Technische Universität München
Fakultät für Informatik

Optimizing Performance Using Dynamic Code Generation

Alexis Friedrich Engelke

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:		Prof. Dr. Pramod Bhatotia
Prüfer der Dissertation:	1.	Prof. Dr. Martin Schulz
	2.	Prof. Dr. Thomas Ludwig, Universität Hamburg

Die Dissertation wurde am 23.06.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.10.2021 angenommen.

Abstract

Dynamic rewriting of machine code is a widely used technique for enabling program compatibility, analyzing program behavior, particularly with the goal of improving security or performance, and for dynamically optimizing performance. Many existing systems focus on optimizing the rewriting step itself and therefore typically roll their own, often low-level, internal code representation. However, such systems attain their low overhead in rewriting by trading performance and code quality of the newly generated code, thereby introducing significant overheads to the execution of complex or longer running programs. Compiler frameworks like LLVM, on the other side, provide high-quality code generators, but require the "source" code to be in a higher-level and more abstract code representation.

Therefore, in this thesis I present a new, performance-focused library *Rellume* to lift machine code to efficient LLVM-IR code, bridging the gap between binary rewriting systems and the LLVM framework and enabling an effective use of LLVM in the context of dynamic binary rewriting systems.

Based on Rellume, I create *Instrew*, a performant LLVM-based framework for dynamic binary translation and instrumentation implementing a novel client–server approach, which increases flexibility by separating the process of code transformation from the actual execution. Performance results on the SPEC CPU2017 benchmarks show that Instrew reduces the execution overhead to less than 1/17th of the widely used binary translator QEMU and less than 1/9th of the state-of-the-art instrumentation framework Valgrind.

Furthermore, I develop the API and library *BinOpt* to exploit run-time-only information of applications by dynamically specializing compiled functions. An application developer can specify code regions and run-time information using library calls, which trigger dynamic optimizations during program executions. I compare three different rewriting approaches and find that LLVM-based binary optimization on the top of Rellume allows for performance improvements of up to 67% on optimized image processing kernels.

This enables the flexible use of LLVM in dynamic binary rewriting, advances the state of high-performance dynamic binary translation as well as instrumentation, and opens up a new class of dynamic optimization opportunities.

Kurzfassung

Dynamische Umschreibung von Maschinencode ist eine weit verbreitete Technik um Programmkompatibilität zu erhöhen, um Programmverhalten zu analysieren, insbesondere im Hinblick auf eine Erhöhung von Sicherheit oder Performanz, und um die Performanz von Programmen dynamisch zu optimieren. Viele existierende Systeme fokussieren die Optimierung des Umschreibens auf Kosten der Performanz und Qualität des neu generierten Codes und führen damit zu einem signifikanten Geschwindigkeitsverlust bei der Ausführung von komplexen oder lang-laufenden Programmen. Auf der anderen Seite stellen Compilerframeworks wie LLVM hoch optimierte Codegeneratoren bereit, erwarten aber, dass der Quellcode auf einer höheren Ebene in einer abstrakteren Coderepräsentation vorliegt.

Daher stelle ich in dieser Dissertation die neue, performanzfokussierte Bibliothek *Rellume* vor, welche Maschinencode zu effizientem LLVM-IR Code transformieren kann und damit die Brücke zwischen Systemen für Binärumschreibung und dem Framework LLVM schlägt. Dies ermöglicht eine effektive Nutzung von LLVM im Kontext dynamischer Systeme für Binärumschreibung.

Basierend auf Rellume erstelle ich *Instrew*, ein performantes LLVM-basiertes Framework für dynamische Binärübersetzung und -instrumentierung. Das Framework implementiert einen neuartigen Client-Server-Ansatz, welcher die Flexibilität des Systems erhöht, indem die Codetransformation von der eigentliche Ausführung des umgeschriebenen Codes getrennt wird. Die Performanzergebnisse auf den SPEC CPU2017 Benchmarks zeigen, dass Instrew den Geschwindigkeitsverlust auf weniger als 1/17 des weit verbreiteten Binärübersetzers QEMU und weniger als 1/9 des Instrumentierungsframeworks Valgrind reduziert.

Darüber hinaus entwickele ich die API und Bibliothek *BinOpt*, um Informationen in Anwendungen, die nur zur Laufzeit vorhanden sind, auszunutzen, indem kompilierte Funktionen dynamisch spezialisiert werden. Ein Anwendungsentwickler spezifiziert Coderegionen und Laufzeitinformationen mittels Bibliotheksaufrufen, welche dynamische Optimierungen während der Programmausführung anstoßen. Ich vergleiche drei unterschiedliche Ansätze für Binärumschreibung und finde heraus, dass ein LLVM-basierter Ansatz aus Basis von Rellume einen Performanzgewinn von bis zu 67% auf optimierten Bildverarbeitungsoperationen erzielt.

Diese Arbeit ermöglicht die flexible Nutzung von LLVM für dynamische Binärumschreibung, erweitert den Stand der Technik im Bereich von hoch-performanter dynamischer Binärübersetzung sowie -instrumentierung und eröffnet eine neue Klasse an dynamsichen Optimierungsmöglichkeiten.

Acknowledgments

First and foremost, I want to thank Martin Schulz for his continuous support, for giving valuable feedback, and especially for giving me the freedom to follow my own research interests. I want to thank Thomas Ludwig for his additional advice and, more importantly, for drawing my interest to High-Performance Computing and performance optimization in general. I also want to thank Arndt Bode for his guidance and advice.

I want to thank those, who contributed to the technical development of the projects: Josef Weidendorfer, not only for motivating my work on dynamic binary optimization, but also for his countless ideas and discussions; David Hildenbrand for implementing Drob and his ideas and support; and Dominik Okwieka for his work on generalizing Rellume to different architectures.

I want to thank my current and former colleagues for providing an enjoyable working environment, in particular Vincent Bode, David Hildenbrand, Beate Hinterwimmer†, Clemens Jonischkeit, Julian Kirsch, Tilman Küstner, Jürgen Obermeier, Amir Raoofy, Carsten Trinitis, and Dai Yang. I am grateful for the assistance of various tutors helping me to run my large lab courses, especially Alexandra Graß, Michael Jungmair, Timm Knörle, Max Oberberger, Benjamin Rickels, and Tobias Schmidt. I also want to thank the colleagues from the Rechnerbetriebsgruppe (RBG) for providing a variety of services together with a solid and stable infrastructure.

Further, I want to thank Julian Kirsch for his advice on academic survival; David Schneller for convincing me to work as student volunteer at SC19 as well as interesting discussions and his support; Charlie Groh for motivating me to finish this thesis[1] and interesting (and funny) discussions; and current and former players of the hxp CTF team for joyful and exciting weekends.

Thanks to Martin Schulz, David Hildenbrand, Josef Weidendorfer, and Jakob Roth for providing valuable feedback on an earlier version of this thesis.

Finally, I would like to thank numerous other people for their discussions, ideas, and support, on technical and non-technical matters. Unfortunately, this page is too small to list all of them and any attempt to list them would be incomplete.

Some of the experiments were conducted on the Bavarian Energy, Architecture and Software Testbed (BEAST) and the CoolMUC-2 cluster at the Leibniz Supercomputing Center (LRZ).

[1] He frequently asked “Bist du jetzt fertig mit deiner Doktorarbeit?” — starting from *day one*.

Contents

1 Introduction

1.1 Motivation

Although processing capabilities of modern Central Processing Units (CPUs) are constantly increasing, efficient and optimized execution of programs is still a very important topic: high performance and energy efficiency demand an effective usage of available computation units. Efficient execution of program code requires transforming the program source into optimized machine code, which can then be executed directly by hardware. This is typically achieved by compiling source files into a single binary file containing the optimized machine code. When the program is subsequently executed, the binary file is loaded into memory by the operating system for execution by the CPU. While the compilation step allows for machine code generation for the target CPU architecture, it also limits portability of the compiled binary file to other systems with different CPU architectures.

Programming of performance-sensitive programs is usually done in low-level languages, like C, C++, or Fortran, that include less abstractions and therefore allow optimizations closer to the hardware level. For the purpose of performance, such languages generally also omit extra checks for program integrity and instead assign the responsibility of ensuring program correctness to the programmer, with potentially severe consequences with regard to security. Further, as the compilation step is strictly separated from program execution, information that is only available while the program is running, for example configuration options, cannot be taken into account for subsequent dynamic program optimization.

These problems can be addressed with the widely used technique of *dynamic binary rewriting*, which transforms machine code of a program during its execution. This technique covers a wide range of applications: program compatibility with other architectures can be achieved by dynamically translating the program code to the target architecture [Bel05]. Binary rewriting is also frequently used for program analysis, where the behavior of the program can be modified to extract additional performance statistics [Luk+05; BGA03] or add extra checks for unintended behavior, capturing possible bugs and security issues [NS07b]. Furthermore, dynamic optimization techniques allow for improving the application performance after the initial compilation process, making use of run-time-only information [WB16].

1.2 Challenges

Systems for dynamic binary rewriting are not only faced with the challenge of parsing, analyzing, and producing machine code on-demand, but also have to perform these processing steps *themselves* with low overhead, as the transformation is performed at run-time during the execution of the program and therefore the associated transformation cost is accounted for in the overall execution time.

Therefore, many systems for dynamic binary rewriting use a low-level Intermediate Representation (IR) for storing and transforming program code, optimized for fast code generation. Such an IR can either be independent of the architecture, allowing to retarget program execution to a different architecture and increasing portability to other platforms [NS07b; Bel05], or may not abstract architecture-specific details at all, allowing for even faster code generation while requiring higher effort and more analyses for actual code modifications [BGA03; Luk+05]. Still, in both cases only few and lightweight optimizations are applied to keep transformation costs low.

However, the lack of optimizations and optimized machine code generation after performing code transformations leads to significant slowdowns on applications with longer execution times. This is especially relevant for applications in the area of High-Performance Computing (HPC), which typically are computationally intensive and also have comparably long execution times, even for small workloads.

This motivates enhancing dynamic binary rewriting techniques with additional and more complex optimizations in combination with an optimizing machine code generator. While implementing such transformations anew is technically possible, it would be accompanied by a significant amount of engineering and maintenance effort. At the same time, such optimizations are already widely implemented by compilers. In fact, most industry-standard optimizing compilers are not only capable of performing a variety of program transformations, but also have highly tuned code generation back-ends for many processor architectures.

One of these widely used compiler infrastructures is the LLVM project [LA04], which not only focuses on performance, but also emphasizes modularity and flexible applicability as design goals. In particular, the framework has a dedicated Application Programming Interface (API) for constructing and transforming program code and is completely usable without further toolchain dependencies. This makes LLVM a well-suited framework for the use case of dynamic binary rewriting. Nonetheless, this requires a transformation to lift machine code to LLVM's IR.

1.3 Approach

This motivates bridging the gap between systems for binary rewriting, operating on low-level machine code, and the LLVM compiler infrastructure, performing high-level code transformations and optimized code generation.

To this end, this thesis describes *Rellume*, a flexible library for lifting machine code to LLVM-IR optimized code for use in dynamic contexts. As such, the library has two main goals: the generated LLVM-IR code shall be performant and also the lifting process itself shall take place with low overhead. Additionally, generating architecture-independent LLVM-IR code enables dynamic retargeting of program code to other architectures. This library serves as a building block for developing performance-focused systems for dynamic binary rewriting based on the LLVM compiler infrastructure.

Based on this lifting library, *Instrew* is an LLVM-based framework for performance-focused dynamic binary translation and heavy-weight instrumentation; Figures 1.1a and 1.1b give a high-level overview on the approach. The original machine code is lifted to LLVM-IR with function granularity using *Rellume*, allowing Instrew to use the optimization and code generation infrastructure provided by LLVM to its full extent. The provided tool API allows for modifications of the program code at LLVM-IR level, enabling tool developers to leverage LLVM optimizations to reduce the overhead incurred by dynamic program instrumentation.

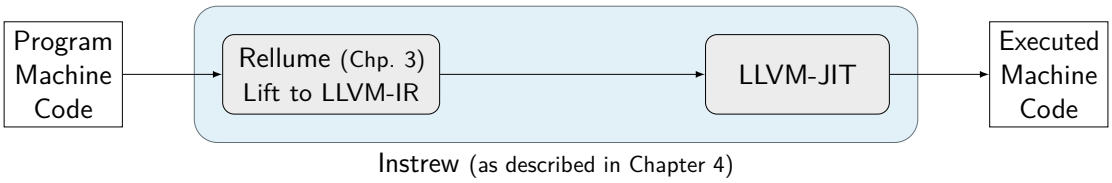
In addition to transparent rewriting of applications, dynamic binary rewriting can also be controlled by the application itself. This enables an application to explicitly trigger dynamic rewriting of specific code portions, for example for the purpose of optimization when they find to have additional information that can lead to performance improvements. *BinOpt* is a library that allows for application-guided specialization of selected functions on additionally available run-time-only data (see also Figure 1.1c). The API of the library is designed to be easily implementable with different rewriting approaches. In particular, one approach (DBLL) uses *Rellume* to perform LLVM-based optimization of machine code at run-time, leveraging the availability of many compiler transformations and a highly tuned machine code generator.

1.4 Contributions

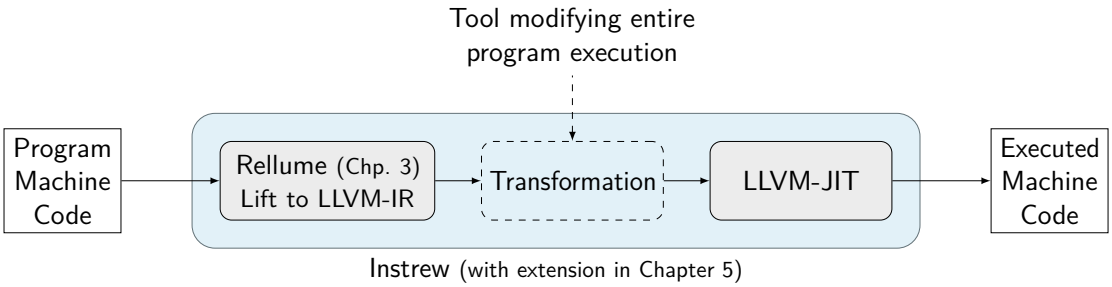
The key contributions of this thesis include:

- Rellume, a performance-oriented library for **lifting machine code to LLVM-IR**, designed for use in dynamic contexts.

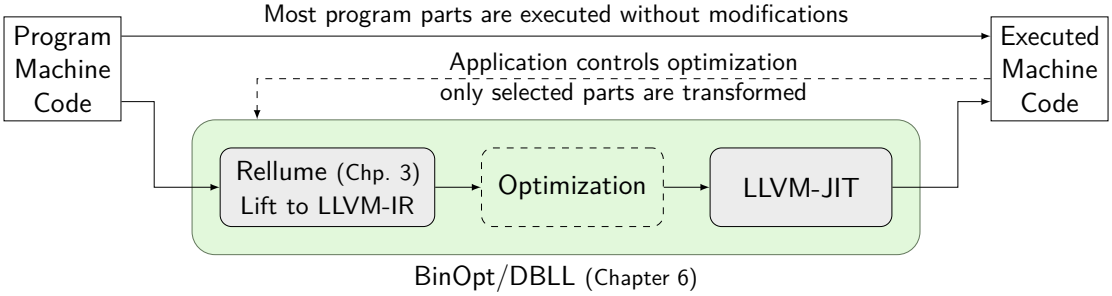
  The library can transform machine code into architecture-independent LLVM-IR code with the same semantics. The produced LLVM-IR code can be compiled again for execution, even on other platforms as long as they share the same pointer width. Optionally, the relation of the lifted code to the machine code is made explicit to enable tracking of instructions, register usage, or memory accesses. The lifting process requires no further information beyond the machine code and relevant information, for example about the



(a) *Overview on the* Instrew *framework. All executed parts of the program are lifted to LLVM-IR using* Rellume *and compiled again using the LLVM JIT compiler, optionally to a different target architecture.*



(b) *The* Instrew *framework used for Dynamic Binary Instrumentation. All executed parts of the program can be transformed by a tool, which can modify the program semantics. Afterwards, code fragments are compiled back to machine code using the LLVM JIT compiler.*



(c) *Overview on the* BinOpt *library. The program is generally executed without modifications. The application explicitly controls which parts are dynamically optimized. With an LLVM-based optimizer, the machine code is then lifted LLVM-IR using* Rellume *and after making use of specified optimization data, the specialized code is compiled back to machine code using the LLVM JIT compiler.*

Figure 1.1: *Overview on the approaches of* Instrew *and* BinOpt *and their relation to the lifting library* Rellume*.*

control flow, is recovered automatically. This makes Rellume also generally suitable for application a dynamic context.

The library not only focus on the performance of the lifted code, but also the efficiency of the lifting step itself, which is especially important in performance-sensitive contexts. The generated LLVM-IR is executable without further transformations and only few, lightweight optimizations are required for high-performance LLVM-IR code.

Currently, x86-64 and 64-bit RISC-V are supported as source architectures; the general concepts also apply to other common architectures and the library was designed to be easily extensible.

- Instrew, a framework for performance-oriented **LLVM-based Dynamic Binary Translation (DBT)** and heavy-weight **Dynamic Binary Instrumentation (DBI)**.

  The Instrew framework pursues a new approach of solely basing Dynamic Binary Instrumentation (DBI) on LLVM, without the use of other IRs outside of LLVM. The transformation of machine code to LLVM-IR relies on Rellume, which allows for processing whole functions at once, enabling for more optimized code transformations and leading to a better usage of the LLVM optimization infrastructure. As machine code is lifted to target-independent LLVM-IR, changing the architecture back-end allows for Instrew to be used as flexible Dynamic Binary Translation (DBT) system with cross-Instruction Set Architecture (ISA) instrumentation capabilities — this particular aspect is particularly advantageous with new and highly customizable ISAs like RISC-V [RIS19].

  Additionally, Instrew implements a novel **client–server approach for DBT**, where the entire translation process happens in a separated server process while the client is only responsible for executing the translated code. This not only increases flexibility in distributed systems, where a single server can rewrite code for many clients, but also can reduce the translation overhead with a permanently running server that caches translated code fragments. Moreover, this design also simplifies research of new techniques for binary translation or transformation: as a consequence of the strict encapsulation of the rewriting process in a separate process, the client can be used as a generic and performance-focused base for developing new approaches.

  Performance results on the SPEC CPU2017 benchmark suite [Sta21] show that Instrew has an average overhead of only 59% over the native execution on x86-64, which is significantly less than the overhead of comparable systems, including the widely used QEMU emulator [Bel05] with 1044%, the LLVM-based DBT system HQEMU [Hon+12] with 135%, and the Valgrind [NS07b] DBI framework with 547%.

- BinOpt, a library for **application-guided run-time binary optimization** and specialization.

  This work provides a new library enabling application or library developers to dynamically specialize machine code on run-time-only information, for example, configuration parameters or results of previous computations. As code fragments and information used for specialization are explicitly specified by the application, this approach allows for an effective usage of developer knowledge and introduces no further profiling overhead during program execution. The operation on machine code not only allows subsequent optimization of dynamically optimized code, also but leads to an easier integration with existing build systems and enables re-use of optimization and code generation work done during the initial compilation step.

  The library provides a **generic API to configure and trigger specializations** from the application. At the same time, the API provides sufficient flexibility for the actual optimization back-end to implement different approaches. In particular, in addition to an LLVM-based optimizer (DBLL), also the binary function optimizer Drob [Hil19] and the tracing binary optimizer DBrew [WB16] are supported.

  A comparison of the three supported optimization approaches shows the differently chosen trade-offs between the performance of the optimized code and the overhead of the optimization process itself. Using BinOpt for a widely used optimized image processing library shows that this approach can lead to significant performance improvements of up to 67%.

1.5 Outline

This thesis is structured as follows: Chapter 2 gives an overview on program execution, binary rewriting, and machine code generation. Chapter 3 then describes the Rellume library for lifting machine code to LLVM-IR in detail. Afterwards, Chapter 4 presents and evaluates the Instrew framework in the context of dynamic binary translation; Chapter 5 covers the use case of program instrumentation with Instrew. Next, Chapter 6 demonstrates the use of dynamic binary rewriting for application-guided performance optimization. Finally, Chapter 7 concludes this thesis with a summary and an outlook on future work.

2 Background

In this chapter, a broad overview on program execution and code transformations with relation to machine code will be given. Afterwards, the field of binary rewriting will be described in more details with particular focus on dynamic approaches operating transparently to the rewritten program. Finally, the prevalent strategy for generating efficient machine code will be described together with a brief outline of the widely used LLVM optimization and code generation framework.

2.1 Program Execution

Generally, two main strategies for executing programs exist [Ayc03]: the first strategy is program *interpretation*, where the program is executed directly statement-by-statement. Here, the CPU only executes instructions from the interpreter itself. The second strategy is *compilation*, where a program is translated to machine code, which is then natively executed by the CPU. Prominent examples for a plain interpreter are shells like Bash [Fre20], where programs ("shell scripts") are executed line-by-line; examples for plain compilers include GCC [Fre21] and the Go compiler [Goo21], where programs are translated to native machine code.

These strategies are not exclusive, however, and various combinations exist by initially compiling the program to some sort of intermediate code representation, for example *bytecode*, which is later interpreted. The CPython interpreter [Pyt21b], for example, compiles programs into bytecode, which is then interpreted for the actual program execution [Pyt21c].

Further, there may also be multiple compilation stages: for example, in the JavaScript engine V8 [V8 17], where parts of the the bytecode are compiled further to machine code during program execution. In fact, also many optimizing compilers internally have numerous different code representations in which different optimizations are applied.

Different approaches of compilation and execution strategies have their respective advantages and disadvantages; the most critical aspect regarding execution efficiency is that programs that are eventually compiled to native machine code for the CPU generally execute faster [Ayc03] as there is no interpretation overhead involved. Thus, in the following, approaches that involve interpreters will not be covered further.

2.2 Code Transformation

Program code is given in a typically well-defined representation. Such representations differ in their appearance, which is often strongly influenced by their usage. The most extreme, but also most important, case is machine code, which generally is designed to be implemented in hardware and is the final form in which program code can natively be executed. Therefore, many kinds of code transformations serve the goal of eventually reaching machine code. An overview on the processes relating different levels of code representation is shown in Figure 2.1.

The process of translating code from a higher-level representation to a lower-level representation — for example, machine code or some kind of bytecode — is generally referred to as *compilation* or *lowering*. There is no constraint on the points in time for the compilation stages: an initial compilation stage may translate a program to bytecode, which immediately before its execution gets compiled further to actual machine code. Compilation at the time of program execution is referred to as *Just-in-Time (JIT) compilation* or *dynamic compilation*, compilation before program execution is referred to as *Ahead-of-Time (AoT) compilation* or *static compilation*.

However, code transformation processes may also occur in the other direction where machine code is translated to a higher-level code representation; this is broadly referred to as *de-compilation* or *lifting*. This kind of transformation is commonly used for analysis of compiled programs where the source code is not available, for example in malware analysis [van07]. De-compilers typically operate statically as they usually do not intend to immediately execute the progam. Note that clearly distinguishing "higher-level" and "lower-level" representations might not always be possible, for example in cases where bytecode is used for both representations.

