), but for vector registers the facet depends on the instruction. For example, the instruction `addps` requires a `<4 x f32>` facet, whereas `paddd` requires a `<4 x i32>` facet.

The value for this facet is then either (a) already available from a previous instruction, (b) available from a newly created PHI node, or (c) generated from the native facet. In the third case, the value is extracted using `trunc`/`lshr` instructions for general-purpose registers and using `bitcast`, `shufflevector` and `extractelement` instructions for vector registers.

***Write Access*** On a write-back of a new value into a register, the facet is always implied by the register itself and the value type. In addition to caching the new value for the facet, the native facet must be updated as well. For general-purpose registers this may involve bitmasking for 8-bit and 16-bit facets or zero-extension for 32-bit facets. For vector registers, an updated native facet is created using proper `bitcast`/`shufflevector`/`insertelement` combinations.

## 3.5 Status Flags

The flags register has to be handled separately, as the individual flag bits are independent of each other. Therefore, the flags register has a separate 1-bit facet for each of the seven stored flags (sign, zero, carry, overflow, parity, adjust and direction). These flags are usually written and evaluated independently from each other and are rarely used in the format specified by the x86 architecture, which in fact is only possible using the `pushf`, `lahf`, and `syscall` instructions in user-mode [22]. To avoid frequent, but useless bit shuffling, code to compute the register value of `rflags` is only generated on demand. The set of the individual flag bits is then considered as the native facet for the flags register.

The four arithmetic flags are computed as specified by the architecture. For common operations, however, we optimized the evaluation of flags such that subsequent queries of conditions (e.g., in jumps) can be easily folded into a single LLVM `icmp` instruction during optimization. The computation of these flags is described in Table 1. For the overflow flag, our lifter also provides a mode to use intrinsic functions provided by LLVM, but this did not lead to noticeable performance changes.

**Table 1.** Computation of flags for common arithmetic operations. The expressions are designed to be easily foldable for all integer signed and unsigned comparison conditions. Parity and adjust flag are not optimized and use bitwise computations.

| | add | sub/cmp |
|---|---|---|
| ZF | $res = 0$ | $lhs = rhs$ |
| SF | $res <_s 0$ | $res <_s 0$ |
| CF | $res <_u lhs$ | $lhs <_u rhs$ |
| OF | bitwise | $(res <_s 0) \oplus (lhs <_s rhs)$ |

The parity flag is computed using the `llvm.ctpop` intrinsic [5], and the adjust flag is calculated using bitwise operations. Both flags are rarely used—we are not aware of *any* recent use of the adjust flag—and we, therefore, do not perform further optimizations.

## 3.6 Memory Access

Memory operands in x86-64 are rather complex and contain up to four optional components: (a) a base register, which may also be the instruction pointer `rip`; (b) a scaled index register, which is a register multiplied by 1, 2, 4, or 8; (c) a constant 32-bit offset; and (d) a segment register as additional offset. Additionally, in 64-bit mode the architecture allows for an address size override to 32-bit, specified for single instructions.

To optimize our generated code for existing LLVM analysis and optimization passes, we generate instructions for pointer arithmetic (`getelementptr`) instead of integer arithmetic. To safely handle all cases of operand combinations, we set the `null-pointer-is-valid` function attribute, which causes the address 0 to be treated as a valid address [5]. This prevents optimizations that consider arithmetic with a `null` pointer as undefined behavior.

Pointer arithmetic instructions in LLVM-IR have a base pointer and one or more offsets as input operands. The distinction between the base pointer and the offsets is not easily possible from machine code, for which reason we apply some simple heuristics. If a base register is present, we assume that the base register holds the base pointer and interpret the constant displacement as offset. Otherwise the constant offset specifies the pointer base. The scaled index register is commonly used as an offset, but not as a base pointer. Note that this heuristic only helps to guide the optimization; it does not impact the correctness or the behavior of the program.

If one of the segment registers `fs`/`gs`, which are commonly used for thread-local storage or system data structures, is specified as additional offset, the segment offset loaded from the CPU structure is interpreted as a pointer base. Finally, for the rare case of memory operations with a 32-bit address size, only integer arithmetic is used.

### 3.7 Instruction Coverage & Limitations

Rellume covers a significant portion of the base x86-64 architecture, including SSE and SSE2, which are also commonly used for floating-point computations on x86-64. The rarely used legacy x87 FPU and MMX instruction sets are not yet supported, though. Further, some instructions cannot be lifted independently of the target architecture, e.g., `syscall`, `cpuid` or `rdtsc`. For these instructions, no default implementation is provided. Instead, the tool using the lifter specifies a custom implementation using the configuration API.

Although our implementation of SSE/SSE2 is sufficient for most programs, it currently does not support floating-point exceptions and different rounding modes. As the handling these properties depends on a configuration register (`mxcsr`), run-time checks would have to be inserted in many places in the code, causing a significant overhead. Implementation of these rarely used features might be possible using the LLVM experimental constrained floating-point intrinsics, but this is left as future work.

### 3.8 Contrasts with Other Approaches

Our lifting approach differs from other lifters which directly transform x86-64 machine code to LLVM-IR in several ways.