The case of transforming machine code to machine code either for the same or a different CPU architecture is referred to as *binary rewriting*, which typically incorpo-

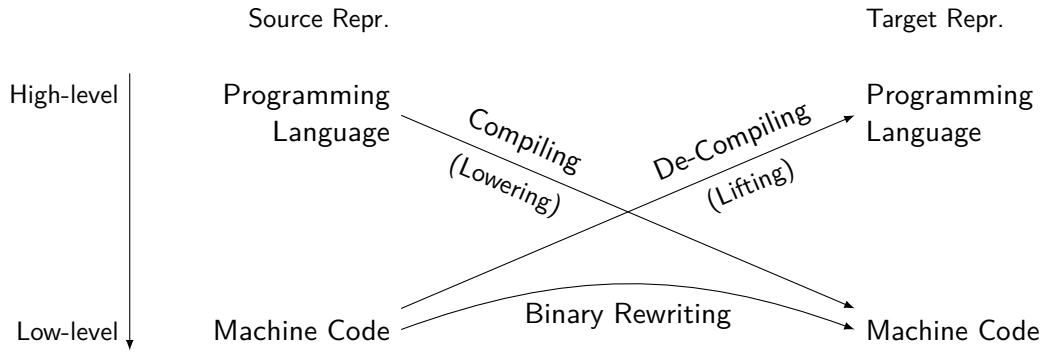


Figure 2.1: *Overview on the naming of code transformation processes between different levels of code representation. These transformations can also involve multiple steps with different layers of code representations in between.*

rates lifting *and* subsequent lowering steps. Like compilation, binary rewriting can be done either statically before program exection or dynamically during program execution.

2.3 Binary Rewriting

There are three main goals for performing binary rewriting: compatibility, behavioral changes, and optimization. The purpose of program analyses is typically accompanied with behavioral modifications, for example, to extract additional profiling data, and, therefore, is not covered separately hereafter.

Most systems that perform binary rewriting operate *transparently* to the rewritten program, that is, the transformed program is not explicitly aware of the rewriting process[1]. Few systems for (dynamic) binary rewriting can also be controlled or influenced by the application itself, which will be referred to as *application-guided* binary rewriting. Given that a large majority of binary rewriting systems operate transparently, only such approaches will be covered in this section; application-guided approaches will be discussed later in detail in Chapter 6.

2.3.1 Rewriting Goals

Following the three main goals, binary rewriters can be classified among the following three non-exclusive categories (see also Figure 2.2):

- *Binary Translation:* execute a program compiled for one source CPU architecture on a potentially different target CPU architecture, without behavioral

[1]There are several ways for an application to detect the static or dynamic rewriting process, rendering these techniques unusable for security purposes [Kir+18].

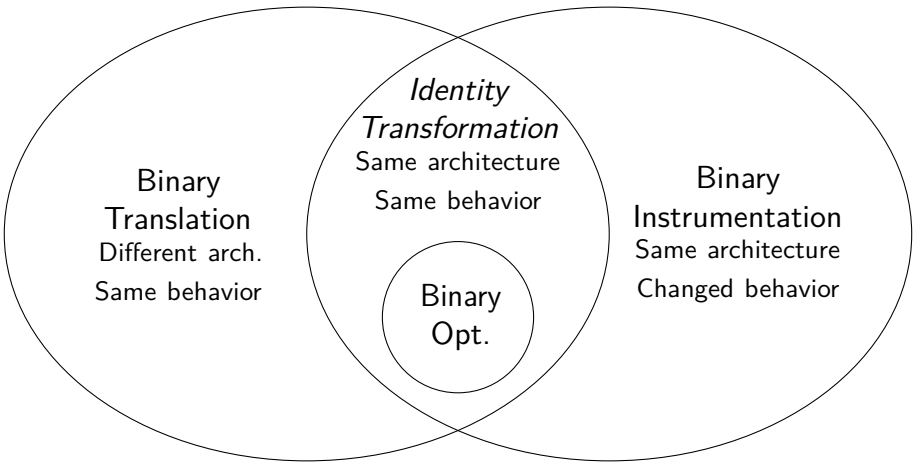


Figure 2.2: *Illustration of the three categories of binary rewriting and their relation.*

changes. The source and target architectures may also be identical and only diverge in available ISA extensions, for example, for executing a program when a required ISA extension is missing on the target CPU. As a special case, the source and target architectures may also be completely identical, in which case the rewriting process becomes an identity transformation.

Example: QEMU [Bel05]

- *Binary Instrumentation:* modify program behavior; for example, to add additional program integrity checks, profiling routines, or improved security enforcement. The source and target architectures are usually identical, but can also be different. A special case is the *null*-instrumentation, where no modifications are performed.

  Example: Valgrind [NS07b]

- *Binary Optimization:* optimize a program without behavioral changes. The typical optimization objective is an improved execution time, but a rewriting system can also aim for other optimization criteria like a size reduction of the program. The target architecture is typically a superset of the source architecture, for example, to enable usage of more recent available ISA extensions. While binary optimization can be technically seen as a special case of binary instrumentation and translation, the goals, and therefore also the rewriting strategies, are fundamentally different; binary instrumenters and translators primarily focus functionality with performance being only a secondary concern.

  Example: Dynamo [BDB00]

For all three categories, the rewriting process can be either done *statically*, where modifications target the executable file containing the machine code, or *dynamically*, where modifications target the machine code loaded into memory for execution. Static rewriting generally has a lower performance impact, but is also more difficult to implement as machine code has to be distinguished from data and all potential jump targets, especially from indirect jumps, have to be identified correctly. Furthermore, handling code generated dynamically from the rewritten program during its execution is impossible. On the contrary, dynamic rewriting avoids this problem and also allows for handling dynamically generated code, at the cost of rewriting overhead incurred during program execution. [Ang06; Pri19; Net04]

Therefore, many widely used frameworks [NS07b; Bel05; Luk+05; BGA03; BH00] for binary rewriting perform dynamic rewriting. In the following, the generic architecture of a transparent dynamic binary rewriting framework will be described.

2.3.2 Transparent Dynamic Binary Rewriting

A transparent user-level dynamic binary rewriting system generally operates as follows (cf. also Figure 2.3): initially, the rewriter, also referred to as the *host*, is loaded into memory by the operating system, which in turn loads the program to be rewritten, also referred to as *guest* program, into its own address space. Additionally, the rewriter sets up a virtual stack for the guest program, which contains the command line arguments, environment variables, and the auxiliary vector for additional information on the program execution, and initializes the state of the emulated guest CPU, notably by setting the emulated stack pointer to the virtual stack and the emulated program counter to the entry point of the guest program. The initialization is completed by starting the main execution loop of the rewriter.

In the main execution loop, the rewriter inspects the virtual program counter and decodes a block of code from that address, which is typically lifted to some sort of IR. At this point, the behavior of the code can be modified for instrumentation purposes. Afterwards, the possibly modified code is then compiled to machine code of the host architecture and stored at a different address. This newly rewritten code is then executed and thereby modifies the virtual CPU state, in particular the new program counter indicating the next instruction to be executed. The main loop then repeats the rewriting–execution process until the guest program terminates.

Optimizations To prevent redundant rewriting of the same code, rewriters commonly install a *code cache*: rewritten code fragments are cached and reused when

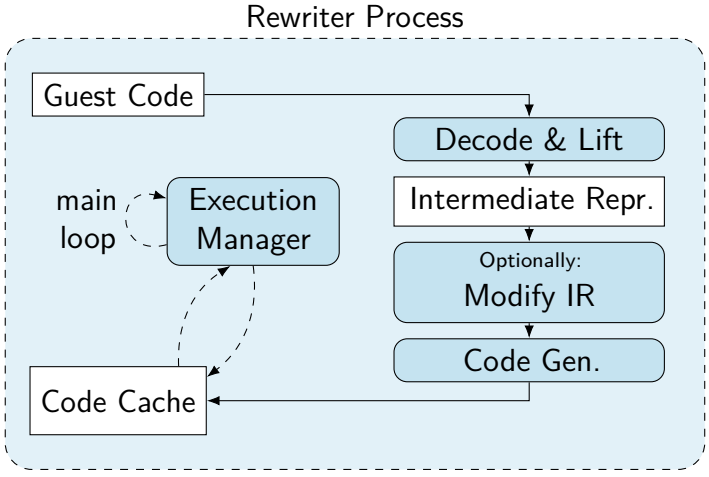


Figure 2.3: *Architecture of a typical Transparent Dynamic Binary Rewriting system. Guest code is lifted to an Intermediate Representation before modifications can be applied; the newly generated code is then stored in a code cache for execution.* Figure slightly modified from [ES20a].

the execution loop encounters a guest address that was transformed previously.

Additionally, to reduce the overhead of dispatching to code fragments, rewritten code blocks can be *chained* by patching the code fragment to directly jump to the subsequently following block, avoiding the extra step of going through the main loop. This optimization, however, is only possible when the target is definitely known and therefore is not applicable for indirect jumps or function returns.

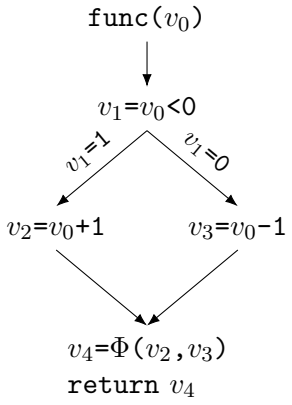
It is also possible to avoid the extra step through the main loop for function returns using a *return-address stack*: whenever a function call is encountered in the guest program, the guest return address and the host address of the corresponding rewritten code is pushed to a shadow stack. When a guest function return is executed later on, the execution can directly continue at the host address after verifying that the shadowed guest address is indeed correct. This verification step is critical, as calls and returns do not have to match at instruction-level, for example, in case of exceptions or `setjmp/longjump`.

Moreover, the rewriting system can identify *traces* of subsequently executed blocks by collecting profiling data of rewritten code and may decide to apply further and possibly more expensive optimizations on frequently executed (*hot*) traces.

Common Rewriting Approaches The key factor for a binary rewriter is the IR used for representing the guest code. One important implication is the size of the blocks that are rewritten at once. For this, two common options have evolved: the first option are *basic blocks* (e.g., DynamoRIO [BGA03]), which is a sequence of instructions with a single entry point and is terminated by the first instruction which potentially modifies control flow, for example a conditional branch. The second option are *superblocks* (e.g., Valgrind [NS07b]), which are similar to basic blocks but only terminate at the first unconditional control flow instruction. Consequently, while a basic block has a single exit point, a superblock can have multiple side exits from conditional branches contained in the instructions. The key advantage of both options is simplicity: basic blocks or superblocks can be easily identified in machine code, allow for a straight-forward analysis without the need of handling an arbitarily complex control flow, and for the same reason also simplify generation of new machine code.

Another implication of the IR is the abstraction from the guest architecture: a binary translator generally employs an architecture-independent IR — during the lifting process, architecture-specific peculiarities are abstracted to simplify code generation for the host architecture. In contrast, a binary rewriter that demands guest and host architecture to be the same additionally has the option of using an architecture-specific IR, maintaining the original guest instructions and modifying them only when actually required.

While using an architecture-independent IR simplifies behavior modifications and reduces effort when retargeting the rewriting system to a new architecture, using an



(a) *Graphical representation.*

```
1  define i32 @func(i32 %0) {
2    %2 = icmp slt i32 %0, 0
3    br i1 %2, label %3, label %5
4  3:
5    %4 = add i32 %0, 1
6    br label %7
7  5:
8    %6 = sub i32 %0, 1
9    br label %7
10 7:
11   %8 = phi i32 [%4, %3], [%6, %5]
12   ret i32 %8
13 }
```

(b) *LLVM IR representation.*

Figure 2.4: *An example for a function in Single-Static Assignment (SSA) form. Every value is assigned exactly one instruction; values from different incoming blocks can be merged using* Φ *nodes.*

architecture-specific IR typically incurs less overhead during rewriting at the cost of increased difficulty for analyses and transformations, as architecture-specific details and limitations have to be respected.

2.4 Machine Code Generation

Generation of efficient machine code is an important challenge, not only in the context of static compilers, but also for Just-in-Time (JIT) compilers and binary rewriters. After program optimizations, a program is typically represented in a higher-level IR, which has to be lowered to efficient code for execution on the target CPU.

2.4.1 Single-Static Assignment Form

Optimizing compilers typically employ an IR based on the *Single-Static Assignment (SSA)* form [RWZ88; Cyt+91]: a value can be assigned the result of exactly one instruction. Whenever a basic block in the control flow graph has multiple incoming edges, a special *phi node* can be used to select a value depending on the actually taken incoming edge. An example of a function in Single-Static Assignment (SSA) form is shown in Figure 2.4a. This form has main benefit of simplifying program analysis, as only the actual data dependencies are considered, therefore facilitates further optimizations, for example the elimination of redundant computations and dead code, and generally eases program modifications during that process.

Lowering from a compiler IR to machine code generally involves three steps [Bli16]: first, corresponding machine code instructions for IR instructions have to be selected, which typically requires lowering instruction sequences for efficiency. Second, the instructions have to be scheduled, determining the execution order of the previously selected instructions. And third, the storage for the individual values has to be allocated, typically including allocating registers, stack frame slots, and inserting instructions for data movement when required. However, finding an efficient instruction sequence is a complex problem and the problem of determining the optimal instruction selection is NP-complete.

2.4.2 Compiler Infrastructures

The objective of avoiding redundant implementation of optimizations and analyses led to the development of compiler infrastructures, which reduce coupling between a language-specific front-end, the program optimizations, and the architecture-specific code generation back-end. As an additional benefit of this separation, such compilers can be ported to new architectures more easily, as many existing components can be reused.

For example, the GNU Compiler Collection (GCC) [Fre] uses up three different IRs, in which different analyses and transformations are be performed. GCC is, however, generally designed to operate as static compiler and therefore heavily relies on external tools for assembling and linking. While a library for usage in JIT contexts is provided in recent versions [Mal20], it has only a rather limited API, relies on the availability of other toolchain components during execution, and only receives comparably little development efforts.

The demand of a flexibly usable code generation library led to the development of the LLVM framework [LA04]. Compiler front-ends like Clang [LLV20a] transform the source language into the main code representation of LLVM, the *LLVM-IR*. The LLVM framework then provides many optimization passes and a machine code generator. The entire process of optimization and code generation, however, can also happen at run-time using the JIT execution engine. In contrast to GCC, the entire framework is designed to be used as a library and therefore provides a powerful and easily usable API.

Today, LLVM provides a high quality optimization and code generation framework for several target architectures and is therefore used by several compilers [LLV20a; Wal21; Arm20] and even graphics drivers [Pau21].

LLVM Intermediate Representation The LLVM-IR [LLV20b] is organized in a *module*, which contains global variables as well as function declarations and definitions. Function definitions are represented in SSA form and consist of one or more basic blocks. A basic block is a sequence of LLVM-IR instructions, ended by a terminator instruction, which is usually a (conditional) branch to other blocks or a

function return. LLVM-IR features a RISC-like instruction set without implicit side effects. In contrast to a CPU instruction set, however, LLVM-IR is strongly-typed, including variable-length integer, vector, and structure types, and models all control flow transfers explicitly. Function call instructions are ordinary instructions that cannot modify the control flow of the caller. An example function is shown in Figure 2.4b.

More complex or target-specific operations are exposed as intrinsic functions, which can be used like normal function call target, but are treated specially during optimization and lowering. An example for an intrinsic function is `llvm.floor` for floating-point values. If this operation is natively supported by the target architecture, this intrinsic gets lowered to a corresponding instruction; otherwise a function call to the standard library is generated. Other intrinsics include operations like `memcpy`, functions for accessing variable arguments, and target-specific vector operations. [LLV20b]

2.5 Summary

The discussions above showed that code generation to natively executable machine code is important for performance. Machine code can be generated from a higher-level code representation, but can also be derived from existing other machine code using binary rewriting. In particular, dynamic binary rewriting during the execution of a program is a widely used technique for achieving compatibility through translation, behavioral modifications through instrumentation, and also for performance optimizations. Many widely used systems, however, rely on low-level code representations to reduce the performance overhead incurred by the rewriting process, and therefore do not benefit from advances in optimizing compiler infrastructures like LLVM. Incorporating an existing compiler infrastructure into a dynamic binary rewriting system requires a performance-focused transformation, lifting machine code to the compiler IR. Therefore, the next chapter will present Rellume, a performance-focused library for lifting machine code to the IR of the LLVM infrastructure.

3 Rellume: Lifting Machine Code to Performant LLVM-IR

This chapter describes an approach to lift machine code to LLVM-IR with the focus on performance of the lifting process and the generated LLVM-IR code. This is implemented in a library named *Rellume* and provides the foundation for further applications described later in this thesis: dynamic binary translation (Chapter 4), dynamic binary instrumentation (Chapter 5), and dynamic binary optimization (Chapter 6.7.3).

Publication Information

The approach described in this chapter was previously published in [ES20a] with further extensions described in [EOS21], in particular the support for multiple architectures and the call–return optimization.

3.1 Motivation

Extracting semantics from machine code is a common problem, not only for the purpose of reverse engineering compiled programs, but also for retargeting binaries for other architectures, instrumenting the program with profiling or debugging code, and post-optimization of binaries. Doing such operations directly on machine code is impractical for several reasons: first, there are encoding limitations that have to be fulfilled, causing significant effort if even minor program changes cannot be encoded in the same instruction. Second, hardware architectures typically have a fixed numbers of registers, requiring major program refactoring for enhancements or workarounds for encoding limitations if no unused registers are available. Third, such operations are strongly tied to a specific architecture and cannot be ported easily.

For these reasons, machine code is usually lifted to a more abstract IR. Such IRs differ in their degree of abstractions implied by their purpose. For example, a binary instrumenter which only targets a single architecture may keep instructions in an architecture-dependent format, enabling fast lowering to new machine code (e.g., DynamoRIO [BGA03]), whereas a binary translator to a different architecture strongly benefits from an architecture-independent code representation (e.g., QEMU [Bel05]).

However, lower-level IRs have another major disadvantage: while they may allow for complex transformations, implementing *optimizations* to reduce the overhead of the transformation requires porting many optimizations from standard compilers to the IR. Since this is a very large effort, current systems typically only perform simple optimizations (e.g., QEMU [Bel05], Valgrind [NS07b]), or none at all (e.g., DynamoRIO [BGA03]).

This drawback can be circumvented by lifting machine code directly to the high-level IR of a standard compiler framework, which already provides several analyses, optimizations, and a code generation back-end. One state-of-the-art compiler infrastructure is the LLVM [LA04] framework, which features a high-quality code optimizer and machine code generator for several architectures. This framework is used for many compilers, e.g., Clang [LLV20a] or the Arm Compiler for Linux [Arm20]. The LLVM-IR is not only suitable for lowering C code, but also allows to express complex semantics such as vector operations. As the LLVM-IR was designed to support large code transformations, it comprises a very extensive API for modifications of the IR. Moreover, LLVM was also designed to be used as JIT compilation framework, which enables use for run-time code generation as required for dynamic binary instrumentation or optimization systems.

Consequently, LLVM is an interesting IR for analyzing and modifying machine code, especially for later execution of the modified code using the built-in JIT compiler.

3.2 State of the Art

Several approaches for lifting machine code to LLVM-IR have been proposed already. Many of these approaches primarily serve the goal of static analysis of binary files, but do not emphasize the possibility that the lifting process can also occur dynamically during program execution. For such static approaches, the performance of the lifting and subsequent optimization process has a lower priority. Other approaches are specifically designed for the dynamic use-case, where the translated is executed immediately and the lifting time is of concern. Table 3.1 gives an overview of the lifters covered here.

Internally, several lifters rely on Tiny Code Generator (TCG) [Bel+19], the IR internally used by QEMU [Bel05], to initially abstract the machine code to a low-level, but architecture-independent, code representation. This way, supporting several architectures is possible with only low effort, since the mapping of the ISA semantics to a simplified representation is performed using TCG. However, some limitations of TCG, like basic block granularity or the lack of proper support for floating-point operations, require additional workarounds during lifting to LLVM-IR.

Table 3.1: *Overview of approaches lifting machine code to LLVM-IR, showing maintenance state, whether the lifted code is executable, whether the lifter can be used in a static or dynamic context, supported architectures, and the overall intention. This table is not complete; several unmaintained lifters or lifters with only partial coverage of a single architecture are excluded. Inspired by a table in the McSema documentation [Tra21].*

Name	Maint.	Exec.	S/D	Architectures	Purpose
S2E [CKC11; Cyb18]	✓	✓	Both	x86, ARM, ...	Analysis
HQEMU [Hon+12]	—	✓	Dyn.	x86, ARM	Emulation
DBILL [Lyu+14]	—	✓	Dyn.	x86, ARM	Instrumentation
McSema [Tra21]	✓	✓	Stat.	x86, ARM, ...	Analysis, Rev.Eng.
Fcd [Clo16]	—	—	Stat.	x86-64	Decompilation
RetDec [KMZ17]	✓	✓	Stat.	x86, ARM, ...	Decompilation
Rev.ng [DFA18]	✓	✓	Stat.	x86, ARM, ...	Analysis
MCTOLL [YS19]	(✓)	✓	Stat.	ARM	Analysis
Dagger [Bou+17]	—	✓	Both	x86-64	Generic
Reopt [Gal21b]	✓	✓	Stat.	x86-64	Recompilation

3.2.1 Static Lifting Approaches

Static approaches can be grouped by their main purpose: analysis of compiled binary files, potentially performing modifications and generating a new binary; and decompilation to human-readable source code.

Binary Analysis and Modification McSema [Tra21] is a lifter for x86(-64), AArch64, and SPARC, supporting a wide range of executable formats and ISA extensions. The main focus of this project is binary analysis and reverse engineering, but the lifted LLVM-IR code is also executable, e.g. for fuzzing. The lifting strategy is as follows: first, IDA Pro[1] is used functions are identified in the binary and recover the Control Flow Graph (CFG) of the functions. Second, for each function a corresponding LLVM-IR function with the recovered structure of basic blocks is created. Third, the individual instructions are lifted to LLVM-IR code; this process is implemented in a separate library called Remill. And fourth, the resulting function is optimized and helper structures used for lifting are removed.

A lifted function generated by McSema takes three parameters [Tra19a]: a pointer to the state of the virtual CPU, the program counter, and a pointer to an object representing memory. These states are modified as part of the execution of the lifted function. During lifting, specific operations are not lifted to native LLVM-IR constructs, but to so-called Remill intrinsics, which act as helper functions. They are used for representing atomic operations, system calls, and all memory accesses

[1] IDA Pro [Hex20] is a commercial disassembler.

of the lifted code [Tra19b; Tra20]. During later optimizations, some of these helper functions are removed again. To avoid strong divergence between the original code and the lifted code as a consequence of optimizations, active measures to prohibit such transformations are taken, such as modifications of data structures to prevent automatic elimination of memory accesses [Tra20].

Dagger [Bou+17] is a modified version of LLVM that includes a lifter from x86-64 to LLVM-IR for dynamic and static usage, without relying on external components. The project, however, appears to be in an early state and discontinued.

MCTOLL [YS19] is a more recent project focusing on static analyses sharing the same goal not to rely on external components for disassembling and CFG reconstruction. Currently supported architectures are AArch32 and x86-64 without SSE extensions.

S2E [CKC11] is a generic framework for binary analyses, internally lifting TCG basic blocks to separate LLVM functions [Cyb18]. These can be combined later, for example, to using control flow analysis data obtained through McSema as done with RevGen [Cyb18]. The S2E lifter from TCG to LLVM-IR can be used as a separate library also in a dynamic context.

Reopt [Gal21b] is a tool for lifting x86-64 binary files to LLVM-IR and was originally designed for the purpose of optimizing binaries [Gal], but generally focuses on code analysis and recompilation. Internally, the actual machine code is extracted from the binary file and functions are recovered. Then, the machine code of the functions is lifted to an architecture-independent Domain-specific Language (DSL) using Macaw [Gal21a; Gal17], which is a framework serving binary analysis and supports symbolic execution of the DSL. From this DSL, the code is subsequently lifted to LLVM-IR; indirect calls appear to be unsupported.

Rev.ng [DFA18; Gus+19] is a tool for static binary analysis and translation, which lifts all code first to TCG and from there into a single function. The code execution is governed by a dispatcher block, which jumps to the basic block matching a given program counter value.

Decompilers Fcd [Clo16] is a decompiler which internally lifts machine code to LLVM-IR before emitting the code as a C-like output. The project supports x86-64 only and appears to be no longer maintained.

RetDec [KMZ17; Křo14] is a more generic decompiler supporting more file formats and architectures. After abstracting file format specifics, the control flow is analyzed and functions are identified. Then, the machine code of the executable is lifted to LLVM-IR into a single function; however, more complex instructions are not handled explicitly and are implemented using helper functions. Afterwards, the lifted code is optimized at LLVM-IR-level, including recovery of data types, local variables, and function signatures. Finally, the LLVM-IR code is lifted further to C, during which further transformations for more readable output are performed. Registers

values are stored in global variables, and the LLVM-IR functions generated during the lifting process have no parameters.

3.2.2 Dynamic Lifting Approaches

Besides S2E [CKC11] and Dagger [Bou+17], which both support static and dynamic operation, only very few dynamic lifting approaches that provide a sufficient coverage of a widely used architecture are known. Many of them integrate with QEMU: LnQ [Hsu+11] is a QEMU-based approach that lifts directly from machine code to LLVM-IR with basic block granularity, but was superseded by HQEMU [Hon+12] with a TCG-based approach. The other QEMU-based approaches simply lift TCG code further to LLVM-IR for enhanced performance [CC10; Jef09; Hon+12; Lyu+14] or for dynamic binary analysis [WLK13], either using a custom lifter or with the S2E lifter library.

3.3 Rellume Lifting Approach

For a performance-oriented lifter designed for use in dynamic contexts, three main goals exist: first, the derived LLVM-IR must be "good" and match the expectations of the existing optimization infrastructure, which is generally targeted at code derived from higher-level languages. Second, the lifting process itself and also the subsequently needed optimizations are required to have a low overhead in a dynamic context. Thus, lifting has to be as straight-forward as possible and avoid unnecessary transformations. Generally, the code generated during lifting should only require few and lightweight optimization passes to satisfy the first goal of "good" IR code; and third, the generated LLVM-IR should make use of built-in LLVM constructs where possible, as these are tightly coupled with the rest of the infrastructure and consequently allow to make effective use of existing lowering strategies.

To satisfy these goals, the most reasonable approach for lifting machine code to LLVM-IR is to perform this step directly without an additional IR. This not only eliminates an extra transformation step, but also avoids restrictions on smaller code granularity and limited expressiveness as exhibited by TCG. A direct lifting strategy, in contrast, also allows for mapping more complex control flow efficiently to LLVM-IR and maintaining a direct relationship with the original code, which is especially relevant for instrumentation applications. Furthermore, extensive use of built-in LLVM functionality not only ensures a good integration with other parts of the framework, but also enables retargetability to compile and execute the lifted code on different architectures.

We implemented this direct lifting strategy in the novel lifting library *Rellume*[2], which lifts machine code with function granularity to target-independent LLVM-IR.

[2] `https://github.com/aengelke/rellume` licensed under LGPLv2

The library is separated into an architecture-independent part, which implements the generic lifting approach, and an architecture-dependent part, which supplies the required information about the source architecture and the instructions semantics. In this section, the architecture-independent lifting approach will be described; the architecture-dependent parts and required modifications for adding a new architecture will be covered in Section 3.4.

3.3.1 Lifting Stages

Lifting an arbitrary sequence of machine code to LLVM-IR is done in three separated stages: decoding, lifting instruction semantics, and finally linking basic blocks. Figure 3.1 shows an example for the lifting result.

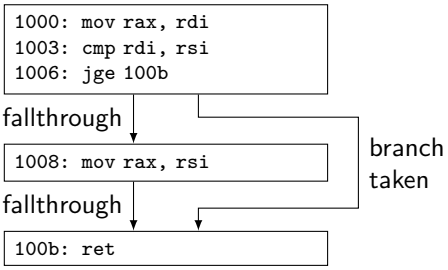
Stage 1: Decoding. Starting from an externally supplied address, machine code from that point on is decoded. The decoding procedure follows direct jumps and both targets of conditional branches, reconstructing the CFG consisting of basic blocks. When creating the division into basic blocks, it is ensured that every decoded instruction is part of exactly one basic block, to avoid having redundant code in the lifted code. The decoder cannot continue decoding after indirect jumps and instructions for function return, as the next address is not statically known. Furthermore, the decoder also does not follow function calls, even if the target address is known. This has two reasons: first, functions are designed to be called from several places, and decoding into a sub-function would result in redundant decoding and further processing. Second, if inlining a function had a significant impact on the run-time performance, it would have been already done by the compiler. There is one exception where decoding continues *after* the function call, this is described later in Section 3.3.7.

As further optimization, the use of jump tables, as commonly generated for `switch` statements, could be detected at this point to possibly continue decoding after indirect jumps; this is left as future work.

Stage 2: Lifting Instruction Semantics. After all instruction have been decoded, a skeleton LLVM-IR function is created and initially filled with basic blocks for each basic block of the reconstructed CFG. Then, for each instruction, LLVM-IR code is emitted individually into the corresponding basic block. To reduce the overhead of emitting unused instructions, some parts of the LLVM-IR code are generated only on demand, with the aim of improving not only the performance of the lifter, but also reducing the function size for later optimization passes. In particular, many unnecessary cast instructions can be avoided.

```
1000: mov rax, rdi
1003: cmp rdi, rsi
1006: jge 100b
1008: mov rax, rsi
100b: ret
```

(a) *Initial code to lift. This function computes the maximum of two signed 64-bit integers given in* `rdi` *and* `rsi`*.*



(b) *Code after reconstruction of the CFG. The function is split into basic blocks.*

```
define void @func_1000(i8* %cpu) {
  %cpu.rsp_off = getelementptr i8, i8* %cpu, i64 40
  %cpu.rsp = bitcast i8* %cpu.rsp_off to i64*
  %cpu.rsi_off = getelementptr i8, i8* %cpu, i64 56
  %cpu.rsi = bitcast i8* %cpu.rsi_off to i64*
  %cpu.rdi_off = getelementptr i8, i8* %cpu, i64 64
  %cpu.rdi = bitcast i8* %cpu.rdi_off to i64*
  %rsp.0 = load i64, i64* %cpu.rsp, align 8
  %rsi.0 = load i64, i64* %cpu.rsi, align 8
  %rdi.0 = load i64, i64* %cpu.rdi, align 16
  br label %addr_1000

addr_1000:
  ; mov is replaced with SSA value propagation.
  ; flag computation eliminated by InstCombine.
  %cond = icmp slt i64 %rdi.0, %rsi.0
  br i1 %cond, label %addr_1008, label %addr_100b

addr_1008:
  ; mov is replaced with SSA value propagation.
  br label %addr_100b

addr_100b:
  %rax.0 = phi i64 [ %rsi.0, %addr_1008 ], [ %rdi.0, %addr_1000 ]
  ; ret loads return address from stack and increments rsp by 8.
  %rsp.0p = inttoptr i64 %rsp.0 to i64*
  %retaddr = load i64, i64* %rsp.0p, align 4
  %rsp.1p = getelementptr i64, i64* %rsp.0p, i64 1
  %rsp.1 = ptrtoint i64* %rsp.1p to i64
  br label %endblock

endblock:
  %cpu.rax_off = getelementptr i8, i8* %cpu, i64 8
  %cpu.rax = bitcast i8* %cpu.rax_off to i64*
  %cpu.rip = bitcast i8* %0 to i64*
  store i64 %retaddr, i64* %cpu.rip, align 16
  store i64 %rax.0, i64* %cpu.rax, align 8
  store i64 %rsp.1, i64* %cpu.rsp, align 8
  ret void
}
```

(c) *Lifted LLVM-IR; for brevity the optimization pass* InstCombine *was already applied and the status flag values are discarded. The entry block loads required registers into SSA variables; subsequently following basic blocks correspond to basic block in the machine code. At the end, the updated register values are written back just before the function returns.*

Figure 3.1: *Example of Rellume's lifting procedure.*

Stage 3: Linking Basic Blocks. Once all basic blocks have been filled with instructions representing their architectural semantics, branches between the basic blocks are added and the phi nodes (cf. Section 2.4.1), which are used to propagate register values between basic blocks, are filled appropriately.

3.3.2 LLVM-IR Functions

The design of the LLVM-IR demands that all code is enclosed in functions, which closely follow functions in higher-level languages like C [ISO17]: a function can have several parameters and has a single return value. From the control flow perspective, a function consists of basic blocks, of which the first is the single entry point, and every basic block is terminated either by a branch to another basic block or a function return.

In machine code, however, there is no explicit concept of functions. Thus, when transforming a set of machine code instructions to LLVM-IR, it has to be enclosed in an artificial function, which only internally contains the semantics of the machine code. Functions created by the lifter therefore have the following structure: they do not return a value and take a single parameter to the state of the emulated CPU, which is modified throughout the execution of the function.

This *CPU state* contains all state relevant for modeling the guest CPU, involving the program counter, the state of general-purpose and vector registers, status flag registers, and special-purpose registers like the base address for the `fs`/`gs` segments on x86-64.

As the CPU state resides in memory, it is important to avoid load and store operations where possible and, therefore, keeping state in SSA registers where possible — this not only makes the code easier to analyze for later optimizations, but also avoids expensive analyses of memory aliasing. Consequently, the values of all registers are loaded from the CPU state once in the entry block and are written back when a consistent CPU state is required, for example, when the LLVM function returns. Within the function, the register state is propagated using SSA registers and PHI nodes.

3.3.3 Control Flow

Whenever a chunk of machine code is lifted, it is divided into basic blocks during decoding. These basic blocks are generally mapped to LLVM basic blocks, the control flow in machine code represented by jumps or fall-through is transformed into branch instructions terminating the LLVM basic blocks. If the target address of a jump is not decoded or not statically known, the function adjusts the program counter to the address of the next instruction and returns after updating the CPU state in memory. Once the address is known during the actual program execution, a new, separate lifting process has to get initiated.

Some instructions have a more complex behavior and cannot be represented in a single LLVM basic block. An example are the repeated string instructions of x86-64, which may loop or not be executed at all depending on a register value, or the division instruction, which may need a special case when the divisor is zero to avoid invoking undefined behavior in LLVM. Such cases require an architectural basic block to be represented by multiple LLVM basic blocks.

3.3.4 Registers

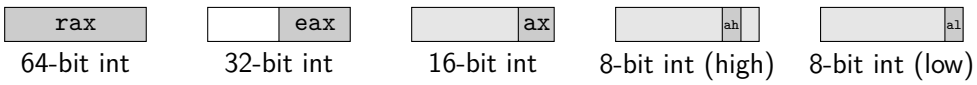
Internally, Rellume distinguishes three kinds of registers: general-purpose registers, vector or floating-point registers, and status flags (see Section 3.3.5). The concept of separating integer and floating-point registers matches the design of several widespread architectures, most notably x86-64 [Int20a], AArch64 [Arm18], Power [IBM17], and RISC-V [RIS19]. Obviously, the number and size of these registers may differ between architectures: while x86-64 has 16 general-purpose registers, AArch64 has 32[3]; and where AArch64 has four status flags, RISC-V has none at all. There may be further registers types — the x86-64 architecture, for example, also has legacy x87 Floating-point Unit (FPU) registers with a size of 80 bits — which are currently not represented, but can be easily added.

While at hardware-level registers are essentially a bitvector of constant size, this modeling is no longer sufficient when lifting to the strongly-typed LLVM-IR. A register can be accessed in different views of the underlying binary representation, which we refer to as *facets*. Such facets can vary in the subset of bits they represent — for example, on x86-64 the 64-bit register `rax` can also be accessed as 16-bit sub-register `ax` or as 8-bit sub-register `ah` — and in their data type — for example, a 128-bit vector register may be interpreted as `<4 x f32>`, as `<16 x i8>`, or as `i128`. Further examples are shown in Figure 3.2. Making such types explicit in the LLVM-IR code is an important factor with regard to the performance of the lifted code, as automatic type inference is not performed during optimizations at LLVM-IR level — compilers with type inference already perform such transformations at an earlier stage.

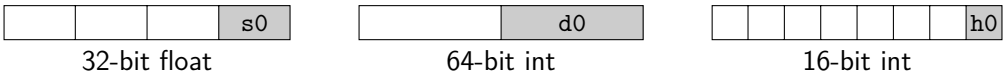
For every register, there is a so-called *native facet*, which is an integer type spanning all bits of a register. This facet must always be specified when a register is modified to ensure that the entire register content is known. All other facets can then be deduced from the native facet of, if written previously in a more suitable type, be deduced from a cached value. The actually used facets for register accesses depend on the lifted machine code instructions.

General-purpose registers usually have the size of a pointer — that is, 64-bit on 64-bit architectures — and are commonly used for scalar integer data of different sizes, or plain data movement, or for pointers. Consequently, general-purpose

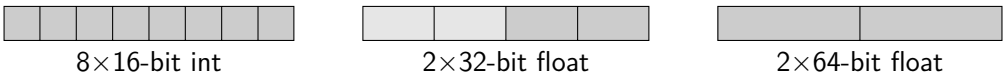
[3]The stack pointer `sp` and `x0`–`x30`, not counting the zero register.



(a) *Examples for scalar facets (dark gray) of general-purpose registers for x86-64: 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. AArch64 only uses a 64-bit and a 32-bit facet, which zeroes the upper part of the register; RISC-V64 in contrast sign-extends 32-bit values to 64 bits.*



(b) *Examples for scalar facets (dark gray) of vector registers on AArch64. Whether untouched parts are left unmodified or set to zero depends on the instruction.*



(c) *Examples for vector facets of vector registers. The vector element type and the vector size depend on the instruction.*

Figure 3.2: *Overview of different register facets. Registers can be accessed as scalar value or as a vector. The actual data type is defined by the machine code instruction, which also defines whether untouched parts of the registers are preserved or zeroed.* Figure is based on [ES20a].

registers have integer facets for accessible sub-registers — x86-64 requires a 32-bit, a 16-bit, and two 8-bit facets (see Figure 3.2a). Further, LLVM strongly distinguishes integer and pointer types to ease analysis of memory aliasing, whereas there is no difference in machine code. To enable lifting integer arithmetic from machine code to pointer arithmetic with `getelementptr` in LLVM-IR where possible, general-purpose registers also have a pointer facet, which is used for memory accesses.

Vector and floating-point registers are generally used for scalar or vector arithmetic of floating-point values or integers and therefore have a variety of facets with different vector sizes and vector element types (see Figures 3.2b and 3.2c).

3.3.5 Status Flags

The status flags register differs from conventional registers: the individual flag bits are almost always tested individually and the combined representation is used very rarely. In particular, x86-64 only has three instructions to read the combined representation (namely `pushf`, `lahf`, and `syscall`) [Int20a], AArch64 has only a single instruction for this purpose [Arm18].

Thus, within Rellume, status flags are represented as different single-bit registers

in the CPU state and therefore are also propagated as different values throughout the lifted code. The combined representation is only computed when actually required.

Between different architectures, the number and purpose of status flags differs; therefore, the number of flags is varying like the number of other registers. The x86-64 architecture, for example, has six status flags (sign, zero, carry, overflow, auxiliary carry, parity) indicating the result of an arithmetic operation and one control flag (direction) affecting the behavior of string instructions. In contrast, RISC-V has does not use the concept of status flags at all and encodes all condition operands directly in conditional instructions.

3.3.6 Memory Accesses

Memory accesses by the lifted code are generally lifted as regular `load` and `store` LLVM-IR instructions with the default (global) address space 0. To allow for an easy distinction of application memory accesses and memory operations of the lifter to access the CPU state, the latter are performed with pointers in address space 1. This has no further impact on the code generation process, but serves a hint to the alias analyses that the lifted code will never access the CPU state, allowing further optimizations regarding the CPU state.

As LLVM is designed for handling code generated from a high-level language, the `null` pointer is usually treated specially, in that arithmetic on this special pointer value is undefined. However, when lifting machine code, this is not necessarily the case; the address value zero may as well be used as address base in combination with a sufficiently large "offset". Hence, optimizations regarding the `null` pointer have to be inhibited on Rellume-generated functions. This is achieved by setting the `null-pointer-is-valid` attribute on such functions.

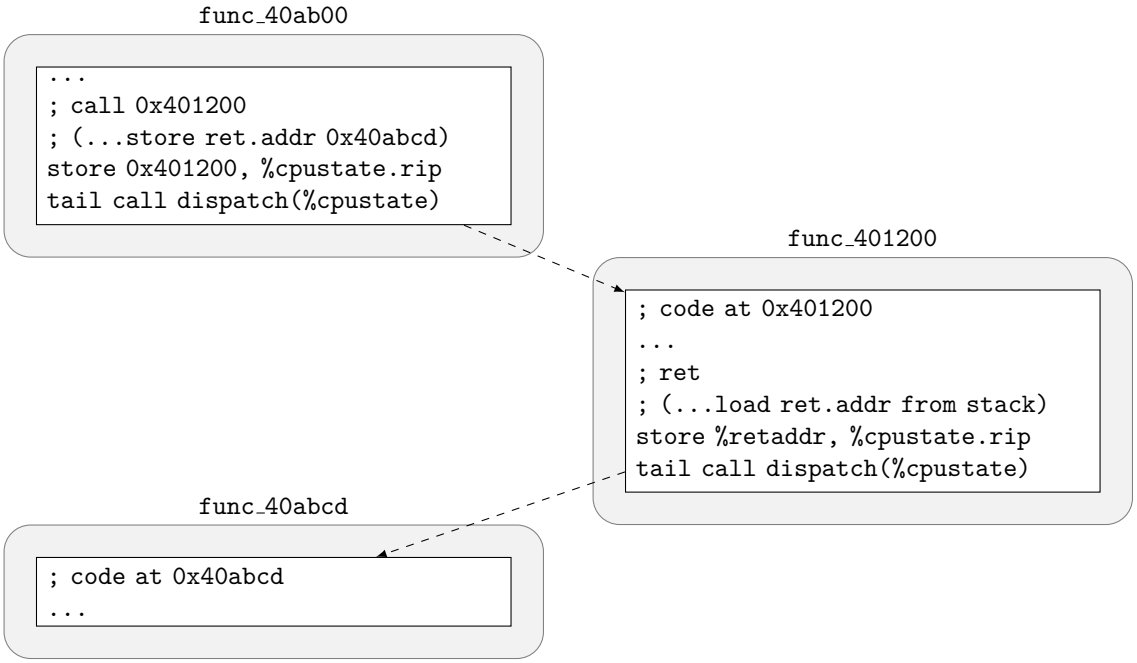
3.3.7 Further Optimizations

In addition to the previously described general lifting approach, Rellume implements three optimizations for improving the performance of the generated LLVM-IR code. These optimizations can be disabled individually, as they may have a negative impact on the overall performance in certain situations.

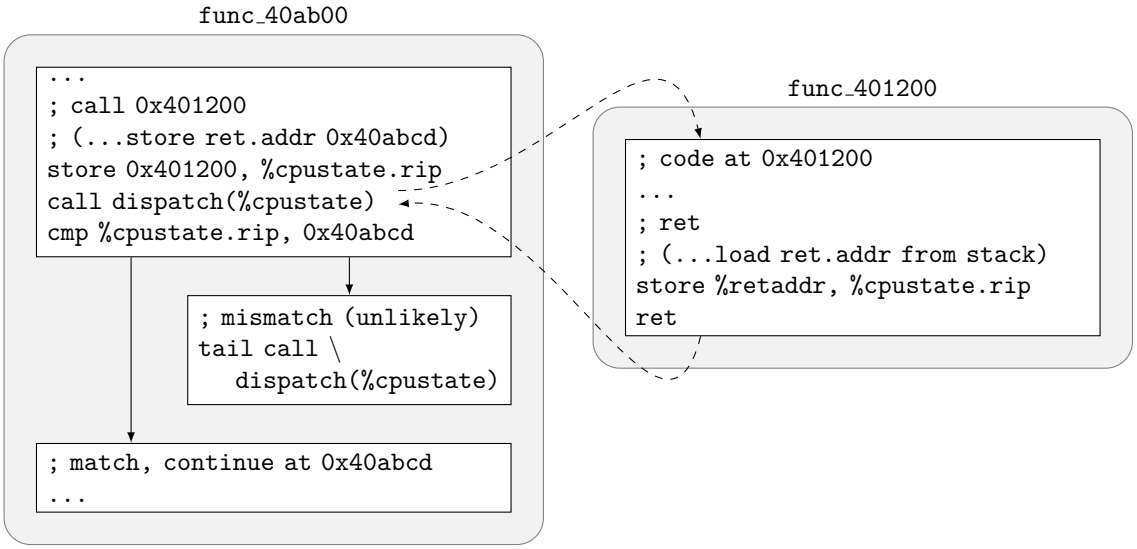
3.3.7.1 Call–return Optimization

In the lifting mode described in Section 3.3.1, the lifting process stops when encountering a function call, and the code of the called function as well as the code after the function return have to be handled separately. While this is a feasible approach, it neglects the fact that most function return ordinarily[4] and, after the function call finished, continue execution at the instruction following the call instruction.

[4] This is not always the case, even on compiler-generated code, e.g., with `longjmp` or exceptions.



(a) *Function call lifted without special call–return handling. The call is realized by storing the return address on the stack and the CPU state; the actual execution is realized by using a dispatcher. Likewise, the function return also jumps to the dispatcher with the return address fetched from the stack.*



(b) *Function call lifted with call–return mode. The function call is realized using a native call. When the called function returns using a native return, the new instruction pointer in the CPU state is compared with the expected return addresses. If the address are equal, the code after the call can be executed, otherwise a jump to the dispatcher is necessary.*

Figure 3.3: *Example for code lifted without and with the call–return mode. In call–return mode, fewer lifted functions are created and fewer calls to the dispatcher are necessary.*

Therefore, the lifted code can be optimized by making use of built-in function call mechanisms in LLVM-IR. A function call in machine code to a function call in the LLVM-IR and a return instruction is lifted as a function return. For correctness, after the lifted function call succeeded, the return address must be verified and only in case of mis-speculation, a dynamic dispatch is required. A comparison of the default approach and the optimized call–return is shown in Figure 3.3.