Many lifters, e.g., McSema [31] and RetDec [9], do not keep register values in SSA registers, but load/store them to the CPU structure on every access. The transformation to more optimizable SSA registers is done using existing LLVM optimization passes. In contrast to our approach, these systems are unable to propagate type information, especially for smaller integer registers and vector types. The concept of storing multiple facets for different types can only be represented efficiently by using SSA registers directly. Additionally, the transformation from load/stores to SSA registers requires additional optimization time, which our lifter avoids.

DBrew-LLVM [17] also has register facets similar to our approach, but is much more eager in computing all values and generating PHI nodes unconditionally. Our lifter generates facets only on demand and therefore produces much smaller LLVM-IR code, which is beneficial for lifting and optimizing performance.

Treating the individual flag bits as separate registers is a common approach and also used by many other lifters [9, 31]. RetDec [9] contains a custom optimization pass that folds common flag evaluations (e.g., less-or-equal) after lifting. DBrew-LLVM [17] uses a cache for the operands of the last integer comparison instruction. Our approach is to compute flags preferably using arithmetic instructions instead of bitwise operations, exploiting the standard instruction combination pass.

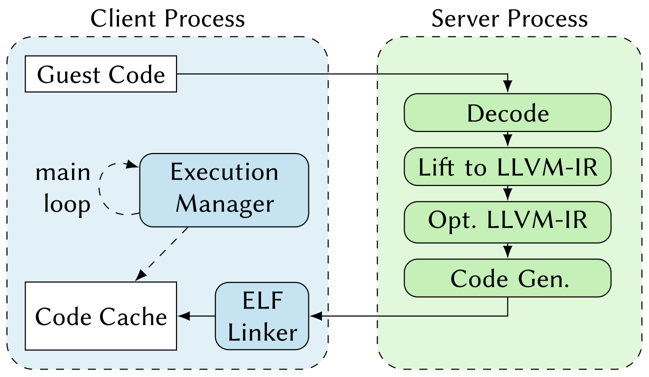


**Figure 3.** Overview of the Instrew client-server architecture. The program execution is controlled in the client process, while instrumentation and code generation happens on the server side. Guest code and instrumented code are transferred using an IPC protocol.

## 4 Rewriting Framework

To use our lifter library Rellume, we integrate it into our new framework named *Instrew*, which is also freely available and open-source[2], licensed under LGPLv2.1+. In the following, we outline the general architecture of our framework and describe further performance optimizations at the LLVM-IR level.

### 4.1 Client/Server Architecture

In contrast to most other binary instrumentation systems [24–26], we split the instrumenter into two processes: a lightweight *client* process, which manages the program state and executes the instrumented code, and a *server* process, which is responsible for actually lifting, instrumenting and compiling new code at the request of the client. They are connected via an IPC communication mechanism, as shown in Figure 3.

***Instrew Server*** The rewriting server is passive and only acts on client requests; it is stateless aside from a small number of configuration options. Whenever a client requests new code for a given address, the server queries the relevant instruction bytes via the same connection. It then decodes these instructions and passes them to our lifter library Rellume, which returns an LLVM function for the decoded instructions. Instrumentations can be applied during lifting to add code at instruction and basic block boundaries, and after the LLVM-IR is generated to transform the program at a higher level of semantics. For the latter, the server process can leverage any standard optimization pass available as part of the LLVM ecosystem. Finally, the LLVM module is compiled to an in-memory ELF object file, which is then sent

[2]https://github.com/aengelke/instrew, accessed 2020-02-17

back to the client. Note that the object file consists of a single ordinary function and can contain relocations and references to other functions, e.g., a system call handler, which have to be resolved by the client before execution.

This design concept of a separate server process has several benefits compared to the usual design of having the rewriter in the same address space. First, it opens the door to additional possible optimizations: a single server process can be used across multiple runs, reducing the initialization overhead; and the rewritten code can be cached across multiple executions of the process if both code and configuration remain the same. Second, it increases flexibility: a single server process can generate code for multiple clients (or threads in a single client), avoiding duplicated rewriting. In fact, the rewriting server can also run on a different, perhaps more powerful machine or even in the cloud. Third, it simplifies tool development, as developers do not need to care about not corrupting the memory of the guest program. This also eases the use of external libraries.

***Instrew Client*** The client is responsible for executing the actual (rewritten) program code and managing the *code cache*, which maps instruction addresses to already rewritten code fragments for future use. At program start, the guest binary is mapped into memory, the CPU state and the stack are initialized according to the System-V ABI [21], and the instruction pointer is set to the entry address specified in the ELF headers. Whenever the client reaches an instruction address that is not yet translated[3], it sends a request to the server to generate rewritten code for the current address. The server may query required instruction bytes and responds with an ELF object file containing a function to start execution at the requested address. Before adding the new code to the code cache, the client has to apply ELF relocations and resolve missing symbols, making the client essentially a very simple object file linker. The dispatcher just looks up the function for the requested address and calls it with the CPU state.

***Communication*** The communication between the two processes is currently implemented as a custom binary protocol sent over UNIX pipes. However, the use of sockets, shared memory regions, or even direct access to the memory of the client process as supported on newer Linux kernels[4] is possible as well and could easily be added.