This optimization has two key advantages: first, a lifted function is more likely to cover the actual function in machine code when it contains calls to other functions. When lifting larger sections of code, this leads to a reduced number of lifted functions. And second, when the lifted code is compiled again to machine code, less dynamic dispatches are required and the return address prediction of the CPU is utilized, reducing the execution overhead and the number of dispatch operations. One disadvantage, however, is the possibly increased complexity of the lifted functions, which may have a negative performance impact of subsequent optimization or compilation.

3.3.7.2 Flag Computation at Function Boundaries

In particular on x86-64, where many arithmetic instructions also update the status flags register, the computation of the flag values can accumulate to a significant overhead. While many of these computations can be eliminated during optimizations, this is not possible at the boundaries of the lifted code. This is the case on indirect jumps as well as function calls and returns. However, especially for function calls and returns, almost all compilers and widely used calling conventions do not require flag values to be preserved across a function call or return instruction[5]. Consequently, unused flag computations immediately before a function call or return can be discarded. As this optimization potentially modifies the program semantics — in particular with hand-crafted assembly code — it can be disabled.

3.3.7.3 Registers for Thread-local Storage

Some architectures provide additional registers designed for use as base address for thread-local storage. For example, on x86-64 [Int20a], the segment registers `fs` and `gs` have a configurable base address and can be used in combination with all memory operands; and on AArch64 [Arm18], the user-accessible system register `tpidr_el0` is generally used for this purpose. These registers are also stored in the CPU state, but unlike other registers, they are not passed as SSA registers through the function, because they are used only rarely. When required, the register value is loaded from the CPU state in memory; optimization passes are expected to combine redundant load operations.

[5] I am not aware any general-purpose compiler which makes such assumptions and never encountered such cases with the benchmarks used in this thesis or other publications.

Nonetheless, on x86-64, also native segment registers can be used instead to avoid these additional memory accesses: the segments `gs` and `fs` are mapped to LLVM's address spaces 256 and 257, respectively, which are defined by LLVM to correspond to these segments. However, this optimization is expected to only have a low performance impact due to the rare use of segment registers.

3.4 Architecture Support

Rellume currently supports the architectures x86-64 [Int20a] and 64-bit RISC-V [RIS19] with common extensions (*rv64imafdc*), but is structured to generally support widespread general-purpose ISAs. The library is designed to be easily extensible towards new architectures, while at the same time being able to re-use most architecture-independent parts like handling of control flow or registers. This section outlines the steps required to add support for a new architecture to Rellume and afterwards describes handling of some specific features of ISAs, which need to be considered when extending the lifter.

3.4.1 Adding a New Architecture

Adding a support for a new architecture to RISC-V requires following these steps:

1. *Specify registers.* As first step, the structure of the CPU state has to be defined. This includes defining the number and sizes of supported registers and their facets.

2. *Supply decoder.* A necessary requirement for lifting machine code is to decode the stream of binary code into separated instructions. The decoded instruction can be stored in arbitrary format, but have to be annotated with information about the size of the instruction and possible control flow modifications. The latter consists a classification (e.g., conditional branch, function return, or unchanged) and, if available, the addresses of possible branch targets. This information is required for the architecture-independent control flow handling.

3. *Implement instruction lifting.* A mapping of instruction semantics to corresponding LLVM-IR code is required. The instruction lifter can rely on the common infrastructure for handling registers and memory accesses.

4. *Extend list of supported architectures.* Finally, the configuration API of the library has to be updated to indicate support for the newly added architecture

In most cases, following these steps suffices to extend the lifter. However, depending on the size and complexity of the instruction set, the third step might be considerable effort, in particular with large ISAs, like AArch64 [Arm18].

3.4.2 Handling Specific ISA Features

Widespread ISAs share several common concepts and approaches, while they also differ in some areas. A compiler IR, like the LLVM-IR, is usually an intersection of features from the supported architectures, making it easy to target them during code generation. This, however, implies that some concepts from ISAs cannot be easily represented in the IR, and some low-level concepts may not be representable at all. Other characteristics can be modeled appropriately, but care must be taken to correctly represent subtle differences of the same concept found in different ISAs. This section outlines some problematic ISA features and possible solutions for mapping these to LLVM-IR constructs.

Integer Division An integer division operation and the related integer remainder or modulo operation have two corner cases: first, the divisor might be zero, in which case the result is mathematically not defined; and second, for a signed division of the minimum integer value by -1, an integer overflow occurs as the binary representation of the quotient needs more bits than the dividend itself.

Different architectures pursue different strategies for handling these corner cases: for example, AArch64 [Arm18] and RISC-V [RIS19] return defined values when encountering such operations, whereas x86-64 [Int20a] raises an exception. To address the diversity of hardware implementations of these cases, programming languages like C [ISO17] and compilers like LLVM [LLV20b] define integer division with such values as undefined behavior, where it is unspecified whether the operation will succeed at all, and if it does, what the resulting value is.

This, of course, is not a viable option when lifting a well-defined operation from machine code to LLVM-IR and therefore lifting an instruction for integer division requires additional checks and handling for these corner cases.

Floating-point Semantics One non-trivial problem of properly representing floating-point semantics in LLVM-IR is the lack of support for different rounding modes; currently, only the default mode *round-to-nearest* is supported[6] [LLV20b]. However, as demanded by the C standard [ISO17], the rounding mode is dynamically configurable using a library function, and therefore, many ISAs implement a register to control the rounding mode [Int20a; Arm18; RIS19]. Additionally, some ISAs support an explicit specification of the rounding mode for conversion instructions (e.g., AArch64 [Arm18]) or for all floating-point instruction (e.g., RISC-V [RIS19]).

As many applications are not highly sensitive on exact floating-point arithmetic, ignoring the dynamically configured rounding mode is generally sufficient for most arithmetic operations. For conversions to integer values, however, the rounding mode

[6] There is ongoing work for proper support of other rounding modes, and parts of this support are available using *experimental* instrinsic functions.

has a highly visible effect, and consequently needs explicit handling. This can be realized by first rounding the floating-point value to the next integral number using LLVM intrinsics, like `llvm.round`, before doing the actual conversion. However, as not all targets have native support for these intrinsics, they may be lowered to library function and require a software implementation.

A bitwise accurate representation of floating-point operations with proper handling of special values like *SNaN*/*QNaN* with native LLVM operations for floating-point arithmetic is not possible. While LLVM and most ISAs closely follow the IEEE-754 standard [IEE08; LLV20b], in some cases, like the computation of the minimum of two numbers, there are minor deviations from the standard with regard to the handling of *NaN* values, and different ISAs specify different operations in such cases. Thus, bitwise accuracy ultimately needs a full software implementation of floating-point arithmetic, which may have a significant performance impact.

Atomic Memory Operations Multi-threaded programs require atomic memory operations for access to shared variables or locks. Consequently, the LLVM-IR provides instructions for atomic *compare-and-swap* and *read-modify-write* primitives combined with a proper lowering to equivalent sequences of machine instructions. On the other side, ISAs like RISC-V [RIS19] provide instructions for atomic *read-modify-write*, but no *compare-and-swap*. Instead, such operations shall be realized using a loop of *load-reserve/store-conditional* instructions, which only succeed, if the memory location at the specified address was not modified by another thread in between.

Such low-level loops cannot be represented in LLVM-IR and correctly translating such sequences to a different architecture requires either hardware support or extensive software handling for tracking memory accesses [Cot+17]. In the current implementation of the RISC-V lifter, such loops are, therefore, currently transformed as non-atomic memory operations. Support for correctly lifting *load-reserve/store-conditional* loops likely requires changes to the LLVM-IR.

Memory Ordering While many architectures, like ARM or RISC-V, implement *weak ordering*, which allows processors to reorder memory operations in a flexible way and therefore require memory barriers in multiprocessing environments [McK10], x86-64 implements *processor ordering* [Int20a], which basically only allows reordering memory load operations before memory write operations. LLVM, however, implements a memory ordering model similar to weak ordering [LLV20b], which allows more optimizations and simplifies targeting architectures with weak ordering.

Consequently, lifting x86-64 memory operations to LLVM-IR would technically require to add atomicity constraints to all load and store instructions. As this would cause the code generator to insert many memory barriers with associated performance costs, this is currently not implemented.

3.5 Library API

The library Rellume provides an API to allow users, for example, a dynamic binary rewriting system, to lift given machine code into corresponding LLVM-IR code.

The API of Rellume is designed to be simple and flexible with regard to support of other and future architectures. Generally, the API is structured as follows: first, the user has to create a configuration object, where they can specify the architecture as well as other options, for example, configuring specific optimizations or the insertion of markers to simplify relating the lifted code with the original code. With this configuration object, a new *lifted function* can be created. This function can be populated with instructions; based from specified starting addresses, all instructions reachable through branches or only individual instructions can be added. Once all requested instructions were added, the function can be finalized, which implies linking the control flow of the provided machine code instructions and returning the LLVM-IR function.

Listing 3.1 shows a usage example of the Rellume API. Initially, the user creates an LLVM-IR module, which will later be used to hold the lifted function. Afterwards, a Rellume configuration is created. At this point, the configuration can be modified; in this case, only the architecture of the machine code is specified as x86-64. After all required configuration options have been set, a new lifted function is created with the configuration and the module. Subsequently, all code reachable from the start address of the code is lifted into a new LLVM-IR function. This function is handed over to the user of Rellume after the lifting process is finalized.

```
1  static const unsigned char code[] = { /* ... */ };
2  int main(void) {
3      // Create LLVM module
4      LLVMModuleRef mod = LLVMModuleCreateWithName("lifter");
5
6      LLConfig* cfg = ll_config_new();           // Create config.
7      ll_config_set_architecture(cfg, "x86-64"); // Set architecture
8
9      LLFunc* fn = ll_func_new(mod, cfg);        // New function
10     // Lift code reachable by following all direct jumps. The last
11     // two parameters allow for an optional virtual address mapping
12     ll_func_decode_cfg(fn, (uintptr_t) code, NULL, NULL);
13     LLVMValueRef llvm_fn = ll_func_lift(fn);   // Finalize function
14
15     LLVMDumpValue(llvm_fn);
16     return 0;
17 }
```

Listing 3.1: *Usage example of the Rellume API. All instructions in* `code` *reachable via direct branches are lifted into the LLVM-IR function* `llvm_fn`*.*

To account for the fact that the machine code designated for lifting may reside at a different address during lifting than during its execution — for example, when a file is mapped to an arbitrary memory address — the functions to decode instructions into a lifted function allow for an optional address mapping: an function to access instruction bytes at a given address from the perspective of the lifted code can be supplied as parameter; if omitted, Rellume assumes that the lifted code resides in the same address space.

3.6 Discussion

A general quantitative evaluation of tools or libraries that lift machine code to LLVM-IR is problematic, as it strongly depends on the surrounding use case. Similarly, a comparison of lifted code fragments is likely misleading given the strongly diverging use cases and design goals of other approaches. This section will, therefore, pursue a discussion of the lifting approach implemented in Rellume. After describing the different operation constraints for static and dynamic lifters, the approach of Rellume will be compared with the actively maintained state-of-the-art lifters McSema [Tra21] and S2E [CKC11] (cf. Section 3.2); Figure 3.4 gives an overview on these different approaches. The performance aspect of the generated code by Rellume and the lifting process itself is evaluated in Section 4.4 for the context of DBT, in Section 5.4 for the context of DBI, and also in Section 6.8 for the context of dynamic binary optimization.

Operation Constraints Dynamic lifters have inherently different operation constraints compared to static lifters.

Static lifters must cover many possible execution cases. Identifying machine code and recovering control flow from a binary file is a hard problem. In particular, distinguishing machine code from constant data and identifying possible targets of indirect jumps are non-trivial tasks. Thus, some lifters like McSema [Tra21] or RevGen [Cyb18] rely on widely-used reverse engineering software for control flow recovery. A lifter operating in a dynamic context, in contrast, does not have to handle all possible cases at once and can defer handling of situations like the identification of dynamic jump targets until they actually occur.

Additionally, when a dynamic lifter is used for immediate execution of the lifted code, the lifter generally has to face tighter performance constraints. Furthermore, not only the time needed for lifting accounts for the overall execution time, but also the time needed for subsequent processing, for example optimization or code generation. Hence, a performance-oriented dynamic lifter has to optimize the performance of the lifting process *and also* the number and complexity of subsequently needed transformations. In the dynamic context, additional analyses for recovering specific

language constructs from machine code can only be used if they are expected to pay off by reducing compilation or execution times.

Another major difference between static and dynamic lifters is the input format: static lifters generally expect an executable file they can analyze completely, whereas a dynamic lifter may not have all information available. For example, when a binary is loaded into memory, typically not all sections are mapped; and dynamically generated code has no associated executable file at all.

In combination, these constraints imply that a system for lifting machine code to LLVM-IR in a dynamic context shall avoid unnecessary code transformations, costly analyses, and handling of unlikely corner cases, but instead has to properly support cases of incomplete information and only partially available machine code.

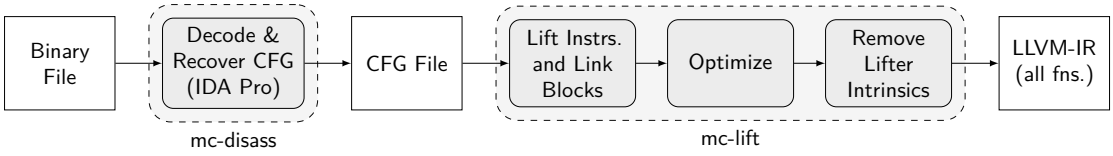
Comparison with McSema McSema [Tra21] is a tool to transform entire binary files in equivalent LLVM-IR code with the focus on binary analysis and reverse engineering, Figure 3.4a gives a rough overview over the lifting procedure. The tool relies on external programs to identify of functions and their separation basic blocks. Additionally, during the initial lifting step, several constructs from machine code, for example memory accesses, are initially mapped as calls to internal *intrinsic functions* of the lifter, which are only removed at a later stage after optimizations. Furthermore, registers are initially lifted as memory access to their CPU state, such accesses are only combined and eliminated at a later stage.

The additionally required transformation to remove lifter intrinsics and the transformation of memory-backed guest registers to SSA registers essentially rewrite most parts of the initially lifted code, with associated optimization costs. The dependency on external (proprietary) tools additionally increases the effort needed for an integration into dynamic systems.

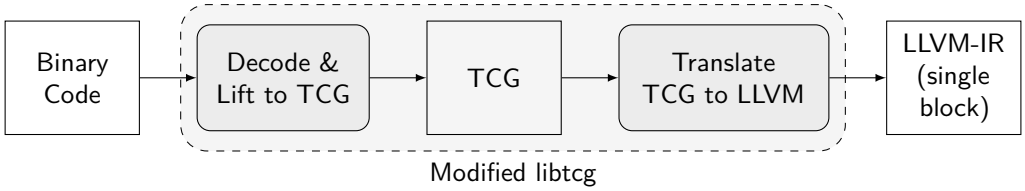
Rellume, in contrast, avoids these extra transformation steps by lifting to the desired idiomatic LLVM-IR code directly and is able to recover control flow dynamically. Rellume also does not attempt to identify global variables or calling conventions, as this is not required for the case of dynamic re-execution. However, this also implies that Rellume is currently (but not inherently) less suitable for static analysis due to the lack of such analyses.

Comparison with (systems based on) the S2E Lifter The lifting component of S2E [CKC11] is based on a modified version of TCG; a basic block of machine code is first transformed to TCG IR code and further transformed into an LLVM-IR function. Figure 3.4b gives an overview over the lifting procedure.

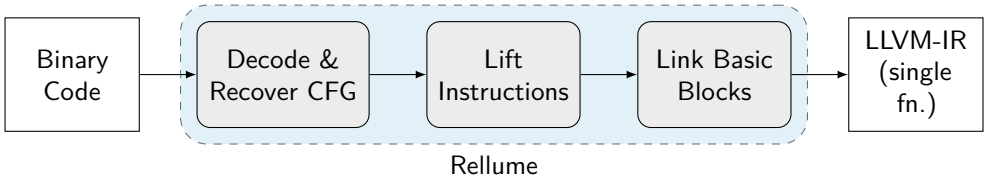
This already exposes a significant limitation of this lifting approach: only single basic blocks are lifted at a time, and lifting separate basic blocks results in separate LLVM-IR functions. Representing the control flow of machine code functions in LLVM-IR functions requires significant further transformations, as per-



(a) *Simplified lifting process of McSema [Tra21]. Functions, basic blocks, instructions, and the CFG are separated with a separate tool into an intermediate file format; the actual lifting process uses this information [Tra19a]. After the instruction semantics have been transformed to LLVM-IR, the code is optimized and only afterwards, so-called* intrinsics *of the lifter (e.g., for memory accesses) are removed.*



(b) *Lifting approach of S2E. LLVM code is derived from previously created TCG code. Only single basic blocks are lifted, for a representation of control flow, multiple blocks need to be lifted separately and subsequently combined.*



(c) *Lifting approach of Rellume. LLVM code is derived from previously created TCG code. Only single basic blocks are lifted, for a representation of control flow, multiple blocks need to be lifted separately and subsequently combined.*

Figure 3.4: *Slightly simplified overview on the lifting steps of McSema, S2E, and Rellume. All three approaches use different internal representations and also differ in their lifting granularity — McSema lifts entire binary files with (possibly) multiple functions, S2E only single basic blocks, and Rellume single functions consisting of (possibly) multiple basic blocks.*

formed by RevGen [Cyb18], which relies on McSema for function identification, or rev.ng [DFA18], which combines all lifted basic blocks into a single function. Thus, such TCG-based approaches avoid significant overhead during program execution at the cost of more expensive transformations.

Rellume avoids these transformations and can generally be used in similar contexts as the S2E lifter. However, Rellume currently supports fewer architectures than TCG and is also less tested.

3.7 Summary

This chapter described the library *Rellume*, which lifts machine code functions to equivalent LLVM-IR code. The library focuses on the performance of the lifted code and also reduced the complexity of required code transformations. During lifting, the control flow of the machine code is reconstructed before the instruction semantics are transformed into semantically equivalent LLVM-IR code. Machine code is lifted with near-function granularity, stopping only function calls and indirect jumps; as an optional optimization function calls are predicted to return ordinarily for even larger granularity. The lifting library currently supports x86-64 and RISC-V, but is designed to be easily extensible towards other widespread contemporary architectures. The next chapter details Instrew, a framework for LLVM-based Dynamic Binary Translation based on Rellume; and Section 6.7.3 describes an approach for dynamic binary optimization based on LLVM and Rellume.

4 Instrew: LLVM-based Dynamic Binary Translation

The previous chapter described a library for lifting machine code to LLVM-IR with focus on performance in a dynamic context. This chapter will build upon this library and introduce the framework *Instrew*, which performs user-level Dynamic Binary Translation and Dynamic Binary Instrumentation solely based on LLVM, allowing for increased optimization possibilities and higher performance. In the following, the Instrew architecture and the use-case of Dynamic Binary Translation without functional changes is described in detail; the instrumentation capabilities for modifying program behavior are covered in Chapter 5.

Publication Information

The approach described in this chapter was previously published in [ES20a] with further extensions and modifications described in [EOS21].

4.1 Motivation

Executing programs that are compiled for a different CPU is a frequent problem, for example with legacy or proprietary software where the source code is unavailable or porting would constitute too much effort. Such code may be compiled for an entirely different ISA than the host architecture, for example, executing x86-64 binaries in AArch64, but might also be compiled for a later version of the same ISA requiring ISA extensions that are unavailable on the target CPU. This point can become especially relevant for RISC-V [RIS19], where ISA extensions can be combined flexibly and vendor-specific extensions are actively encouraged.

A very similar situation occurs in computer architecture research when developing new ISAs or ISA extensions. An evaluation of newly added instruction is principally possible by using FPGAs — such simulators are very accurate, but also time-intensive during development and therefore unsuitable for rapid design iterations. Furthermore, for many parts of the design process, cycle-accurate performance results are not needed; for example when optimizing compiler support, statistics on executed instructions and their effectiveness can already suffice.

While plain interpretation of the program instructions is possible, interpretation of machine code adds massive overhead, very similar to interpretation of programs.

Thus, this overhead can be significantly reduced by translating code fragments from the guest to the host architecture using Dynamic Binary Translation (DBT). The remaining overhead depends on the quality of the translation and also the time taken for the translation itself.

Therefore, a DBT system faces the challenge to generate efficient machine code for the host architecture with a low translation and optimization overhead. Especially longer-running applications, however, benefit from more optimizations, outweighing the effort needed for the more costly translation process. This motivates to further emphasize the quality of the translated code using of a more advanced optimization and code generation procedure, similar to those found in standard compilers.

4.2 State of the Art

Binary translation can be performed dynamically during program execution by translating code when required or statically before program execution by translating the executable file as a whole. Dynamic approaches generally allow for complete program execution whereas static approaches are faced with the problem to discover all possibly executed code fragments before the program is executed — handling of dynamically generated or modified machine code is not possible with static translation alone. The advantage of static binary translation over dynamic translation, however, is the absence of translation overhead at the run-time of the translated program combined with the ability to perform further optimizations as all code is available and known during the translation process. Nevertheless, dynamic approaches are much more common, most likely due to their applicability on more applications without translation limitations and their ease of use.

4.2.1 Dynamic Binary Translation

User-level DBT is the most common approach for binary translation. Many of these approaches are based on QEMU [Bel05], an infrastructure for virtualization with support for many architectures. This is achieved by adding an abstraction layer between the guest and the host architecture. Other dynamic approaches exist as well, but are typically specialized on specific guest–host architecture combinations.

4.2.1.1 QEMU-based Approaches

QEMU [Bel05] is the de-facto standard for many kinds of virtualization and DBT-based ISA emulation. As such, it covers a wide range of commonly used architectures for guests and hosts. There are two emulation modes [QEM20a]: in system emulation, a full system with different privilege levels and a nested Memory Management Unit (MMU) is provided to emulate a full operating system. Contrary, in user-mode

emulation, user-space programs are emulated and system calls are passed through to the host operating system where possible. If the guest architecture is the same as the host architecture, QEMU can use hardware virtualization exposed through KVM, if available, otherwise QEMU falls back to regular DBT [Bel+17].

Internally, the binary translation component of QEMU is based on the Tiny Code Generator (TCG) [Bel+19; QEM20b; Bel+20] IR. TCG is a virtual ISA with a fixed set of registers and a strong type system. As TCG itself originates from a compiler back-end, it has the notation of functions, but within QEMU, only basic blocks are translated in one step. The virtual instruction set only supports integer operations; floating-point arithmetic and other complex guest instructions are only supported by including calls to helper functions [Bel+20]. During emulation, the front-end lifts basic blocks of guest code to TCG, where lightweight optimizations like the elimination of dead instructions are performed. Afterwards, the back-end compiles the code for the host architecture. For precise handling of CPU exceptions from the host, a mapping from the instructions of the translated code to the guest code and the associated virtual CPU state is maintained [QEM20b].