## 4.2 Translation Granularity

For the granularity of rewriting and instrumentation, there are several possibilities:

- *Instruction:* each instruction gets translated one-by-one. This allows accurate detection of faults, but causes very high overhead.
- *Basic Block:* a sequence of instructions is decoded until the first branch. With this method only instructions that are actually executed will be accessed.
- *Function:* instructions are encoded as far as possible, following conditional branches and only stopping at indirect branches, call and return instructions. Here, instructions that never get executed could be accessed and rewritten as well.

In our framework, we opted for the *function* granularity, as it has three major benefits: (1) the optimization possibilities from LLVM's whole-function and loop transformations can be exploited as complex control flows are also lifted to the LLVM-IR, leading to a better register allocation and more optimized code; (2) it simplifies the client as no chaining of blocks is necessary for good performance; all possibilities where chaining applies are already handled when decoding and generated in a single function block; and (3) it reduces the number of required switches between the client and the server.

We note that our approach works with unmodified LLVM libraries. Implementing a chaining optimization is rather difficult with upstream LLVM — HQEMU [19], for example, requires a patched LLVM library to support a specific operation mode closely tied to their usage [4].

## 4.3 Optimizations for LLVM-based Dynamic Rewriting

To further increase performance, we added three optimizations to our LLVM-based rewriter.

***HHVM Calling Convention*** For x86-64 targets, LLVM supports the `hhvmcc` calling convention, which was originally designed for the HipHop Virtual Machine [29], a JIT compiler for PHP and Hack [27]. In contrast to other calling conventions, 15 arguments can be passed and 14 values can be returned in general-purpose registers. Additionally, only 2 of 16 general-purpose registers are callee-save, namely `r12` and `rsp`.

This calling convention is also very beneficial for our case of binary rewriting: we can keep the values of 12 general-purpose registers in host registers during a context switch (e.g., the lookup/translation for a return or an indirect jump target). This reduces the amount of memory accesses at the beginning and end of a code block significantly. In addition, we pass a pointer to the CPU structure as parameter and get the address of the next instruction as additional return value. Note that we intentionally leave one register unused to reduce register pressure on the host side, avoiding register spilling for a translation cache lookup. The full register mapping can be found in Table 2.

[3]We consider an address as *translated* if and only if the address is the entry of a translated code fragment.

[4]Linux system calls `process_vm_readv`/`process_vm_writev`

**Table 2.** Use of host general-purpose registers at entry and exit of compiled code blocks based on the HHVM calling-convention. Registers `r12` and `rsp` are callee-saved, register `r14` is intentionally unused to reduce register pressure.

| Reg. | Usage | Reg. | Usage |
|---|---|---|---|
| `rax` | Guest `rax` | `r8` | Guest `r8` |
| `rcx` | Guest `rcx` | `r9` | Guest `r9` |
| `rdx` | Guest `rdx` | `r10` | Guest `r10` |
| `rbx` | New `rip` (on exit) | `r11` | Guest `r11` |
| `rsp` | Host Stack | `r12` | CPU struct ptr. |
| `rbp` | Guest `rbx` | `r13` | Guest `rbp` |
| `rsi` | Guest `rsi` | `r14` | — |
| `rdi` | Guest `rdi` | `r15` | Guest `rsp` |

As a further enhancement, it would be possible to use the SSE registers as well to keep more values in registers. However, this is currently not possible with upstream LLVM 9 and would require custom patches, decreasing the portability and maintainability of our instrumentation framework.

***Use of native fs/gs registers*** The x86-64 architecture has two special segment registers, namely `fs`/`gs`, whose base address can be used as additional offset to memory operands [22]. The base address of these segment registers can typically only be configured from kernel space. On Linux, this is done using the `arch_prctl` system call. While the base address can be added using additional calculations in the rewritten code, it is also possible to emit code that uses the native segment registers on x86-64 targets. We forward any corresponding `arch_prctl` system call to the host and then use the architecture-dependent LLVM address spaces 256 and 257 for `gs` and `fs`-based memory accesses. These address spaces correspond, by definition, to the respective segment registers [5].

***Optimization of flag usage on calls/returns*** Many arithmetical instructions on x86-64 overwrite the status flags in the `rflags` register. As these flags are only needed rarely, avoiding explicit computation of the flag value is relevant for performance. While within an LLVM function the dead code elimination pass will remove such computations where possible, this is not possible across boundaries of code blocks. Due to the use of function granularity, this is the case before all function calls and returns in addition to indirect jumps.

We are not aware of any compiler or calling convention that requires the status flags to be preserved over the boundaries of a `call` or `ret` instruction. Therefore, we assume that all status flags can be discarded when such an instruction is encountered. For compatibility, we provide an option to disable this optimization.

### 4.4 System Call Handling

In order to correctly handle system calls that may collide with the instrumentation system (e.g., `fork` or `arch_prctl`), all system calls are redirected to a separate function in the client. In most cases, the system calls are forwarded to the host operating system without modifications.

As a side-effect of intercepting all system calls, these can also be changed such that an application can run on changing architectures, e.g., Intel x86-64 code translated into AArch64 on an ARM platform. This requires changing the system call number, but for some syscalls also the order of arguments and the layout of passed data structures differ. The latter may involve using a temporary buffer.

## 5 Evaluation: Rewriting Overhead

To evaluate the quality of Instrew and Rellume, we first focus on the overhead caused by our framework with a null-instrumentation, that is, executing the code without functional changes, compared to an uninstrumented baseline. For this, we apply Instrew on the SPEC CPU2017 benchmark suite. We also compare the overhead of our tool against the no-instrumentation tool of Valgrind 3.15.0 [26] (Nulgrind), the state-of-the-art tool for heavy-weight dynamic binary instrumentation. Unfortunately, we could not directly compare against DBILL [25] as sources are not publicly accessible.

We did not compare against Pin [24], because Pin has a different scope in that it only allows insertion of function calls, but no further transformations or modifications of the code, as we do in our system.

Furthermore, to evaluate the quality of our x86-64-to-LLVM lifter, we run the benchmarks with HQEMU 2.5.2 [19], a QEMU variant that uses LLVM for code generation and optimization. We use HQEMU to emulate x86-64 user-level binaries on the same architecture and configured it to use the LLVM-only mode. Note that we had to patch HQEMU to avoid an option name collision with LLVM.

### 5.1 Setup

We use the reference input set for all benchmarks and run them single-threaded to avoid interference. Our target platform is based on Intel Xeon CPUs (E5-2697 v3, Haswell) clocked at 2.6 GHz (3.6 GHz in Turbo mode), 17 MiB L3 cache and 64 GiB main memory, running SUSE Linux 12 with Linux kernel 4.12.14-95.32 in 64-bit mode. All code is compiled using GCC 9.2.0 with the `-O3` option and linked against glibc 2.22. To avoid compatibility problems, the benchmarks are not specifically tuned for the target architecture, but for a generic x86-64 machine, implying that only SSE2 but no later extensions (including AVX) are generated. For Instrew, we use LLVM 9.0; for HQEMU we used LLVM 6.0.0, which is the latest supported version.

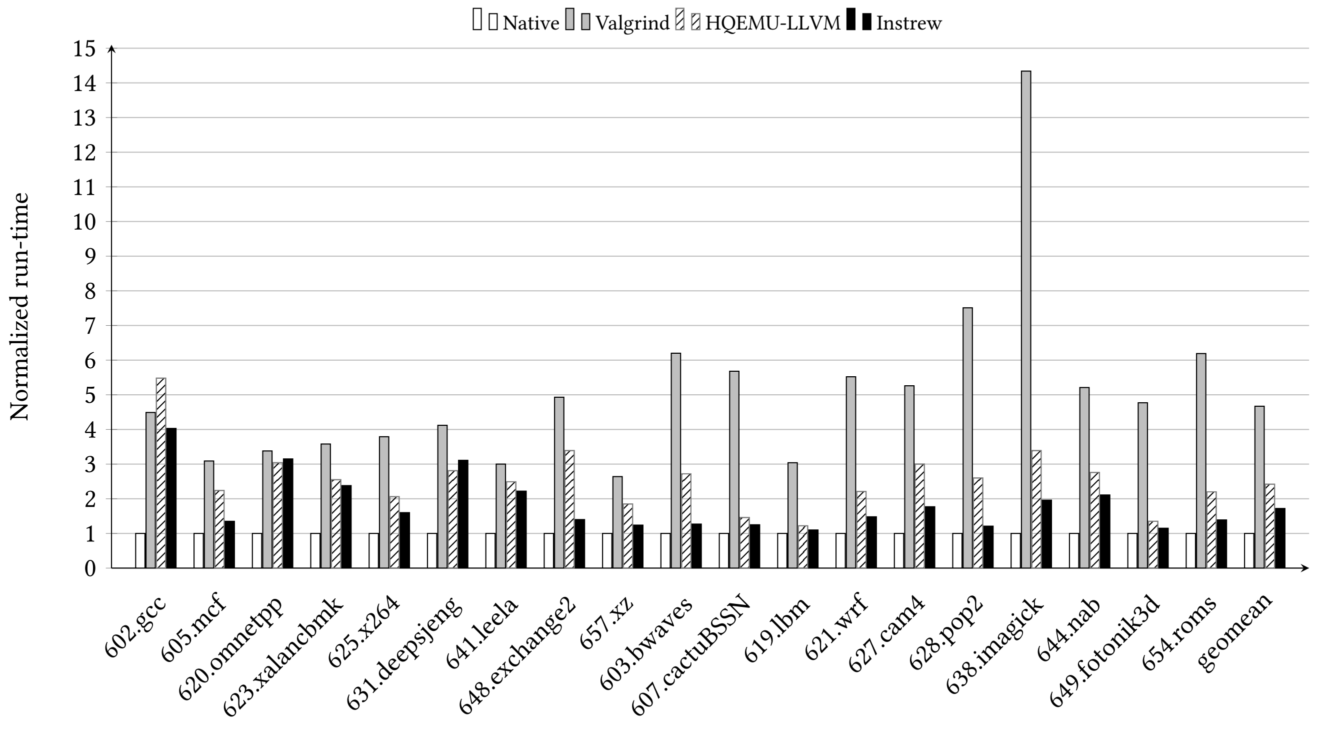


**Figure 4.** Performance results on the SPEC CPU2017 benchmarks on x86-64 with the *ref* workload.

We only use 19 of the 20 SPEC CPU benchmarks: we excluded 600.perlbench, because it used x87 FPU instructions, which are currently unsupported by Instrew.