The binary translator, the original code, the translated code, and all memory allocated by the guest program reside in the same address space for simplicity. To prevent the guest program from accessing memory of the binary translator itself, all guest memory allocations and memory accesses are translated to add an additional constant offset, so that memory below this offset is unaddressable for the guest.

LLVM-based Optimizations As a consequence of the basic block granularity and the simple virtual instruction set, QEMU/TCG, while designed for portability and precise emulation, is not focused on efficient emulation. Consequently, several systems based upon QEMU for improved performance have been proposed, in particular by using the LLVM framework for optimizations.

Chipounov et al. [CC10] translate basic blocks from TCG further to LLVM-IR to make use of the LLVM optimizing code generator, but find that this simple approach incurs a high translation overhead from LLVM, which cannot be compensated. Jeffery [Jef09] experiments with a hybrid approach, where LLVM is used on a different core in parallel to the translation using TCG, replacing the TCG-generated code when the LLVM-generated code is ready. However, with this approach they find that the optimized code was only executed rarely.

LnQ [Hsu+11] were the first to achieve performance improvements by replacing TCG with LLVM inside the QEMU framework. Their translation approach is based on translating basic blocks by inlining prepared code fragments for the individual instructions. However, further details on performed transformations or the actual code fragments remain unclear. HQEMU[1] [Hon+12] achieved slightly higher performance improvements by keeping TCG as fast-path code generator and

[1] HQEMU appears to be a successor of LnQ.

running the LLVM translation only on hot traces in a separate thread. Sequences of TCG code blocks are lifted to LLVM-IR for optimized code generation of longer code sequences. A variant of HQEMU [Hsu+15] designed for use on less powerful hardware supports executing the LLVM-based optimization on a different system, however, this execution mode is no longer supported in newer versions.

Based on this prior work it can be observed, that an increased translation granularity is necessary for effective use of LLVM and its optimizations. This, however, is difficult to realize with QEMU and especially with TCG due to the limitation to basic block translation and a rather small virtual instruction set.

Multi-threaded Emulation With the increasing availability of multi-core processors, also support for multi-threaded programs inside DBT became a subject of research. Problems not only arise in the handling of the code cache, which can be either unified for all threads or separated for each thread, but also in properly mapping memory semantics of the guest architecture to the host architecture.

COREMU [Wan+11] is a port of QEMU to support multiple threads with a unified code cache, but with an incomplete implementation of the semantics for atomic memory operations. PQEMU [Din+11] experimented with both, shared and unified code caches, but made no advances for a correct implementation of memory consistency and atomic operations. Pico [Cot+17] implements different schemes for handling atomics based on a software emulation or hardware transactional memory; the software emulation was later merged into upstream QEMU. Qelt [CC19] further improved the code translation performance by partitioning the code cache for each thread and added support for using the host FPU for floating-point emulation. Furthermore, they added an instrumentation API to TCG.

4.2.1.2 Other Dynamic Binary Translators

Other approaches besides using QEMU exist, but are usually focused on selected guest–host pairs. One of the first DBT systems was Mimic [May87], which dynamically translates from System/370 to RT PC by translating blocks of code at once. Walkabout [CLU02] is a binary translator that focuses on decoupling source and target architectures for eased retargetability, but has a high performance overhead.

Since the publication of QEMU, however, fewer new DBT systems were proposed. One of them is HERMES [Zha+15], which does not perform optimizations on the IR used during the translation, but applies post-optimizations on the translated machine code.

Rv8 [CH17], RISCV-DBT [Guo18], and RIA-JIT [Dor+20] are recent approaches targeting optimized translation of programs compiled for the emerging RISC-V architecture to x86-64, experimenting with different optimization strategies like instruction fusion and load–store elimination. All of these systems use the same address space for the guest program and the binary translator itself, like QEMU.

Windows supports executing 32-bit x86 binaries on AArch64 using a transparent dynamic binary translation layer, which caches translated code fragments across execution of the program, further details on the translation mechanism are unknown [Mic21]. Apple includes a binary translator for x86-64 binaries to run on AArch64 processors in recent released of macOS, making use of hardware support for the x86-64 memory model and caching of translated code fragments [App21; Søg20]; further details on the actual translation process are unknown.

Transkernel [Guo+19] is a kernel-level approach targeting heterogeneous systems and uses QEMU to transparently migrate to and execute processes on processor cores with a different architecture.

4.2.2 Static Binary Translation

Binary translation can also be performed offline, where an executable file compiled for a source architecture is rewritten to be natively executable on a probably different target architecture. The general procedure begins with an initial decoding step, where machine code is lifted to some architecture-independent IR. From this IR, a new executable file for the target architecture is generated. During the lifting step, further analyses may take place to recover more information about functions and program semantics; and during the lowering step, target-specific optimizations are performed. [Ang06; CV00] In this section, only more general static binary translation systems without a fixed source–target architecture combination will be covered.

The critical point of static binary translation is to discover as much possibly executed code as possible in the executable file. As code and data may be interleaved and cannot be separated clearly, the translated binary file includes a copy of the original file [CV00; Cyb18]. As it is impossible to discover and therefore translate all possibly executed code statically — for example, dynamically computed jump targets or JIT-compiled machine code — for full program correctness, the binary translator additionally need to add an emulator for machine code of the source architecture [CV00].

The University of Queensland Binary Translator (UQBT) [CV00; Cif+01] is a static binary translator framework with support for multiple code generation targets. Initially, the source binary code is decoded and lifted to an architecture-independent code representation and the control flow graph is recovered. For the compilation to different architectures, multiple approaches are implemented, including a back-end based on a standard C compiler. For handling of untranslated code, different possibilities are outlined, including an interpretation of a lower-level and more fine-grained code representation obtained during lifting as well as a brute-force approach to cover all possible code bytes during the translation process.

LLVM-based Approaches Many newer, more general approaches are based on the LLVM framework [LA04] by lifting the source binary code to LLVM-IR and

then using the LLVM optimizer and compiler to produce a new executable binary.

LLBT [She+12; SHY14] is a tool which lifts ARMv6 binaries to LLVM-IR, lifting all translated instructions into a single function. Indirect jumps are handled by looking for jump tables and potential targets of function returns combined with a dispatcher to dynamically jump to translated code fragments. The data sections of the source binary are kept without modifications; accesses to literal pools, which are commonly stored at the end of each function in the code segment, are handled specially during lifting.

Rev.ng [DFA18; Gus+19] originally was a tool for LLVM-based binary analysis, but has been extended to support static binary translation as well. The binary code is initially lifted to TCG, the IR also used by QEMU, from where it is lifted further to LLVM-IR. As done in LLBT, all code is lifted to a single LLVM-IR function combined with a dispatcher for handling indirect jumps. To allow for more optimizations, a further extension [Gus+19] identifies functions during lifting and moves them to separate LLVM-IR functions, while keeping the main dispatcher intact.

RevGen [Cyb18] also uses TCG to lift machine code semantics to LLVM-IR, but creates a LLVM-IR function for each basic block, which are subsequently combined using control flow graph information obtained through analysis using McSema [Tra21].

4.3 Instrew Architecture

The LLVM compiler infrastructure provides a flexible framework for high-quality code optimization and compilation. Thus, LLVM has already been used for binary translation [Hsu+11; Hon+12; She+12; DFA18], exploiting the architecture-independent LLVM-IR that provides sufficient abstraction from the original code and enables an effective usage of features provided by the host CPU. However, existing dynamic approaches were limited by small translation granularities of basic blocks or superblocks, which result from the integration with existing infrastructures; whereas static approaches were limited by the need to cover as many cases as possible during their translation step.

These problems can be solved by combining a dynamic translation mechanism, for higher flexibility, with a larger translation granularity, for enabling further and more complex optimizations and thereby using LLVM's optimization infrastructure to its full extent. This approach is implemented in the *Instrew*[2] framework. In Instrew, entire functions are lifted to LLVM-IR using Rellume (see Chapter 3) without an additional code representation layer in between. The resulting increased translation time generally pays off over longer runs.

[2] `https://github.com/aengelke/instrew`, open-source and licensed under LGPLv2.1+.

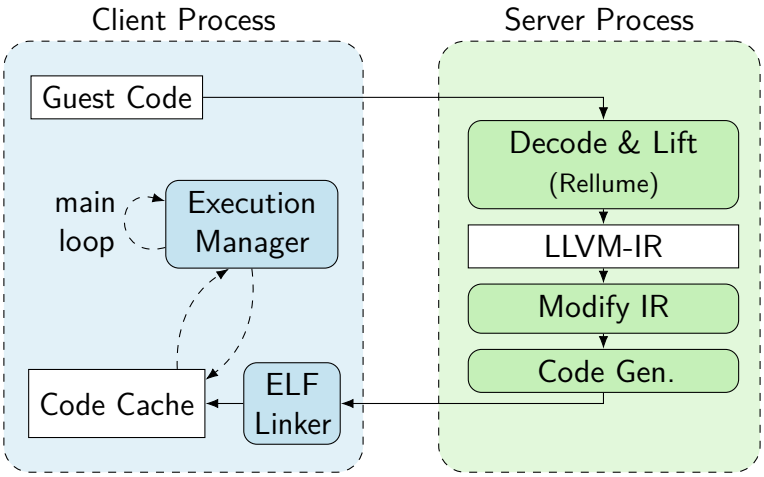


Figure 4.1: *Overview of the Instrew client–server architecture. The client process is responsible for controlling the program execution and executing the rewritten code fragments, while the server process performs the actual rewriting and optional instrumentation process. The communication between these processes is realized using an IPC protocol.* Figure slightly modified from [ES20a].

Client–Server Architecture Contrary to many existing dynamic binary rewriters, Instrew performs the code rewriting process in a so-called *server process* which is separated from the process that executed the rewritten code, which is referred to as the *client process*. The two processes communicate via an Inter-process Communication (IPC) mechanism. The overall architecture is depicted in Figure 4.1.

There are several benefits of this separation: first, it enables further optimization possibilities: for example, a permanently running rewriting server may cache rewritten code for use with multiple executions, reducing the translation overhead. Second, the rewriting process can be performed on a machine different to the machine used for executing the program. This is especially relevant for parallel applications running multiple threads or even on multiple compute nodes and avoids redundant rewriting of code. And third, this architecture provides a flexible foundation for developing different dynamic binary rewriting approaches. As the client is not aware of the rewriting process itself and merely processes rewritten code fragments, a different rewriting server that adheres to the communication protocol can be used instead or in addition to the LLVM-based rewriter.

4.3.1 Server Process

The Instrew server process is initially spawned by the client process and is generally passive. After the initial configuration handshake, the server waits for code rewriting requests from the client. Upon receiving such a request, the server responds to the client with requests to send the necessary instruction bytes. To avoid frequent or

redundant instruction requests, code bytes are requested with page-size granularity and cached in the server process. These instructions are then decoded and lifted to LLVM-IR using Rellume (see Chapter 3) with function granularity, optionally also using the call–return optimization described earlier in Section 3.3.7.1[3].

After lifting the function to target-independent LLVM-IR code, selected optimization passes provided by LLVM are applied to eliminate any overheads added during the lifting stage, such as unused status flag computations. Depending on the target and the configuration, the interface of the lifted function is adjusted to match the expectation of the client. Finally, the code is compiled into an ELF object file and sent back to the client.

4.3.2 Client Process

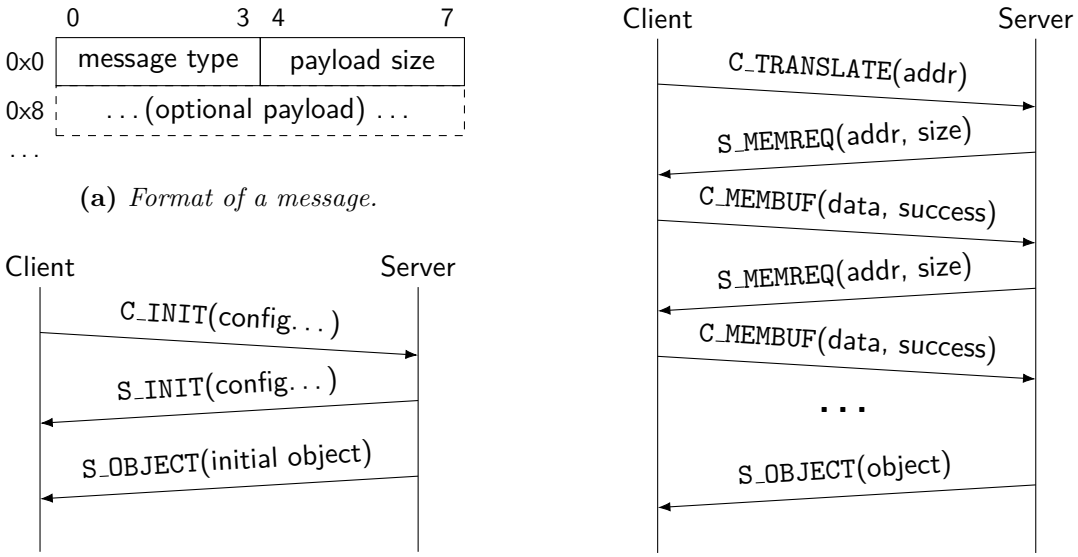
The Instrew client process manages the program execution flow and maintains the code cache of already rewritten program parts. As such, it initially establishes the connection to the server process, maps the emulated program into memory, and sets up the register state of the emulated CPU. Afterwards, it enters the main dispatch loop, which looks up the corresponding functions in the code cache and executes them. Whenever the dispatcher encounters a guest code address that was not translated before, a translation request is sent to the server. Once the translation process finishes, the returned ELF object is processed by storing the contained code in the code cache, applying relocations, and resolving symbols — this essentially turns the client into a simple run-time linker.

Besides the dispatching and linking functionality, the client also implements a system call abstraction to abstract Application Binary Interface (ABI) differences between the guest and the host architecture. System calls for spawning other threads are, however, currently not supported, this is left for future work. The function for handling system calls is accessible to rewritten code under a special function name. Additionally, also implementations of important standard library functions are provided, to which calls may be generated automatically by LLVM during compilation. These include `memcpy` and `memset`, but also 128-bit integer arithmetic used by x86-64 division instructions and specific floating-point operations like `floor` or `round`, which may have no corresponding hardware instruction on the target architecture.

4.3.3 Communication Protocol

The communication protocol between the client and server is a binary protocol with a primary focus on simplicity and efficiency. Currently, the communication

[3]There is an option to disable these optimization, as the increased granularity may have a negative overall performance impact due to increased code generation times



(a) *Format of a message.*

(b) *Initialization Protocol*

(c) *Rewriting Protocol*

Figure 4.2: *Overview on the communication protocol between client and server process.*

is performed using UNIX pipes, which also allow the server to run on a different machines using SSH, but using sockets or shared memory regions is possible as well and can be implemented easily. The communication protocol relies on messages, which have an 8-byte header containing the type as well as the size of the payload and the payload itself (see Figure 4.2a).

Initialization Protocol For the initial handshake between client and server (see also Figure 4.2b), the client sends a configuration object (`C_INIT`) for the rewriting process, which encodes command-line options like optimization or debugging parameters. The server then responds with a message (`S_INIT`) containing configuration options for the client, which currently includes information about the function interface of translated code. Additionally, the server sends a message (`S_OBJECT`) with a (possibly empty) initial object for linking to the client, which can contain additional helper functions that should be available to further objects.

Rewriting Protocol The protocol for rewriting code fragments works as follows (see also Figure 4.2c). The client sends a request (`C_TRANSLATE`) to the server, containing the guest address for translation as payload. If the server needs more instruction bytes from the client, the sends a memory request (`S_MEMREQ`), which contains the address and the size of the requested memory region. The client answers with a message (`C_MEMBUF`) containing the corresponding memory region and a final

byte indicating whether all bytes could be successfully read[4]. This step repeats until the server has collected all required instruction bytes for the rewriting process. Eventually, the server sends a message (`S_OBJECT`) with the object file containing the rewritten code. The client then processes this object file and executed the enclosed code.

4.3.4 Structure of Code Fragments

All code fragments received by the client from the server are object files in the Executable and Linkable Format (ELF) format. However, for simplicity of the client, only a subset of possibilities given by the format is supported.

Usage of ELF Symbols The most important restriction are the names of public functions. To store all functions in a hash table with a numeric key for simple lookup, each function name has to adhere a specific format: the first letter is "`Z`", followed by the number for the hash table index in octal representation. A more meaningful function name can follow afterwards, but is ignored for symbol resolution and only serves for debugging purposes. For functions that correspond to guest code, the respective guest address is used as numeric key and so that the function can be called by the dispatcher; for other functions, numbers which correspond to invalid guest addresses are used.

The ELF object may also reference functions from previous objects and also functions exposed by the client, for example for handling system calls. With this method, also the dispatcher itself can be called. This is used when the call–return optimization is active: the dispatcher can be called like an ordinary function to implement nested calls, which, as a consequence, simplifies the code generation process.

Function Interface Generally, functions that correspond to guest code and are called by the dispatcher have to follow a specific interface: these functions take a single parameter, namely the pointer to the CPU state, which is modified throughout the execution. However, this requires that all CPU state must be written back to memory at the boundaries of code fragments, inducing additional overhead. This can be avoided by using more a specialized calling convention: for x86-64 targets, LLVM supports the HHVM calling convention, where 12 guest registers can be kept in host registers. To this end, the server can refactor functions to use this special calling convention and eliminate unnecessary accesses to the CPU state in memory;

[4]Due to the direct use of the `write` system call for reading guest memory, an access to an invalid address do not cause a signal (`SIGSEGV`), but only causes the system call to fail with `EFAULT`. This case can be detected easily and the information about the error can be sent to the server with the final *success* field.

the client is informed about the selected register convention during the initialization handshake.

This transformation works for all supported guest architectures. Unfortunately, this approach only works on x86-64 hosts due to the absence of similar calling conventions on other targets. A more general calling convention that allows to explicitly specify register constraints would be required to generalize this approach to other targets; this is left for future work.

4.4 Evaluation

The overall performance of Instrew, but also the quality of the translated code fragments and the efficiency of the rewriting process, are evaluated using the SPEC CPU 2017 [Sta21] benchmark suite, which covers a wide range of workloads derived from real applications. As guest architectures, x86-64 and RISC-V64 are used, and the code is then dynamically translated to the host architecture, where x86-64 and AArch64 are used. The performance of Instrew is compared against a natively compiled baseline using the same source code for each host architecture, QEMU [Bel05] for all four guest–host combinations, and HQEMU [Hon+12] in its parallel hybrid mode for the combination of x86-64–AArch64. Rv8 [CH17] turned out to be entirely de-functional due to several implementation issues, including incorrect loading of ELF files as well as emulation issues leading to misbehaving applications and segmentation violations, presumably caused by bugs in the code translation process.

4.4.1 Setup

The x86-64 host platform is based on Intel Xeon CPUs (E5-2697 v3, Haswell), 17 MiB L3 cache and 64 GiB main memory, running SUSE Linux 15 SP1 with Linux kernel 4.12.14-197.40 in 64-bit mode. The AArch64 host platform is based on Cavium ThunderX2 99xx CPUs, 32 MiB L3 cache and 512 GiB main memory, running CentOS Linux 8 with Linux kernel 4.18.0-193.6.3.

All benchmarks are run with the reference input workload size in single-threaded mode; benchmark 600.perlbench was excluded due to use of x87 FPU instructions and unsupported system calls. Native code on the x86-64 platform is compiled with GCC 9.2.0; native code on the AArch64 platform and all cross-compiled code is compiled with GCC 10.2.0. For all compilations, optimization level `-O3` is used during compilation and benchmarks are statically linked against glibc 2.32.

For Instrew, commit `1cdb1bc`[5] is used together with LLVM 11. In addition to the *base* Instrew, the variations using the *HHVM* calling convention (for x86-64 hosts only) and the *call–return* optimization are evaluated. For QEMU, version

[5] `https://github.com/aengelke/instrew/tree/1cdb1bc`

5.2.0 is used, and for HQEMU, version 2.5.2 is used in combination with LLVM 6.0, which is the latest supported version of HQEMU and was patched according to the documentation.

4.4.2 Results

This section will describe the results of the translation grouped by their target architecture[6].

4.4.2.1 Translations to x86-64

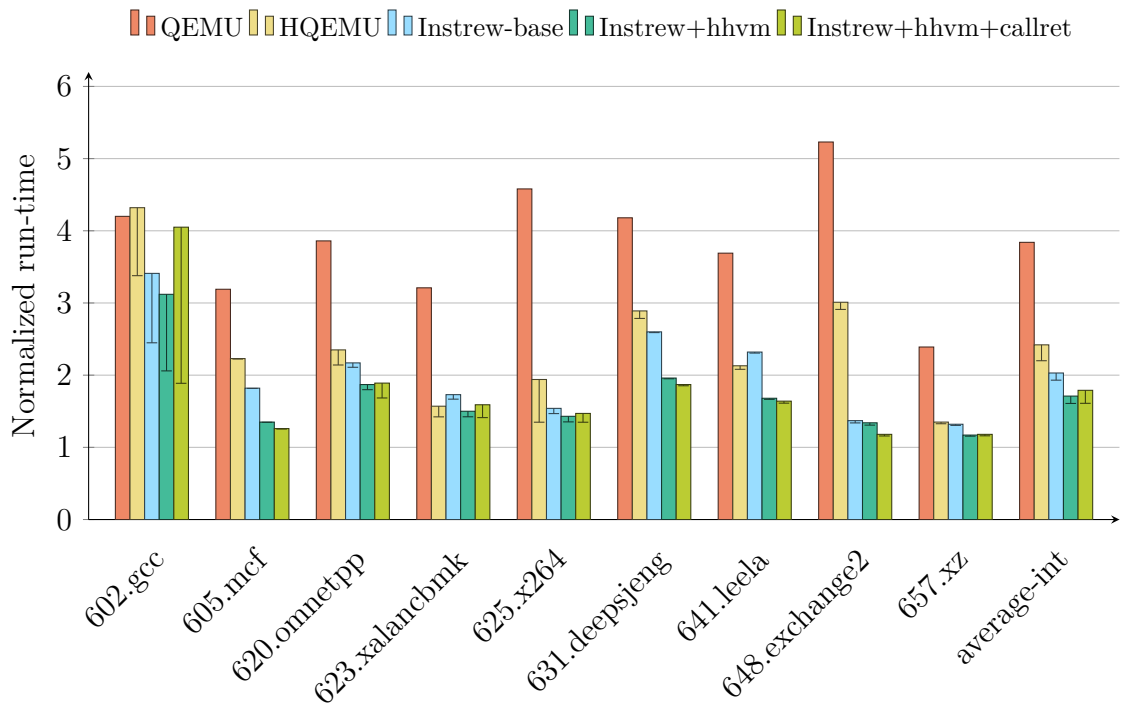
The overhead of the identity translation from x86-64 guest programs to x86-64 hosts is shown in Figure 4.3. For Instrew, the most performant configuration (*Instrew+hhvm*) has an overhead of 59% (int: 71%, fp: 48%) over the natively executed code, which is less then half the overhead of HQEMU with 135% (int: 142%, fp: 128%) and even more significantly lower than the overhead of QEMU with 1044% (int: 284%, fp: 1728%).