## 5.2 Results

Figure 4 shows the benchmark results. We can see that Instrew has an average overhead of 72%, which is significantly lower than the overhead of Valgrind with 367%. Especially on floating-point benchmarks, the better handling of floating-point operations and SIMD instructions becomes visible. While Valgrind has an average overhead of 490% on this subset, Instrew has an overhead of only 43%.

The results further show that our approach to lifting x86-64 machine code to LLVM-IR generally has a significantly lower overhead than HQEMU in LLVM-only mode (overhead of 142%). Also here, the performance difference is emphasized on floating-point benchmarks.

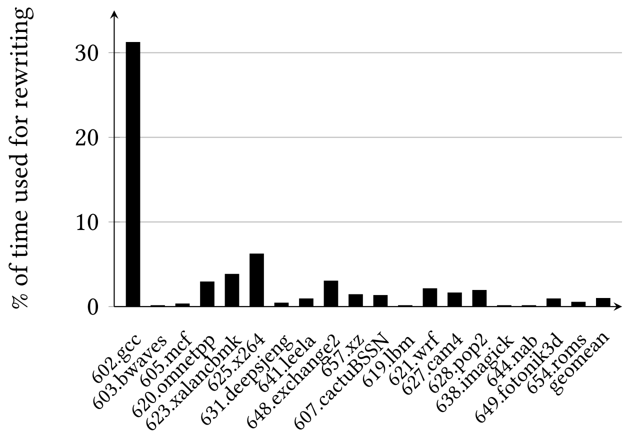
To better understand the reasons for the slowdown of Instrew, we also analyze how much wall-clock time is spent for rewriting, including code lifting, optimizations, code generation and code transfer between the processes. These results are shown in Figure 5. While the time spent for rewriting is usually low in comparison to the total run-time (cf. Figure 5a), this is not the case for the 602.gcc benchmark, where 31% of the time is spent on code generation. The reason is that this benchmark executes a significant amount of code. As the benchmark consists of three independent payloads, the code is actually rewritten three times.

For the benchmarks 620.omnetpp and 631.deepsjeng, Instrew is even slower than the HQEMU in LLVM-only mode. The main reason for this is the amount of function calls executed in these benchmarks. In contrast to Valgrind or HQEMU, Instrew does not trace into function calls, inferring a dispatching overhead not only for function returns, but also for calls. These benchmarks would benefit from decoding into nested functions, however, with unforeseeable effects for rewriting time.

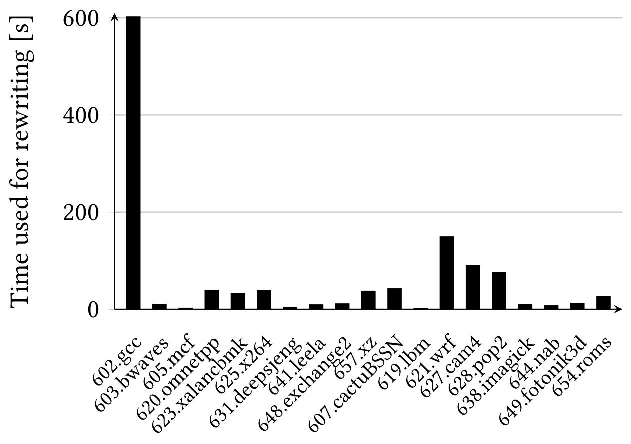
During the rewriting process itself, 65% of the time is spent on machine code generation (cf. Figure 5c). The SelectionDAG code generator is known to have performance problems, but the replacement back-end (GlobalISel) is not yet ported for x86-64 [7] by the LLVM community. Once completed, we expect substantial further improvements also for our infrastructure.

## 5.3 Discussion

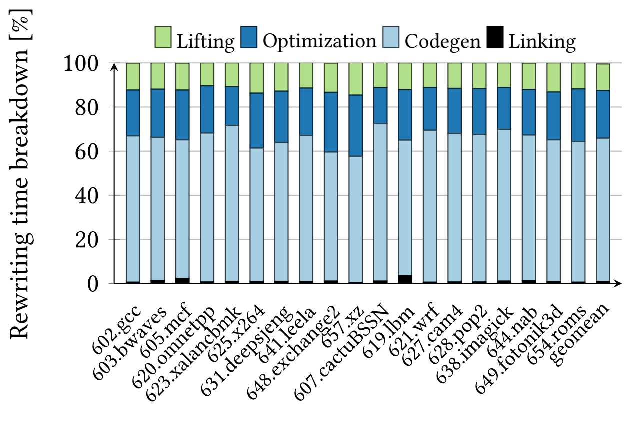
Instrew provides clear performance advantages over Valgrind on many benchmarks due to more advanced optimizations and due to its use of a high-quality code generator. Especially on floating-point benchmarks the generated code comes close to the performance of the original code. This clearly shows that the lifter generates LLVM-IR code that



**(a)** Time used for rewriting relative to the total running time of the benchmarks.



**(b)** Absolute time used for rewriting the benchmarks.



**(c)** Breakdown of rewriting time into decoding/lifting, LLVM-IR optimization, machine code generation, and linking.

**Figure 5.** Time used by Instrew for generating and optimizing code for the different benchmarks.

can be compiled again to performant machine code. Further, the comparison with HQEMU shows that our lifting approach leads to a notably more efficient usage of the LLVM compilation infrastructure.

Moreover, the average overhead of Instrew on the SPEC CPU2017 INT benchmarks with 109% is also significantly lower than the overhead reported for DBILL with 240% on SPEC CINT2006 [25]. Even though this comparison is not fully accurate, it gives an indication that avoiding TCG as additional intermediate code representation is advantageous.

Due to the fact that LLVM has a rather slow code generator, every code generation has a significant cost. This cost is the biggest general drawback for the adoption of our approach: the rewriting time has to amortize over the run of the program, which either requires codes with smaller code footprints and/or longer running times. The latter, though, is not uncommon for many realistic workloads, especially in the High Performance Computing (HPC) space. Additionally, ongoing developments in the LLVM community will further reduce this issue in the future.

## 6 Use Case: Dynamic Binary Translation

To demonstrate the use of our framework, we show how Instrew and Rellume can used for retargetability. We choose this use case, as this demonstrates the entire framework and as it is one of the most extreme use cases. In particular, we use Instrew as dynamic binary translator to execute programs compiled for x86-64 on an AArch64 machine. We compare the performance of Instrew with QEMU [10], the state-of-the-art emulation tool running in user-space emulation mode, as well as the performance of the program compiled natively for AArch64. We would have also included HQEMU [19] in our comparison, but experienced crashes when running with any program, presumably due to memory corruption.

The evaluation system for this use case is an ODroid C2 single-board computer equipped with 4×ARM Cortex-A53 clocked at 1.5 GHz and 2 GiB of memory. The board uses Debian 10 (Buster) with Linux kernel 3.14.79. All code are compiled with GCC 8.3.0.

Due to time and resource limitations, most notably memory, on the ODroid platform, we use a subset of the SPEC CPU2017 benchmarks and due to memory constraints, we only use the *train* benchmark workload. We note that the memory constraint is imposed by the benchmark itself and not by the instrumentation systems.

***Results*** Figure 6 shows the results of our experiments. The x86-64 code emulated with QEMU on AArch64 is on average 5x slower than the code compiled and optimized for AArch64. With Instrew, this slowdown is significantly lower at 3x, providing an average speed-up of 40% compared to QEMU.

Due to the smaller workload size, the rewriting and optimization using LLVM takes relatively more time. However,

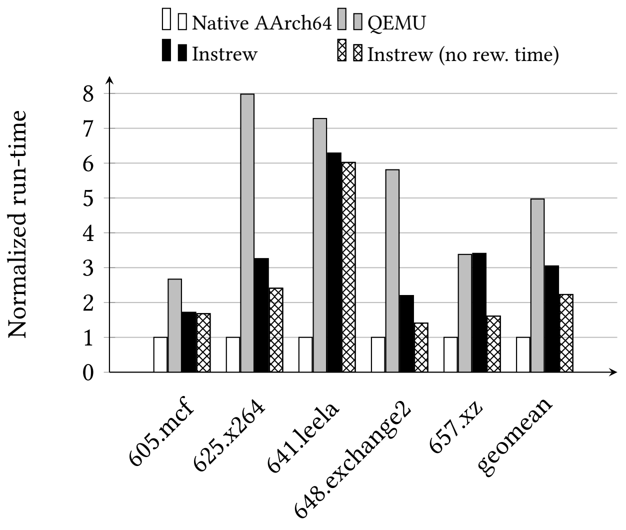


**Figure 6.** Performance results on some SPEC CPU2017 benchmarks with train workload emulating x86-64 binaries on AArch64. The *native* run-time is the same code compiled directly to AArch64. Due to the small workload size, rewriting time has a significant performance impact.

the actual impact depends on the benchmark and the workload. For example, on the 605.mcf benchmark, the rewriting time is negligible, whereas on the 657.xz benchmark more time is spent on rewriting code for AArch64 than on the actual execution of the program itself.

Excluding the rewriting time, Instrew is still on average 2.23x slower than the code directly optimized for AArch64. We note that a translation cache in the Instrew server can be implemented, avoiding the rewriting time on subsequent program executions.

***Discussion*** From our results, we can see that significant speed-ups for cross-ISA translations can be achieved by involving LLVM into the translation process. This is in line with previous findings reported by Hong et al. [19]. Unfortunately, we were unable to include HQEMU in our comparison here, but also here we expect a slightly better performance compared to their LLVM-only mode due to translation with function granularity and our optimized handling of SIMD instructions. However, it also demonstrated another advantage of using LLVM: by relying on a widely used, tested and supported infrastructure, we achieve a more portable and robust solution that can be used across platforms.