In comparison to the base Instrew, the optimization to use the HHVM calling convention reduces the average overhead from 78% to 59%, as 12 guest registers are always kept in host registers, significantly reducing the overall memory accesses at the boundaries of translated code fragments. However, this improvement is smaller for the floating-point benchmarks (55% → 48%). This has two main reasons: first, the HHVM calling convention does not include floating-point registers, so that all Streaming SIMD Extensions (SSE) registers still must be stored in memory. And second, the floating-point benchmarks generally have a longer execution time with less switches between code fragments, reducing the relative performance gain.

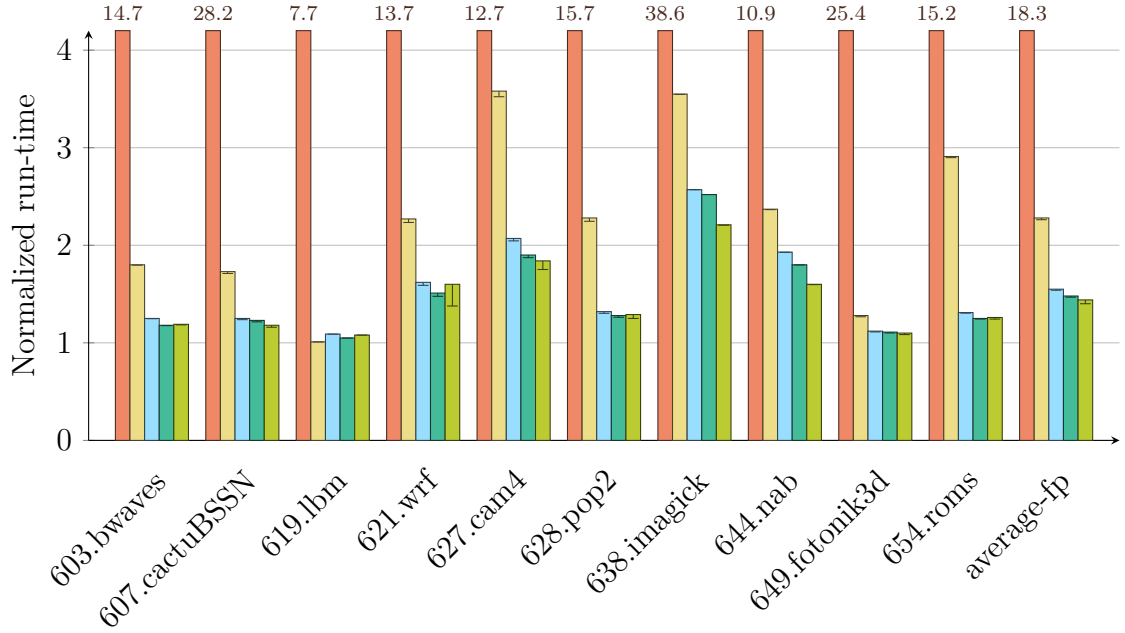
The call–return optimization leads to performance improvements on benchmarks that execute many function calls, for example on 605.mcf or 638.imagick. On other benchmarks, the performance of the translated code is improved, but negated by the increased translation time, which is driven by the increased size of code fragments. The most notable example for this behavior is the benchmark 602.gcc, where 53% of the time are used for translation (without call–return opt.: 34%). Overall, the call–return optimization in combination with LLVM 11 leads to a slight benefit of run-time performance but is overshadowed by the overhead of code generation.

When translating RISC-V to x86-64, the performance results are generally similar, as shown in Figure 4.4: the average overhead of *Instrew+hhvm* over the natively compiled code is 58% (int: 80%, fp: 37%); this is much lower than the overhead of QEMU with 827% (int: 210%, fp: 1383%). However, the results differ from x86-64

[6]Note: in literature, the mean is usually computed using the *geometric mean*, however, this thesis uses the *arithmetic mean* (average) to account for the fact, that there is no multiplicative relation between different benchmarks. Further differences from numbers previously published in [EOS21] are mainly caused by using a newer version of LLVM.



(a) *Results for SPEC CPU 2017 Integer benchmarks.*



(b) *Results for SPEC CPU 2017 Floating-Point benchmarks.*

Figure 4.3: *Performance results when translating x86-64 to x86-64, normalized to the execution of the natively compiled code. The relative translation time for Instrew and HQEMU is shown using error bars.*

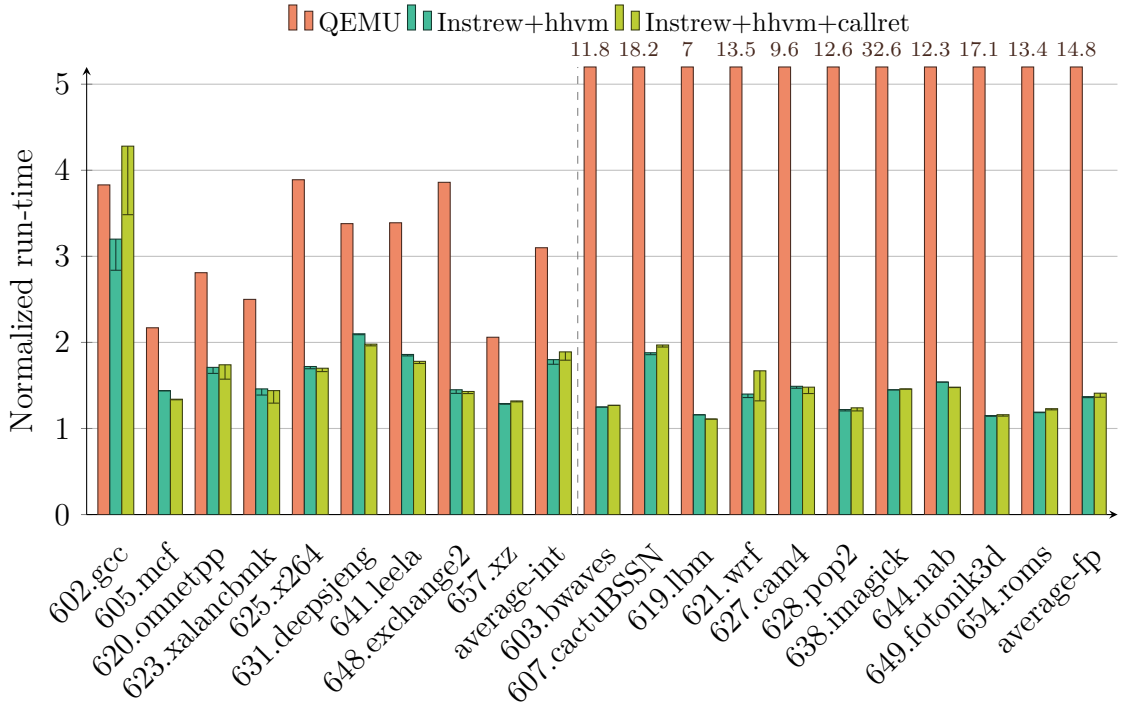


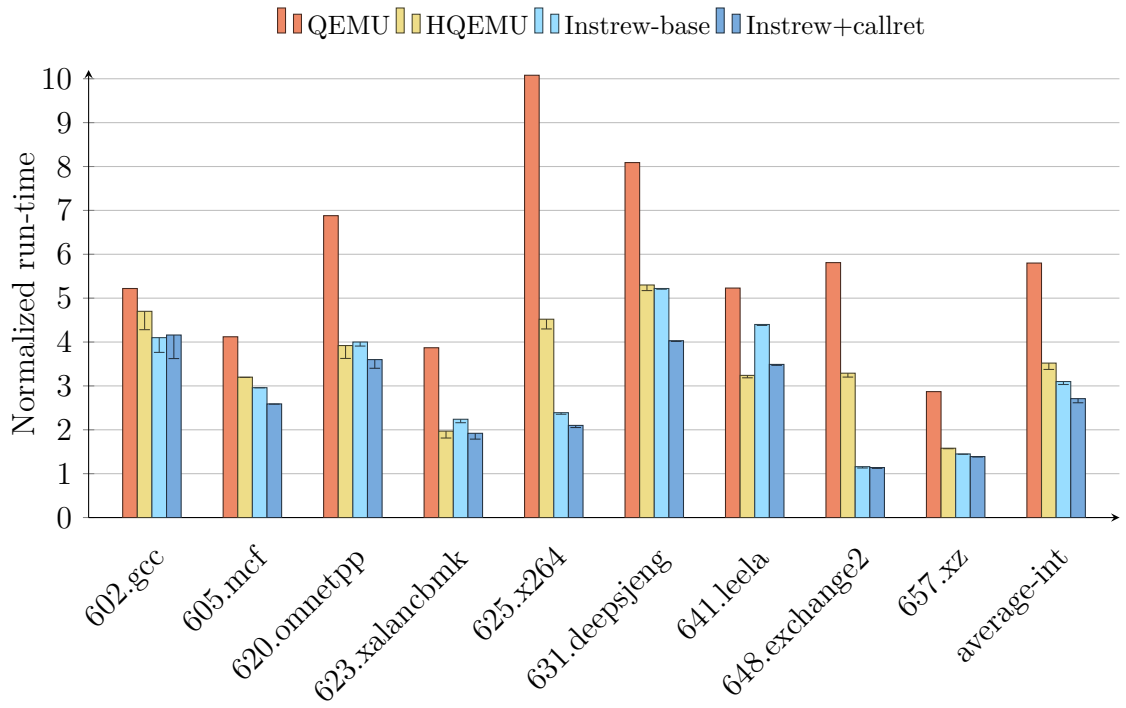
Figure 4.4: *Performance results of the SPEC CPU 2017 benchmarks when translating RISC-V64 to x86-64, normalized to the execution of the natively compiled code. The relative translation time for Instrew is shown using error bars.*

as guest architecture in an unexpected aspect: the floating-point benchmarks have a lower overhead when translating from RISC-V, despite the fact that RISC-V does not yet have standardized vector instructions. This has two reasons: first, some of the benchmarks do not benefit strongly from vector instructions; and second, the combination of Rellume and LLVM generates further optimized x86-64 machine code even if the input machine code was not strongly optimized.

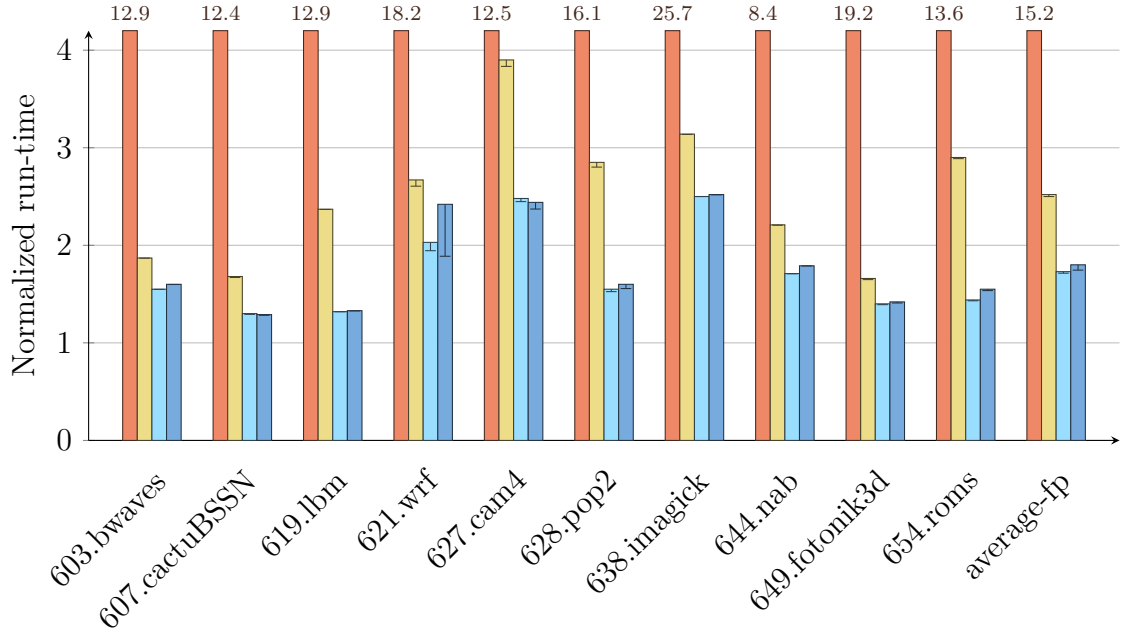
4.4.2.2 Translations to AArch64

The overhead over a natively executed program when translating x86-64 guest programs to AArch64 hosts is shown in Figure 4.5. For Instrew, the most performant configuration (*Instrew+callret*) has an overhead of 123% (int: 171%, fp: 80%) over the natively executed code, which is significantly less then half the overhead of HQEMU with 200% (int: 252%, fp: 152%) and also significantly lower than the overhead of QEMU with 973% (int: 480%, fp: 1417%).

A major source of overhead are frequent memory accesses for loading and storing the values of guest registers. Due to the lack of a calling convention that allows a flexible use of host registers, all guest registers have to be stored to memory whenever



(a) *Results for SPEC CPU 2017 Integer benchmarks.*



(b) *Results for SPEC CPU 2017 Floating-Point benchmarks.*

Figure 4.5: *Performance results when translating x86-64 to AArch64, normalized to the execution of the natively compiled code. The relative translation time for Instrew and HQEMU is shown using error bars.*

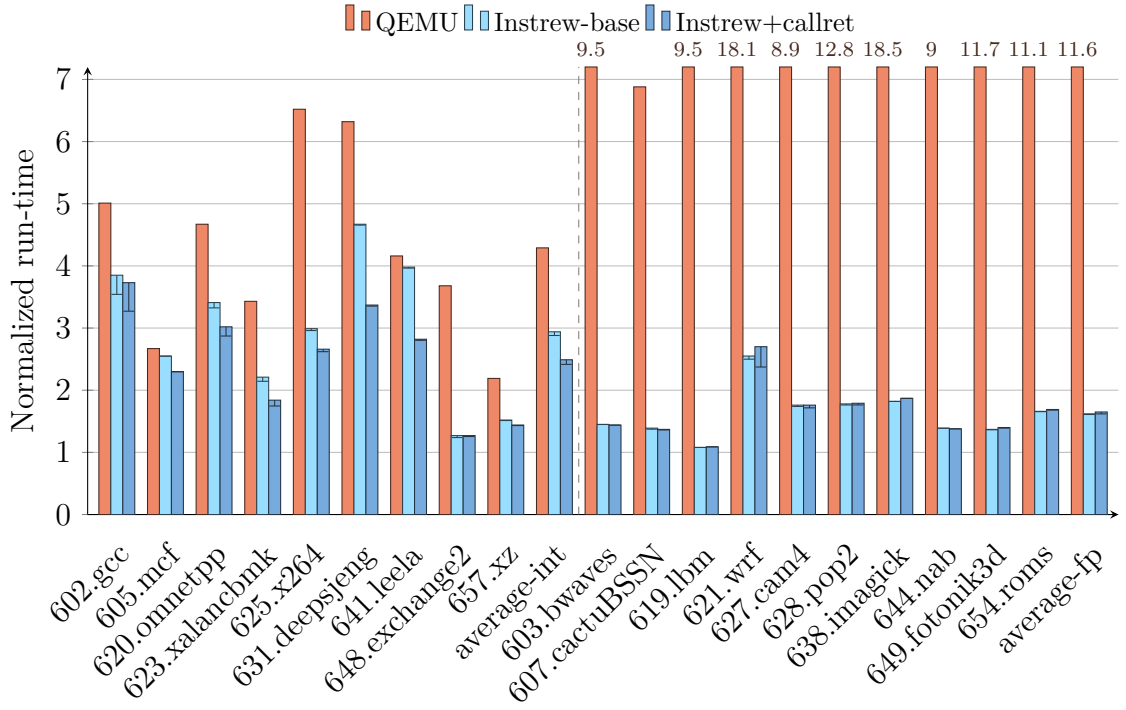


Figure 4.6: *Performance results of the SPEC CPU 2017 benchmarks when translating RISC-V64 to AArch64, normalized to the execution of the natively compiled code. The relative translation time for Instrew is shown using error bars.*

the program execution continues at a different code fragment. The increased amount of memory operations subsequently also affects the dispatcher, which needs to fetch the address of the next code fragment from memory. This causes pipeline stalls of several cycles on each dispatch, contributing to the overall overhead.

The call–return optimization, however, is much more effective on the AArch64 host machine than on x86-64 hosts. As the target of function returns are predicted using CPU facilities, generally less dispatch operations are required and therefore, in addition to possibly better use of code optimizations, also reduce the overhead incurred by the dispatcher. In combination with the lower code generation times for AArch64 targets, the call–return optimization reduced the overall overhead from 210% to 171% on the integer benchmarks.

When translating RISC-V to x86-64, the performance results are generally similar, as shown in Figure 4.6: the average overhead of *Instrew+callret* over the natively compiled code is 105% (int: 149%, fp: 65%); this is again much lower than the overhead of QEMU with 714% (int: 329%, fp: 1060%). The overhead is slightly lower than with x86-64 guests, presumably due to the larger register count of RISC-V, causing less accesses to stack memory in the guest code.

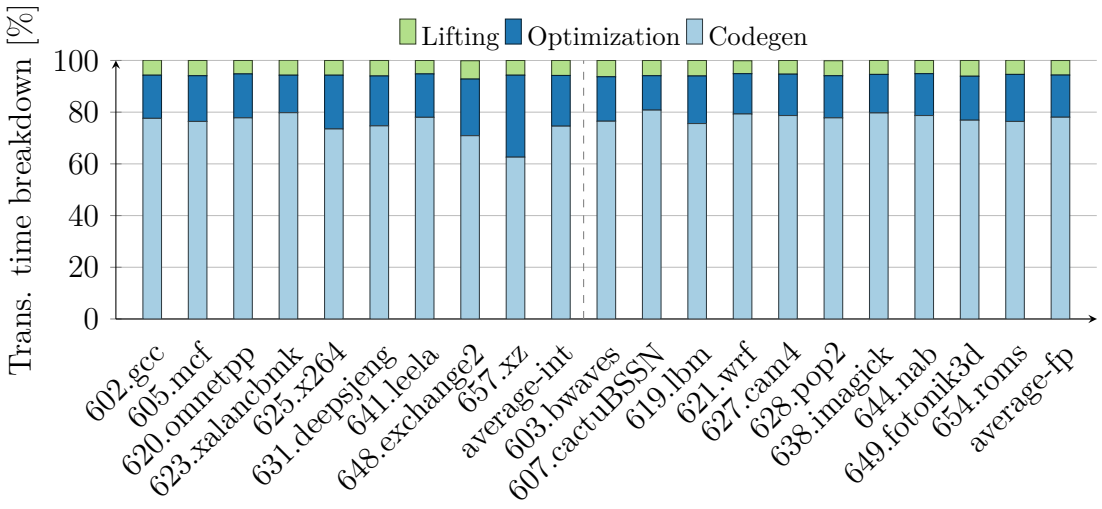


Figure 4.7: *Translation time breakdown for Instrew+hhvm when translating x86-64→x86-64. The linking time is not shown here, as it is always below 1%.*

4.4.2.3 Translation Times

To gain further insights into the performance of the rewriting process, the server process is modified to measure the time spent for lifting, optimization and code generation. Figure 4.7 breaks down the translation times for x86-64→x86-64 translations. For all benchmarks, the main part of the translation time is the code generation back-end of LLVM with an average of 76%. The fraction of time spent in Rellume for lifting the machine code is in the range of 5–7%. The time spent for communication with the client and the linking step on the client side is below 1% in all cases. The call–return mode increases the code generation time, leading to an average distribution of 4/13/83% for lifting/optimization/code generation.

One exception is the 602.gcc benchmark with an extraordinarily high rewriting time. This is rooted in the number of indirect jumps contained in the code, which is caused by heavy use of `switch` statements and indirect function calls on the original code. This not only leads to a high number of code fragments, but also to redundant code translation, as for each executed jump target, the remainder of the function is translated anew.

Another important factor for the translation time is the host architecture. Code generation on the AArch64 host is significantly faster than on the x86-64 host, both in absolute terms and in relation to the benchmark execution time; despite the fact that all benchmarks are generally slower on the AArch64 machine. For x86-64→AArch64 translations, this leads to an average time distribution of 5/21/74%.

The guest architecture, on the other side, has only a rather small impact on the translation time. Due to complexity of the operations from several instructions, using

x86-64 as guest architecture leads to higher translation times on some benchmarks (e.g., 602.gcc), but nearly unchanged translation times on others (e.g., 620.omnetpp).

4.4.3 Discussion

The results clearly show that Instrew achieves significant performance improvements over state-of-the-art systems for DBT. Especially in comparison to HQEMU, which also relies LLVM for machine code optimization, the performance overhead over the natively optimized code is more than halved.

This implies that the approach of directly lifting from machine code to LLVM-IR without an extra intermediate step, like TCG, is beneficial. Especially the larger translation granularity, which exposes common control flow constructs like loops in the LLVM-IR, allows for better usage of the LLVM optimization and code generation infrastructure.

Further performance improvements can be achieved when using a calling convention which does not require frequent write-back of the entire guest state to memory. This, however, is currently only supported on x86-64 hosts with the HHVM calling convention. Due to a missing generalized *all-registers* calling convention in LLVM, this concept cannot be easily ported to other target architectures and for implementing this concept in a target-independent manner, changes to the LLVM-IR are likely necessary.

A significant issue, especially for applications with a short execution time, is the translation time. In particular, the SelectionDAG instruction selector is known to have performance issues and there is an ongoing effort in the LLVM community to add a new back-end, GlobalISel, which addresses this problem [LLV21]. However, as Instrew's client–server also allows for a permanently running server or an extra caching layer between the client and the server, there are also other possibilities to reduce the impact of translation costs.

Finally, Instrew works with recent versions of LLVM without modifications — in contrast to HQEMU, which requires a patched version of an older LLVM version, decreasing maintainability and decouple it from recent improvements in the LLVM framework.

4.5 Summary

In this chapter, the Instrew framework for LLVM-based Dynamic Binary Translation was described. Basing on Rellume, machine code is lifted directly to LLVM-IR without an additional layer of code representation in between. Instrew implements a novel client–server approach for translation, where the entire process of lifting and translation is encapsulated in a separate server process. The client is responsible for managing program execution and sends translation requests to the server when

required. The communication protocol is designed for simplicity and conceptually also allows to use the same client with a different rewriting approach. Performance results on the SPEC CPU2017 benchmark suite on x86-64 show that Instrew has an average overhead of just 59% over the natively executed code, which is less than half of the overhead of HQEMU with 135% and even more significantly lower than plain QEMU with 1044%.

5 Instrew for Dynamic Binary Instrumentation

The previous chapter described the Instrew framework for performant DBT/DBI based on LLVM, with a particular focus on the overall translation process and the use-case of Dynamic Binary Translation. This chapter extends the Instrew framework with an API to additionally enable Dynamic Binary Instrumentation on LLVM-IR level, allowing to dynamically modify program behavior.

Publication Information

The general approach described in this chapter was previously published in [ES20a].