However, from the fact that the generated code is still 2.23x slower than the optimal reference code, we conclude that just the use of a high-quality code generator is not sufficient to achieve optimal performance for binary translation. Information about the integrity of function returns and additional pointer alias information, especially for the stack layout, would lead to a higher code quality. This kind of information is usually not available in the executable files, as it is discarded during compilation.

## 7 Related Work

Several tools exist for the purpose dynamic binary rewriting and instrumentation. Typically, these either target the purpose of instrumentation on the same architecture, or they perform dynamic translation to a different architecture.

***Dynamic Binary Instrumentation*** DBILL [25] is an instrumentation framework based on LLVM. Instead of lifting directly from machine code to LLVM-IR, they first lift the machine code to TCG and from there to LLVM-IR. While this allows them to easily port their instrumenter to other architectures, information about the original instructions is lost at the point where instrumentation can be performed. The biggest drawback, however, is the lack of floating-point support, making it unusable for a large class of applications.

For heavy-weight dynamic binary instrumentation, the state-of-the-art tool is Valgrind [26]. All code is lifted with superblock granularity to the architecture-independent VEX intermediate representation, where instrumentations and other code modifications can be performed. Before new machine code is generated, some simple optimizations are applied. The instrumenter runs in the same address space as the instrumented program.

DynamoRIO [12] also allows for heavy-weight transformations and represents program code in a low-level, but architecture-dependent intermediate representation. Instruction modifications can be either performed via an architecture-independent API, or by directly modifying the instructions. However, no optimizations are applied. Also, there is no address space separation between instrumenter and the guest.

Pin [24] allows inserting calls to functions in the instrumenting tool at any point in the program. However, modifications to the program and heavy-weight transformations are not possible.

DynInst [14] runs the instrumenter in a different process than the instrumentation target. The target process is controlled and modified using the `ptrace` debugging API of the kernel. DynInst has an API to modify and replace specific functions within the instrumented program. However, DynInst is not targeted at heavy-weight and complete program transformations.

The QuarkslaB Dynamic binary Instrumentation (QBDI) [1, 3] framework is a dynamic binary instrumenter, which also uses LLVM. However, they do not use the architecture-independent LLVM-IR, but the LLVM-MC machine code representation for their instrumentation. Although QBDI attempts to separate the instrumenter from the instrumentation target, they run in the same address space and even share the same heap and loader [2], which adds further requirements to the instrumented program.

***Dynamic Binary Translation*** QEMU [10, 28] is an emulator supporting several architectures as host and guests. All guest code is lifted to the architecture-independent intermediate representation TCG, which then can be compiled to several other (host) architectures. For complex instructions, TCG can emit calls to pre-compiled functions which emulate a specific behavior. QEMU supports full-system emulation with a software memory management unit or dedicated hardware support as well as the emulation of Linux user-space programs in the same address space.

HQEMU [19] is an extension of QEMU which detects hot code traces and optimizes them using LLVM in a separate thread. As with DBILL, machine code is lifted from the TCG IR to the LLVM-IR. A client-server architecture for HQEMU has been proposed as well [20]. In contrast to our approach, they keep the TCG-only translator as fast-path in the client to reduce the performance overhead.

***Lifting x86-64 to LLVM-IR*** Regarding our work regarding lifting x86-64 binary code to LLVM-IR, we are aware of several projects doing something similar, although the goals differ. Such lifters are mostly targeted at static software analysis and reverse engineering. McSema [31] lifts compiled x86 binaries to LLVM-IR and also tries to recover functions and global variables, targeted for software analysis. RetDec [9] and fcd [15] are decompilers that use LLVM as intermediate representation. Both projects only support a limited amount of instructions and the resulting LLVM-IR is not intended to be executed again. S2E/Revgen [16] uses McSema and TCG to lift entire binary files to LLVM-IR and allows retargeting to other architectures, but is not tuned for performance or binary size.

Such systems designed for static analysis do not have strong performance requirements. While this technically does not render them unsuitable for binary instrumentation, they are hard to incorporate for a performance-focused rewriting system. For example, VMILL [32] is prototypical binary instrumenter based on McSema/Remill, but has significant performance issues, mostly because all memory accesses are handled using a software address translation.

We know only one project, which lifts x86-64 machine code directly to LLVM-IR with the intention of executing the lifted code at run-time: DBrew-LLVM [17] is a library that uses LLVM-IR to perform dynamic code optimization at runtime. User-specified functions are lifted from x86-64 machine code to LLVM-IR, optimized, and compiled back to machine code. The compilation takes place in the same address space and is explicitly configured by the program itself. In contrast to this work, our lifter does not have limitations such as lacking registers (segment registers, direction flag) and an enforced calling convention. Furthermore, we focus not only on the performance of the compiled code, but also on the performance of the rewriting process itself.

## 8 Summary & Outlook

In this paper, we presented Instrew, a dynamic binary rewriting and instrumentation framework based on LLVM. Instrew lifts x86-64 machine code at function granularity to LLVM-IR, which allows using the high-quality code generator of LLVM. Our lifting approach has a special focus on the performance of the lifted code. Instrumentation and behavioral changes can be made at the level of LLVM-IR. We also propose the use of a client-server architecture for binary instrumentation, splitting off the instrumenter in a separate process to allow further optimizations like caching. In addition, our framework is retargetable, allowing the execution of x86-64 programs on other 64-bit architectures supported by LLVM, e.g., 64-bit ARM. On the SPEC CPU2017 benchmarks our framework has an average performance overhead of just 72%, which is a 1/5 of the overhead of the state-of-the-art instrumentation tool Valgrind. The performance improvement is even larger on floating-point benchmarks.