5.1 Motivation

Dynamic behavioral modification of compiled binaries is useful for for a wide range of use cases. A common example are tools which add additional integrity checks, for example by tracking the bounds of memory allocations [NS07a; BZ11; Ser+12], detecting use of uninitialized memory [NS07a; SS15], or revealing possible race conditions [JT08; SI09]. An entirely different application of program instrumentation is performance analysis and collection of performance metrics, often allowing deeper insights into performance characteristics of a program: for example, Callgrind [WKT04] dynamically inserts tracing code to gather accurate data on function timings and also memory accesses, enabling a simulation of cache behavior. ZeroSpy [You+20] puts extra checks into the program to dynamically detect redundant zeros, guiding optimizations to avoid inefficient redundant computations. Other applications include security enhancements and hardening [Haw+17; Zha+14] as well as assuring secure execution of programs [War+12].

The general technique to achieve such behavioral modifications is *program instrumentation*: additional code is inserted into the code before its execution. Such instrumentation can occur at different stages: it can occur during compilation, where a pass in the compiler instruments the program with the requested additional behavior. This, however, may be a significant effort as all parts of the program and libraries have to be recompiled for full coverage; and may be impossible if the source code is not entirely available.

Thus, program instrumentation is frequently performed at binary level. Like binary translation, this can either happen statically in a separate step before the program execution, or dynamically while the program itself is running. While static binary instrumentation may be beneficial for applications where the instrumented code is executed more often, only dynamic approaches can guarantee covering all possible situations as discussed previously in Section 4.2.

Consequently, several frameworks for Dynamic Binary Instrumentation (DBI) exist [NS07b; Luk+05; BGA03]. However, existing tools generally aim for low overhead of the instrumentation process itself and therefore either do not abstract from the original machine code instructions and therefore make code optimization significantly more difficult, or use a reduced architecture-independent IR to simplify instrumentation, but only perform few optimizations during subsequent code generation. In both cases, meaningful instrumentation results in a significant slowdown due to a lack of code optimization and optimized machine code generation.

A solution to this problem is to incorporate a compiler framework like LLVM [LA04] into the code generation process to provide optimizations and an high-quality machine code generator. DBILL [Lyu+14] implemented this idea based on HQEMU [Hon+12], but generally suffers from the same limitations as HQEMU. Moreover, the missing support for floating-point operations due to the use of TCG excludes a whole class of applications.

By leveraging Instrew (see Chapter 4) and adding an API to enable program instrumentation, these limitations can be avoided, yielding a high-performance framework allowing heavy transformations combined with a high-quality optimizer and code generator.

5.2 State of the Art

Similar to binary translation, binary instrumentation tools can perform code modification either statically, ahead of the program execution, or dynamically, during the execution of the program, and the same advantages and disadvantages apply, as discussed previously in Section 4.2. Consequently, most widely used tools for binary instrumentation perform such operations dynamically.

However, as binary instrumentation tools generally target the same target architecture as the code was previously compiled for, it is not necessary to abstract from the guest architecture: tools can also employ an architecture-dependent IR to perform code modifications, for example by representing storing the original instructions in the IR or by avoiding a higher-level IR at all and operating directly on machine code.

The internal code representation has a direct impact on instrumentation possibilities, the instrumentation time, as well as the feasibility of optimizations and code generation. IRs with sufficient abstraction of the underlying ISA allow for more

heavy transformations and thereby also for better optimizations, as tools do not need to care about low-level encoding limitations or register availability. However, such tools also face the problem to generate entirely new code from the abstract IR, often resulting in considerable overhead over the original code.

On the contrary, IRs that are strongly tied to the target architecture can typically reuse the original code and therefore avoid the overhead of instruction selection; but at the same time, the missing abstraction increases the difficulty to perform actual code modifications and subsequent optimizations, leading to high overhead with many code modifications.

The remainder of this section will describe state-of-the-art frameworks for binary instrumentation, classified by the level of abstraction from the guest architecture.

5.2.1 Frameworks with Architecture-independent IR

One of the most popular and widely used frameworks for heavy-weight DBI is Valgrind [NS07b]. Internally, Valgrind implements an architecture-independent SSA-based IR named VEX, which can represent code chunks with superblock granularity. The relationship between the VEX code and the original code is made explicit by special instructions to *get* or *put* values from/to an emulated register file. Before code is compiled back to machine code, some lightweight optimization passes are applied, for example, elimination of dead or redundant instructions. The instrumentation tool operates in the same address space as the guest program and can arbitrarily modify the VEX code. In addition to inspecting, adding, and modifying VEX instructions, a tool can also insert special calls to external helper functions.

DBILL [Lyu+14] is a modification of HQEMU [Hon+12] to base the instrumentation process on LLVM. As with HQEMU, the LLVM-IR representation for machine code is derived by lifting chains of TCG code. However, as instrumentation at the level of LLVM-IR prevents using a bypass using the TCG code generator and instead demands all code chunks to be lifted to LLVM-IR regardless of profiling information. As TCG does not natively support floating-point operations, but instead relies on external helper functions to provide the semantics, DBILL does not support applications involving floating-point arithmetic.

5.2.2 Frameworks with Architecture-dependent IR

For binary instrumentation without a higher-level abstraction from the target architecture, several approaches and tools exist. These can be classified in dynamic tools, which perform instrumentation during program execution, and static tools, which instrument the binary code completely prior to the execution and produce a new, modified binary file. Particularly static tools are primarily focused on instrumentation in the context of enhancing program security.

Dynamic Instrumentation Pin [Luk+05] is a DBI system that allows instrumentation tools to insert calls to helper functions at arbitrary points in the program. Internally, the original code is not lifted to an abstract IR, but stored as decoded machine instructions in basic block chunks. Recent versions of Pin [Int20b] perform simple optimizations, such as inlining calls to simple instrumentation functions, whereas for functions with a non-trivial control flow a call instruction is inserted. While the tool API also provides simple means to modify instructions of the program, they are rarely used.

DynamoRIO [BGA03; Bru04] was originally developed as a system for transparent dynamic binary optimization, but shifted its focus to binary instrumentation. Internally, the instructions are represented as decoded machine instructions. Tools can modify these instructions directly while observing encoding restrictions of the target architecture, or can use DynamoRIO's architecture-independent API, which provides common operations for multiple architectures. DynamoRIO itself does not perform any optimizations after the instrumentation.

HDTrans [Sri+06] focuses on reducing the translation overhead by optimizing the instruction decoding and the mapping to possibly modified instructions using lookup-tables. Additionally, basic blocks are dynamically combined to traces, which can also cover the targets of indirect jumps. Program instrumentation, however, is only possible by exchanging entries in the lookup-table for specific instructions; general modifications of code and therefore more complex instrumentation payloads appear to be unsupported.

DynInst [BM11] follows an entirely different approach to dynamic instrumentation: instead of rewriting the entire application code, an external tool can actively and dynamically replace individual functions. This requires the tool to have deeper knowledge of the application code, which it can gain via binary analysis. The instrumentation tool runs in a different process, code modifications of the application process are performed using debugging mechanisms like `ptrace`. For actual code modifications, an architecture-independent API is provided, but instructions are not lifted to an IR independent of the target architecture.

Static Instrumentation One of the first systems for performing static binary instrumentation was ATOM [SE94], which allows the insertion of function calls at arbitrary points in a program compiled for Alpha CPUs. Simple optimizations are applied to reduce the call overhead of instrumented helper functions.

With the appearance of DBI systems, the focus of static instrumentation tools shifted towards security applications. One popular motivation for several tools is to harden applications by enforcing control flow integrity or sandboxing policies, thereby increasing the difficulty of exploiting security vulnerabilities [Aba+05; MM06; ZS13; Haw+17]. Some of these approaches, however, require the use of a specific compiler setup to have sufficient information about the machine code to

perform their operations. Another use case is to enforce policies on unknown and potentially adversarial binaries [War+12; Zha+14].

Such hardening tools, however, did not find wider usage and more recent approaches for enforcing control flow integrity are based on compiler or CPU extensions [Cor19]. Further, recent versions of Linux include multiple mechanisms to enforce general security policies, like restricting system calls with an arbitrary filter with *seccomp* [Cor12].

5.3 Instrumentation Tool API

Instrew supports tools for the purpose of enabling modifications of the translated code and exposes this functionality as API for use by tools. Such code modifications are performed in the server process, and in particular between the step of lifting machine code to LLVM-IR and the code generation step, to make effective use of LLVM's optimizations.

To achieve this goal, an *Instrew tool* is a shared library object, which is dynamically loaded into the server on a request by the client. A tool is required to specify two functions: first, it has to provide an initialization function. This function can parse arguments passed to the tool, indicate compatibility with specific lifting modes — for example, to request markers about instruction boundaries —, and can load an initial set of tool-library functions into the client process; and second, a tool has to provide an instrumentation function, which can arbitrarily modify a function of LLVM-IR code lifted by Rellume. In particular, the tool can also inline helper functions and can run additional optimization passes.

As the tool itself has no access to the client memory, as it resides in the server address space, a tool can specify a set of functions in LLVM-IR to be compiled and sent to the client before the program execution starts. These functions can be called from subsequently instrumented code. For example, a set of "tool library" functions can be compiled once, including complex code paths, so that cold code does not have to be re-compiled for each instrumented function. To keep tools target-independent, library functions are compiled dynamically from LLVM-IR code, which can be derived from other programming languages, like C, using a suitable compiler, like Clang [LLV20a].

For storing data, tools are provided with two non-exclusive options: the first option is a small area with a size of 48 bytes in the CPU state, which is always available. As this is usually not sufficient, the second option are functions for manual allocation of larger memory regions. As the client only provides very simple means for memory management via the `mmap`/`munmap` system calls, a tool currently has to manage memory manually. Enhancements of the tool API to simplify memory management are subject to future design work.

A tool can intercept system calls by wrapping all calls to the system-call-emulation

function in the client with an own function. This way, it is possible to detect all I/O operations and also the program exit, which, for example, can be used to write data collected during the execution.

Following the fact that Instrew is also a DBT system, the instrumented code can also be executed on a different architecture. For example, RISC-V code can be instrumented in the architecture-independent LLVM-IR, and this instrumented code can be then compiled for and executed on a x86-64 machine. This flexibility also enables different use cases for architecture and compiler development, as the target architecture of the static compilation no longer has to match the architecture where the code will be executed.

5.4 Evaluation

The overall performance of Instrew is evaluated using the SPEC CPU 2017 [Sta21] benchmark suite, which covers a wide range of workloads derived from real applications. The widely used instrumentation tool Valgrind [NS07b] is used as main comparison, as it has very similar capabilities for instrumentation and code transformation. As an additional comparison, Pin [Luk+05] and DynamoRIO [BGA03] are employed due to their widespread use. However, note that this is not a fair comparison, because Pin and DynamoRIO have a smaller scope of possible code modifications.

For the actual instrumentation, two cases are considered: in the first case, the instrumentation systems were configured to perform an identity transformation without actual modifications — this is an extreme case to evaluate the overhead introduced by the instrumentation system itself. In the second case, an instrumentation tool that counts the number of dynamically executed instructions is used. These tools are in all cases designed to update a counter in memory at the end of each basic block. This is by no means the optimal way to collect this information for all of instrumentation systems, but represents the case of a naively written tool without complex optimizations.

5.4.1 Setup

All experiments were conducted on a platform with the x86-64 architecture. The dual-socket system is based on Intel Xeon CPUs (E5-2697 v3, Haswell), 17 MiB L3 cache and 64 GiB main memory, running SUSE Linux 15 SP1 with Linux kernel 4.12.14-197.40 in 64-bit mode.

All benchmarks are run with the reference input workload size in single-threaded mode; benchmark 600.perlbench was excluded due to use of x87 FPU instructions and unsupported system calls. All code is compiled with GCC 9.2.0 and optimization level `-O3`, benchmarks are statically linked against glibc 2.32.

For Instrew, commit `1cdb1bc`[1] is used together with LLVM 11. Based on the results described in Section 4.4, Instrew is configured to use the *HHVM* calling convention. For Valgrind, version 3.15.0 is used, for Pin, version 3.18 is used, and for DynamoRIO, release 8 is used.

For the instruction counting tool, the example tools of Pin (`inscount2_mt`) and DynamoRIO (`inscount`[2]); due to a lack of a sufficiently simple example without easily avoidable overhead, Valgrind was not evaluated.

5.4.2 Results without Instrumentation

The performance results for the first case without actual program modifications are shown in Figure 5.1[3]. With Instrew, the execution is on average 59% (int: 71%, fp: 48%) slower than the natively executed code. This is significantly lower than the overhead of Valgrind with 547% (int: 275%, fp: 791%), but also higher than the average overhead of Pin with 26% (int: 40%, fp: 13%) and DynamoRIO with 19% (int: 25%, fp: 13%).

The main source of overhead in Valgrind is the abstraction from the original code to its architecture-independent IR, VEX, in combination with only few and light optimizations during code generation. One source of overhead, which particularly affects the floating-point benchmarks, is the handling of vector instructions: such instructions are split up into more simple instructions during the lifting step from machine code to VEX, but these sequences are not recombined to the original instruction during code generation. Instrew also split up complex instructions into sequences of simpler LLVM-IR instructions, but the LLVM code generator later is able to merge these again, resulting in a significantly lower performance impact.

Without performing actual instrumentation, Pin and DynamoRIO are generally faster than Instrew on nearly all benchmarks, as their smaller transformation scope allows them to reuse large parts of the original machine code. On the benchmark 621.wrf, the Pin-based execution is even faster than the native execution, most likely because of tracing optimizations.

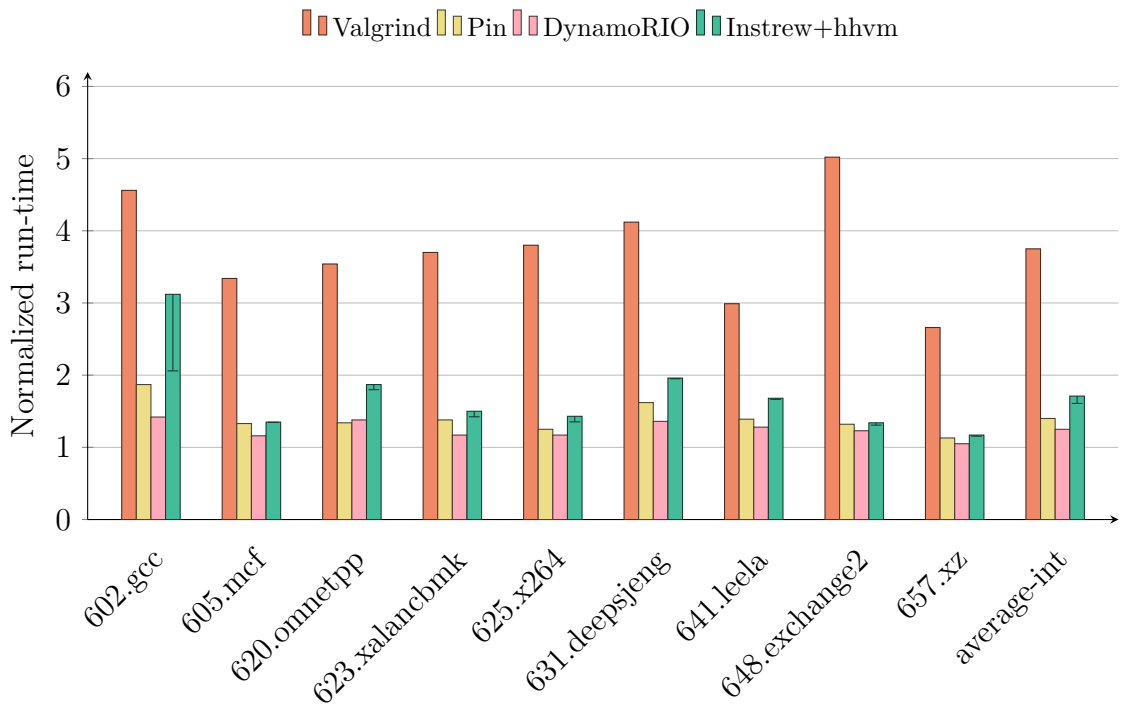
5.4.3 Results with Instrumentation

The performance results for the case with an instrumentation that counts the number of executed instructions are shown in Figure 5.2. With Instrew, the average

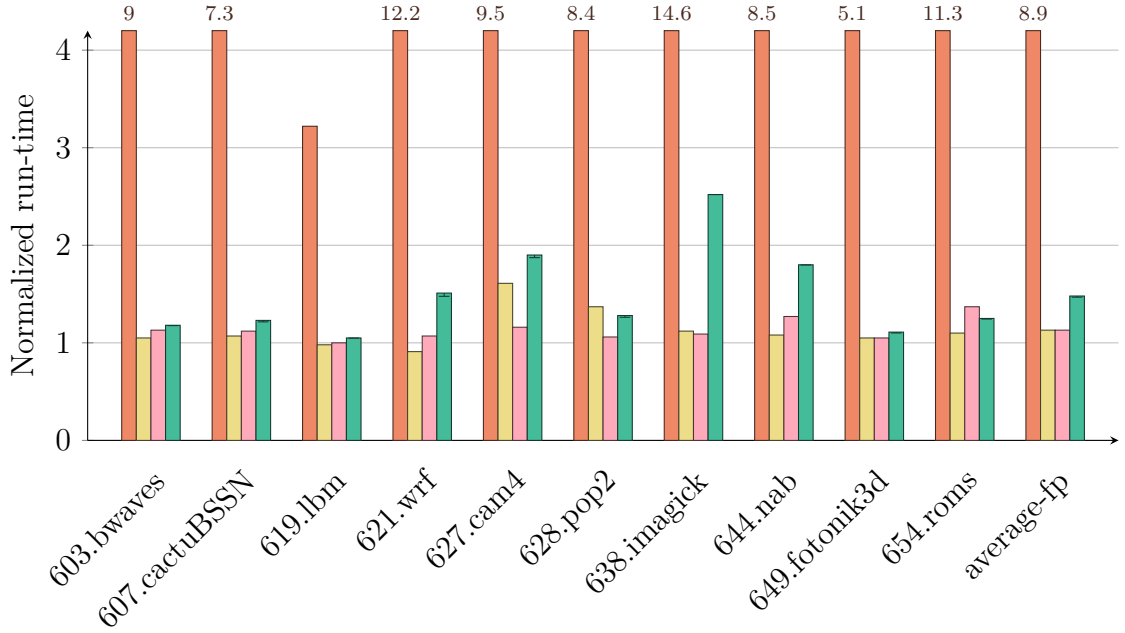
[1] `https://github.com/aengelke/instrew/tree/1cdb1bc`

[2] The tool was patched to print the number of instructions to `stderr` instead of `stdout` to not interfere with the SPEC CPU2017 benchmark infrastructure.

[3] Note: in literature, the mean is usually computed using the *geometric mean*, however, this thesis uses the *arithmetic mean* (average) to account for the fact that there is no multiplicative relation between different benchmarks. Further differences from numbers previously published in [ES20a] are caused by further performance optimizations and a newer version of LLVM.



(a) *Results for SPEC CPU 2017 Integer benchmarks.*



(b) *Results for SPEC CPU 2017 Floating-Point benchmarks.*

Figure 5.1: *Performance results on x86-64 with no instrumentation, normalized to the execution of the natively executed code. The relative translation time for Instrew is shown using error bars.*

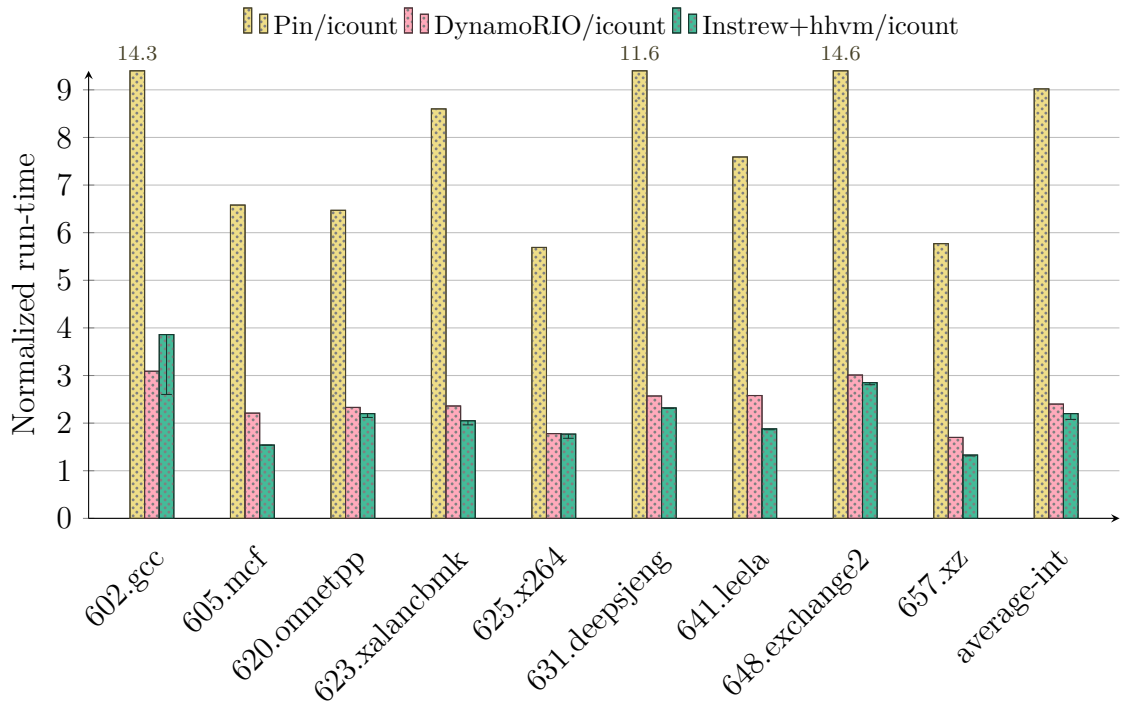


Figure 5.2: *Performance results on x86-64 with instruction count instrumentation, normalized to the execution of the natively executed code. The relative translation time for Instrew is shown using error bars.*

execution overhead increases to 120% on the integer benchmarks. However, the systems with an architecture-dependent code representation are generally slower in this case, with Pin having an overhead of 802% and DynamoRIO having an overhead of 140% on the integer benchmarks.

Due to the lack of optimizations in DynamoRIO and Pin, which additionally appears to fail inlining the instrumentation payload, Instrew has the lowest instrumentation overhead on nearly all benchmarks. One particular exception is 602.gcc due to the high compilation times as a consequence of the code size.

5.4.4 Discussion

As a consequence of the optimized machine code generation, Instrew is notable faster than Valgrind, even when no code modifications are performed. This shows that for a DBI system with a high-level and architecture-independent IR, the use of a compiler infrastructure, like LLVM, is highly beneficial, despite the higher overhead for compilation.

When no instrumentation is performed, tools like Pin or DynamoRIO have a low overhead of the native code, as they not only avoid the compilation overhead, but

also reuse most of the original instructions. With Instrew, in contrast, LLVM has to reconstruct a matching machine code sequence from the LLVM-IR instruction, which may be slower than the original sequence. Thus, it is not unexpected that Instrew has a higher overhead than DynamoRIO or Pin in this case.

However, the picture changes when actual code modifications are performed: in these cases, LLVM's optimizations result in a lower overhead than DynamoRIO, which merely connects instructions, but does not optimize code sequences. This shows, that already for simple program instrumentations, the use of LLVM and an abstract IR leads to a lower performance impact than systems that only operate at instruction-level, provided that the overall program execution time can amortize the optimization overhead.