Further optimizations are possible in the handling of function calls to reduce the dispatching overhead. A further reduction of the rewriting overhead could be possible by caching rewritten code across different program runs, but would require a more persistent infrastructure at the OS level. The latter would then also open the door more dynamic OS and runtime environments enabling on the fly instrumentation, optimization and translation of any code as default operation without user intervention.

## References

[1] [n. d.]. QBDI: A Dynamic Binary Instrumentation framework based on LLVM. https://github.com/QBDI/QBDI, accessed 2020-02-17.

[2] [n. d.]. QBDI User Documentation. https://qbdi.readthedocs.io/en/stable/user.html, accessed 2020-02-17.

[3] [n. d.]. QuarkslaB Dynamic binary Instrumentation. https://qbdi.quarkslab.com, accessed 2020-02-17.

[4] 2018. HQEMU v2.5.2 Technical Report, installation guide. http://csl.iis.sinica.edu.tw/hqemu/download/quickstart-2.5.2.pdf, accessed 2020-02-17.

[5] 2019. LLVM 9 Documentation: LLVM Language Reference Manual. http://releases.llvm.org/9.0.0/docs/LangRef.html, accessed 2020-02-17.

[6] 2019. LLVM 9 Documentation: The Often Misunderstood GEP Instruction. http://releases.llvm.org/9.0.0/docs/GetElementPtr.html, accessed 2020-02-17.

[7] 2020. LLVM 10 Documentation: Global Instruction Selection. https://llvm.org/docs/GlobalISel/index.html, accessed 2020-02-17.

[8] Arm Limited. [n. d.]. Arm Compiler for Linux – Arm Developer. https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-compiler-for-linux, accessed 2020-02-17.

[9] Avast Software. [n. d.]. RetDec. https://retdec.com/, accessed 2020-02-17.

[10] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[11] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE)*. 9–16.

[12] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO)*. 265–275.

[13] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. IEEE Computer Society, 213–223.

[14] Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications (IJHPCA)* 14, 4 (2000), 317–329.

[15] Felix Cloutier. [n. d.]. fcd. http://zneak.github.io/fcd/, accessed 2020-02-17.

[16] Cyberhaven. 2018. Translating binaries to LLVM with Revgen. http://s2e.systems/docs/Tutorials/Revgen/Revgen.html, accessed 2020-02-17.

[17] Alexis Engelke and Josef Weidendorfer. 2017. Using LLVM for Optimized Lightweight Binary Re-Writing at Runtime. In *Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*. 785–794.

[18] Hal Finkel. 2016. Intrinsics, Metadata, and Attributes: The story continues. In *2016 LLVM Developers' Meeting*.

[19] Ding Yong Hong, Chun Chen Hsu, Pen Chung Yew, Jan Jan Wu, Wei Chung Hsu, Pangfeng Liu, Chien Min Wang, and Yeh Ching Chung. 2012. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *International Symposium on Code Generation and Optimization (CGO)*. 104–113.

[20] Chun-Chen Hsu, Ding-Yong Hong, Wei-Chung Hsu, Pangfeng Liu, and Jan-Jan Wu. 2015. A dynamic binary translation system in a client/server environment. *Journal of Systems Architecture* 61, 7 (2015), 307–319.

[21] Jan Hubička, Andreas Jaeger, Michael Matz, and Mark Mitchell. 2013. System V Application Binary Interface, AMD64 Architecture Processor Supplement. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf.

[22] Intel Corporation. 2019. *Intel 64 and IA-32 Architectures Software Developer's Manual*. https://intel.com/sdm, accessed 2020-02-17.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *SIGPLAN Conference on Programming language design and implementation (PLDI)*, Vol. 40. 190–200.

[25] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. DBILL: an efficient and retargetable dynamic binary instrumentation framework using LLVM backend. In *International Conference on Virtual Execution Environments (VEE)*. 141–152.

[26] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN notices*, Vol. 42. 89–100.

[27] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. 151–165. https://doi.org/10.1145/3192366.3192374

[28] QEMU Developers. 2019. Documentation/TCG. https://wiki.qemu.org/Documentation/TCG, accessed 2020-02-17.

[29] Philip Reames. 2015. LLVM Phabricator: Calling convention for HHVM (D12681). https://reviews.llvm.org/D12681, accessed 2020-02-17.

[30] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*. 17–30.

[31] Trail of Bits, Inc. [n. d.]. McSema. https://www.trailofbits.com/research-and-development/mcsema/, accessed 2020-02-17.

[32] Trail of Bits, Inc. [n. d.]. VMILL (repository). https://github.com/lifting-bits/vmill, accessed 2020-02-17.

[33] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. 2004. A tool suite for simulation based analysis of memory access behavior. In *International Conference on Computational Science*. Springer, 440–447.