5.5 Summary

This chapter described the instrumentation capabilities of Instrew and the API provided for instrumentation tools. With this approach, instrumentation can be applied at LLVM-IR level independently of the actual CPU architecture and LLVM's optimizing code generator can be used for to improve the efficiency of the instrumented code. Furthermore, this approach enables Dynamic Binary Instrumentation even when the architecture of the program and the host CPU architecture do not match, as Instrew will automatically compile code for the correct architecture. Performance results on the SPEC CPU2017 benchmarks show Instrew has an average overhead of only 59% over the native execution, which is significantly less than the state-of-the-art tool for heavy binary instrumentation Valgrind with an overhead of 547%. Moreover, already for light instrumentation use cases, Instrew is even faster than the instrumentation systems Pin and DynamoRIO, which use an architecture-specific IR.

6 BinOpt: Application-guided Runtime Binary Specialization

The last two chapters described the application of binary rewriting transparently to the application to reduce overhead when performing dynamic translation or instrumentation. However, when exploiting information that is only available during program execution, run-time binary rewriting can also be used to perform further optimizations, improving application performance beyond the possibilities of static compilation. Especially in High-Performance Computing (HPC), sources of such information are manyfold and resulting performance improvements can improve the efficiency of the overall application. This chapter describes a library-based programming model for specializing program code to such run-time information, named *BinOpt*, together with three different strategies to use this information for actual optimizations. Performance results show that even on real-world code significant performance improvements can be achieved.

6.1 Motivation

Performance is crucial for many applications, especially in High-Performance Computing (HPC), but also in other computationally-intensive domains like image processing. Therefore, recent optimizing compilers support many different optimization strategies, aim at improving the use of compute capabilities offered by modern hardware architectures. Such optimizations include different strategies for loop transformations, automatic vectorization, and folding of expressions and constant data.

The traditional approach for producing optimized machine code consists of two strictly separated stages: initially, at *compile-time*, the source code is analyzed, optimized and executable machine code is generated for the target platform. Depending on the size and structure of the source code, optimizing compilation may take a considerable amount of time. Afterwards, at *run-time*, the previously generated machine code is loaded into memory and executed. The time required for loading is usually negligible and the same binary file can be executed several times. To avoid frequent re-compilation, programs typically provide configuration options to operate on different inputs or otherwise adapt to a changed environment.

However, the approach of strict separation into the two phases has a major

drawback: available information at compile-time is incomplete, limiting possibilities for further transformations and inhibiting opportunities for optimization. Examples for such missing information are manyfold:

- High-level programming models for distributed systems usually determine their data layout at run-time, adding an extra level of indirection, which cannot be resolved during compilation.

- Applications using multiple processors regularly use dynamic load balancing to improve resource utilization — at the cost of another level of indirection.

- Many applications or application parts have an initial setup phase, processing configuration options and other input data. Resulting information, like matrix sizes, is only available at run-time, preventing constant propagation and the elimination of dispatch functions among other optimizations.

- Often the actual hardware configuration is unknown during compilation, forcing the compiler to make conservative assumptions about available hardware features, like ISA extensions or the system topology — with the consequence that available hardware resources are not utilized.

Although it is possible, in theory, to generate many variants of the code at compile-time, this approach is not only highly impractical, because code size and compilation time grow with the number of variants, but also very limited, because in many cases the possible range of specializations is simply too big to cover all likely variants.

Run-time Code Specialization Effective utilization of hardware features requires a run-time component that produces *specialized variants* for performance-sensitive code regions as required, making use of run-time-only information for optimization.

The simplest strategy to detect potentially performance-sensitive parts is to rewrite all parts of the program as they are executed, similar to dynamic binary instrumentation systems. Obviously, with this strategy many code parts with limited potential for performance improvement will undergo the rewriting procedure, causing avoidable overhead. A refined strategy for transparent optimization is to profile the application while running and trying to identify performance-sensitive regions. Essentially, this replaces overhead of code rewriting with profiling overhead.

In fact, both strategies suffer from not *knowing* run-time constant information and performance-sensitive code sections. However, application developers *do have* this knowledge. The only way to reliably exploit this knowledge is that the code specialization process is explicitly guided by the application itself — a strategy henceforth referred to as *application-guided optimization*.

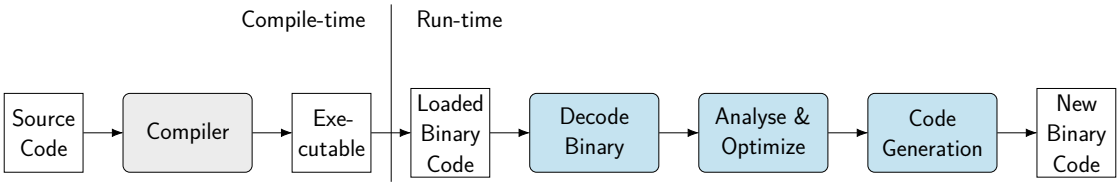


Figure 6.1: *Overview of the dynamic optimization process based on binary rewriting. The machine code loaded at run-time is decoded, tuned for run-time-only information, and then compiled again to new, optimized machine code.*

Application-guided Optimization Optimizing a section of machine code by specializing it for a given run-time setting can be achieved using three general methods. The first option is to simply trigger a recompilation at run-time, providing more information to the compiler. This approach, however, comes with substantial limitations: sources or suitable compilers may not be available at run-time, especially in HPC environments, and in addition, high compilation times are likely to negate performance benefits. Using some intermediate representation instead of the original source code suffers from similar problems, with the addition of being hard to integrate into existing build systems.

The second option consists of modifying the program to make use of JIT-compilation libraries, for example as provided by the LLVM [LA04] framework, to specifically generate optimized code for performance-critical sections. While this approach clearly allows for the highest performance benefits, it is also requires high effort in development and maintenance and also does not allow for the optimization of code residing in other shared libraries.

The third option is to optimize compiled machine code, which is always available as it must be executable for the CPU. While with this approach program semantics must be reconstructed from machine code, it also has several other benefits: it is independent of the used compiler, permitting the use of vendor-supplied compilers, and further allows optimization across boundaries of different programming languages and even pre-compiled shared libraries. Moreover, specialization on binary-level has the benefit of shifting several expensive compilation steps to compile-time, including the initial selection of instructions and registers, reducing the run-time overhead of code generation.

By making use of ABI guarantees at function borders, a binary specializer can be easily exposed as a library, as first shown by Weidendorfer et al. [WB16]. An application can specify a function as optimization target together with further information about known parameters or memory regions. The library can then optimize the function an create a specialized variant. Figure 6.1 shows an overview of the optimization process.

6.2 Optimization Opportunities and Limitations

Conceptually, many optimizations performed by a compiler can also be applied to machine code. However, possibilities for transformations are more restricted, because a considerable amount of semantic information from the source code is lost during the process of machine code generation. Prominent examples are information on data types, the layout of the stack frame, and pointer aliasing. Further, while ordinary compilers are already constrained by the semantics of the underlying language specification, like C or C++, machine code has even tighter semantics, and transformations can only be performed if they can be proven to be correct, i.e., the observable state does not change.

6.2.1 Beneficial Transformations

As a consequence of the semantic limitations, optimization possibilities are in general either low-level or require sophisticated analyses to prove correctness of the transformation in a particular setting. In different contexts, the following transformations have shown to be possible and in many cases beneficial, roughly ordered by decreasing complexity:

- *Loop transformations* [AC71] (e.g., loop unrolling, loop fusion) eliminate loop overhead and reduce the amount unproductive instructions and branches. Optimizations like loop unrolling are more beneficial on small loops where the loop overhead is comparably high; excessive loop unrolling, however, is usually problematic due to size constraints of the instruction cache.

  Run-time knowledge of iteration count, e.g., due to constant propagation, or a size reduction of the loop body, e.g., due to dead code elimination, may enable this transformation.

- *Vectorization* [Mal+11] improves usage of available SIMD computation units.

  This optimization may become beneficial or an existing vectorization scheme can be improved as a consequence of prior loop transformations, dead code elimination, and constant propagation.

- Improved *register allocation* [AC71] is important to reduce memory accesses by keeping as much working data as possible in registers directly in the processor core.

  Following constant propagation more registers become available, which can be used to avoid spilling data on the stack; and as a consequence of loop transformations, more available registers may allow for exposure of more parallelism to the processor and help to avoid stack spilling.

- Improved *instruction scheduling and selection* [AC71] to the target micro-architecture may allow for a faster program execution and better utilization of compute resources, for example as offered by available ISA extensions.

  This optimization is almost always beneficial, namely if any part of the code has changed during another transformation or the target processor has been unknown at compile-time.

- *Constant propagation* [AC71] folds operations with all-constant input data to another constant value, reducing performed computations, code size, and eventually also memory accesses. This optimization may enable many other transformations.

  In the context of run-time specialization, this is the key transformation which allows to exploit the benefit of run-time-only data by propagating it through the program code and thereby triggering other optimizations.

- *Dead code elimination* [AC71] reduces the code size and eliminates branches.

  Elimination of conditional branches with a known condition is directly enabled by constant propagation. While on modern processors the direct impact is comparably low due to improved branch prediction logic and trace caches for instructions, a code size reduction may have an impact on heuristics for decisions on other transformations and reduces code generation time.

- *Inlining* or procedure integration [AC71] reduces the overhead incurred by the call/return sequence and may enable other optimizations like constant propagation.

  While compilers generally perform inlining as well, it can easily be the case that inlining previously was considered as non-beneficial — changed circumstances, like a smaller code size of the callee, may result in a changed estimate. If the call target is not statically known, inlining may only become possible at run-time as a result of constant propagation.

6.2.2 Limitations

Run-time optimizations are not only constrained by the increased effort of reconstructing information about the stack frame and proving correctness, also their usefulness and performance benefit imposes practical limitations.

In general, when performing run-time optimizations, the performance gains must outweigh the transformation effort. In particular, this is only the case if the optimized code is executed multiple times: the cost for decoding, analyzing, and code generation in software is several times higher than executing these instructions once. Note that a loop inside the transformated code suffices for executing the code

multiple times, in which case, however, rewriting the code part outside of the loop contributes overhead which needs to be amortized.

This observation has immediate consequences for the application of run-time binary optimization: first, the technique has to be applied very selectively to only those code portions, where a high benefit can be achieved. Second, the data for which a specialization is created must not change too often, otherwise performance gains are negated by too many rewriting operations.

Another limitation arises for the kind of information that enables optimization: almost all transformations that benefit from available constant data only benefit from *integer constants*, for example loop iteration counts or indices into data structures. Floating-point constants, on the other side, are hard to propagate due to restrictive semantics, and even if unsafe optimizations were allowed, it is rarely the case that a significant amount of computations can be eliminated. Additionally, several widespread architectures have only limited (e.g., AArch64 [Arm18]) or no support (e.g., x86-64 [Int20a]) for encoding floating-point constants directly in the machine code as immediate operand.

Corollary: The described approach for run-time optimization is *not* a general-purpose optimization and also *not* suitable for all kinds of constant data. Its applicability depends on the type of the known data, its use, and the overall workload size.

6.3 Configuration Data

The approach of application-guided dynamic binary optimization requires knowledge about run-time data, which does not change throughout the execution of a part of the program (or at least is unlikely to do so). Essentially, in many cases such data is some kind of configuration to the program or a kernel included therein. In the following, the term *configuration data* will be used to refer to all kinds of run-time-only data. Such configuration data may have a variety of sources and lifespans:

- *Machine specifications* like processor features or network topologies are available from the early beginning of the program and is always[1] constant throughout the entire execution of the program.

  This kind of information is generally available and typically used by specialized libraries like Basic Linear Algebra Subprograms (BLAS) [Bla+02] and run-time systems like Message Passing Interface (MPI) [Mes15] implementations.

[1] Exceptions are programs which support transparent movement between different nodes.

- *Program arguments,* for example as specified on the command-line, are a very common form of constant-data, for example, matrix dimensions. These may include scalar constants, but also files or even code fragments like plug-ins. This kind of configuration data is constant across the whole execution of the program.

  Information about the relevance and impact of program configuration options are usually only known to the application developer.

- *Run-time systems* like the one of an Open Multi-Processing (OpenMP) [Ope20] implementation or an MPI [Mes15] library support configuration parameters, dynamically configurable data structures, for example, MPI data types, and tools which can modify their behavior. Usually such data and tools are processed at program start and remain unchanged throughout the entire execution.

  This information is only partially known to the application developer, but always known to the developers of the run-time systems. Some parts of this configuration like tools are usually *transparent* to the application.

- *Data from set-up phases* is typically created by preprocessing input data, processing configuration options, or initializing other data structures. Configuration data produced in such a phase is usually constant throughout the (typically longer) run of the actual computation routine.

  This kind of information is usually unique to the application developer.

- Dynamic load balancing enables more efficient use of compute capabilities by re-distributing and re-organizing data structures during execution. Therefore, the *run-time data layout* and data distribution can change, either triggered explicitly by the application or transparently by a run-time system. This may also have an impact on the actual data layout; for example, halo regions may be stored differently to simplify communication patterns. Typically, run-time systems try to keep this information is constant over longer time spans to avoid overhead of frequent changes, but there are no strict guarantees.

  This information is usually only known to the developer specifying the data layout and defining trigger points for layout changes. This may be the application developer or the developer of the run-time system, and this information may also be transparent to the respective other component.

- *Parameters of compute kernels* may differ for each run of the kernel, and such a kernel may be called multiple times in the course of a computation routine. Such parameters can, for example, describe sizes of matrices or specify an exact data layout, and are constant throughout the execution of the kernel.

  This constantness of this information is usually known to the kernel developers.

6.4 Use Cases

In the following, different application scenarios that contain high-impact configuration data, so that they can potentially benefit from run-time specializations, will be described.

Image Processing Kernels Image processing operations, like color filters, blur filters, or affine transformations, typically include many configuration options, which are specified when the kernel is called. Consequentially, such options often determine varying code paths, where run-time checks can be eliminated, indirect function calls, which — if resolved — can be inlined, and varying operation region sizes, where the operation loops may be unrolled and vectorization may be used more effectively. Additionally, also the support of newer ISA extensions may be used to further specialize code for the currently available processing facilities.

Linear Algebra Kernels Implementations of linear algebra functions, for example the BLAS [Bla+02] functions, include several configuration parameters, like the size of the input vectors or matrices, stride sizes, and matrix layout. These configuration options must be checked for validity and optimized implementations typically dispatch internally to a specialized implementation for a given configuration. While the hereby incurred overhead can be negligible for larger operations, it can add up when many small operations are executed. As the configuration parameters are usually known (and therefore their validity), the initial check and dispatch can be skipped, directly jumping into the computation part of the kernel.

A more advanced binary optimizer may even go one step further and optimize the actual computation part of the kernel. However, as such routines are usually highly optimized and may even involve hand-written assembly, care must be taken not to create a less performant specialization.

Sparse Data Structures Sparse data structures have one or more internal layers of indirections to reduce their memory footprint, often involving some kind of indexing with integer data, e.g., in the Compressed Sparse Row (CSR) format [Saa00]. In many cases, also the control flow depends heavily on the data and may be hard to predict for hardware. Specializing operations on sparse data structures has the potential to yield high performance improvements: indirections can be removed and integer offsets or control flow decisions can be directly encoded in machine operations, reducing irregular memory access patterns and avoiding unpredictable branches. Further, depending on the actual indices, it may be possible to avoid expensive gather/scatter operations and instead make use of contiguous memory accesses. Subsequently, for example, this may also enable improved use of Single Instruction Multiple Data (SIMD) extensions. However, specialization is still only

useful if specialized operation is run multiple times, for example, when using the same sparse matrix for several multiplications.

Run-time-specified Data Structures While the layout of data structures is known for applications, either directly or through template instantiation, this may not be the case for libraries or run-time systems. These need to adapt to data structures configured at run-time. A prominent example are MPI data types, which are configured using an API before they can be used in communication operations [Mes15]. Also other libraries for data serialization face a similar situation. In these cases, once the actual data type is configured completely, the packing and unpacking code can be specialized. This optimization is more beneficial for small data structures with non-contiguous data, as for large structures the performance of such code is limited by memory bandwidth.

Plug-ins Many applications and libraries provide plug-in mechanisms for external tools to monitor or modify program behavior. Such plug-ins are usually realized by registering callback functions, which are called under specific conditions, or by intercepting libary calls using mechanisms of the dynamic linker[2]. In both cases, the underlying mechanism consists of one or more indirect function calls. Such tool interfaces also exists in run-time systems in HPC environments, for example the OpenMP tool interfaces (OMPT, OMPD) [Ope20], which rely on callbacks, or the MPI profiling interface [Mes15], which relies on the run-time linker to intercept calls to MPI.

In performance-critical sections, such callbacks can add up to a significant overhead, which can be eliminated by resolving indirections and optionally also inlining the code of the plug-in.

Auto-tuning Automatic tuning of applications and libraries is a widely researched field [TCH02; PE06; CH15; MFG16]. Such frameworks strive to optimize several tuning parameters of applications and libraries to optimize for one or more objectives, typically including performance. Among other tunable parameters, such systems usually include dynamic parameters of the application and their libraries as well as optimization options used during compilation in their search space.

Using run-time optimization, the overall performance of the tuning process can be improved. Kernels can be specialized for dynamic parameters, avoiding extra overhead from the tuning framework as the measured code is more similar to what a compiler would generate. Further, and more importantly, compilation options, like the loop unrolling threshold, vectorization strategies, or target-specific optimizations, can be modified on a per-function basis without the need to recompile the entire program or library, optimizing the tuning process itself. Moreover, when the original

[2]Example: by this means the heap allocator can be exchanged.

code makes use of data type abstractions, the data layout can be efficiently changed without re-compilation by replacing and inlining accessor functions at run-time.

6.5 Library Approach

The approach of application-guided dynamic binary optimization can be implemented using an API. The developer of an application or library explicitly specifies functions designated for optimization, specifies the configuration data for use with optimization, and triggers the optimization process itself. The result is a new function specialized for the configuration data, which can be subsequently used instead of the original function.

With DBrew [WB16] and Drob [Hil19], two library-based approaches have been proposed already. However, the libraries not only have different optimization scopes and strategies, but also have different APIs, making a comparison of the approaches significantly more difficult.

This motivates the development of a new *unified API*, which provides a common interface for different optimizers. Developers can then use this unified API instead of the rewriter-specific APIs to easily substitute the optimizer, allowing an easy comparison without requiring substantial code changes.

6.6 Unified API

The API design of a library very important for its usage, especially for a library which enabled run-time rewriting of user-specified compiled functions. This thesis proposes a *unified API* for function-level binary rewriting, allowing multiple implementations of binary rewriters/optimizers to be used without the need to modify the program code for different rewriters.

In particular, the following requirements for the design of the API are especially relevant:

- The API shall be *general* for the user. From the user's perspective, it shall accept any function that follows the normal C ABI without further restrictions.
- The API shall be *fail-safe* and not require any error handling on the side of the user, reducing effort for adaption. Whenever the actual implementation is unable to handle a given function, the only option is to provide the existing function as result, without any modifications.
- The API shall be *future-proof*: it shall allow for a trivial implementation of the required functionality. This implies that using the API does not necessarily add a hard dependency on a rewriting system, reducing the adoption barrier.

For example, when the application or library needs to be ported to another architecture without an available rewriting system, API calls can be easily stubbed out with a no-operation implementation.

- The API shall be *flexible* for the implementation of a binary optimizer. From that perspective, the API shall not limit the capabilities of the concrete optimization library and also allow for library-specific configurations.
- The API shall be *compatible* with other compiled programming languages and not be restricted to C or C++. As many programming languages support bindings to other libraries following the C ABI for both directions, using library functions in the user's language and passing functions written in the user's language as function pointers to C functions, we deduce that the API must be compatible with C.

We designed an API matching these requirements and realized it in the library *BinOpt* [ES20b], which supports different rewriters using the same user-facing interface. The library currently is focused on the x86-64 architecture and the Linux operating system kernel, as this is the most widespread architecture for highly performance-sensitive applications, including many HPC platforms. The remainder of this section will detail the main functionality of the API and its impact on both sides, the user of the API and the developer of a rewriting library implements the API. Listings 6.1 and 6.2 show usage examples of the library, and Appendix B includes a complete description of the API.

To satisfy the criteria of compatibility with other compiled programming languages, the BinOpt API adheres the common subset from C and C++. To allow multi-threaded use, a user has to initially construct a thread-specific handle into the rewriting library. The handle can be released when no longer required.

API Design

The function `binopt_init()` *creates a new handle into the rewriter and the function* `binopt_fini(handle)` *releases the handle and all associated memory allocations, including rewritten functions.*

Rewriter Implementation

A rewriter must not share state between different handles, which implies that global variables cannot be used to store state. A rewriter does not need to free resources when the handle is finalized, but obviously is encouraged to do so.

All functions that are being rewritten must follow the C ABI, which on x86-64 Linux platforms is the System V ABI [Mat+14]. All data needed to generate a specialized variant of the function, including a pointer to the function itself, is stored inside a *configuration object*.

```c
int func(int a, int b) {
  return a - b;
}
int main(void) {
  // Create a new handle into the library
  BinoptHandle h = binopt_init();
  // Create a configuration for func
  BinoptCfgRef bc = binopt_cfg_new(h, func);
  // Specify signature, two parameters, int(int, int)
  binopt_cfg_type(bc, 2, BINOPT_TY_INT32, BINOPT_TY_INT32, BINOPT_TY_INT32);
  // Specify second parameter as constant int 42
  binopt_cfg_set_parami(bc, 1, 42);

  int(*nfn)(int, int) = binopt_spec_create(bc);

  // Call new version, uses 42 instead of 16
  int res = nfn(48, 16);
}
```

Listing 6.1: *Example usage of BinOpt. The library calls are used to specialize the function* ***func*** *for the constant second parameter 42. Note that the second constant* can *be propagated, but this is not required: if no optimization occurs for whatever reason, the returned function* ***nfn*** *is also allowed to use the specified parameter value.*

```c
int func(size_t len, const int* buf) {
  int res = 0;
  for (size_t i = 0; i < len; i++)
    res += buf[i];
  return res;
}
int main(void) {
  // Example array
  int array[] = { 4, 3, -1, 4 };

  // Create a new handle into the library
  BinoptHandle h = binopt_init();
  // Create a configuration for func
  BinoptCfgRef bc = binopt_cfg_new(h, func);
  // Specify signature, two parameters, int(ulong, void*)
  binopt_cfg_type(bc, 2, BINOPT_TY_INT32, BINOPT_TY_UINT64, BINOPT_TY_PTR);
  // Specify first parameter to array length
  binopt_cfg_set_parami(bc, 0, sizeof array / sizeof array[0]);
  // Specify second parameter as pointer and mark associated array as constant
  // This is the combined form of the following call sequence:
  //   binopt_cfg_set_parami(bc, 0, &array);
  //   binopt_cfg_set_mem(bc, array, sizeof array, BINOPT_MEM_CONST);
  binopt_cfg_set_paramp(bc, 0, array, sizeof array, BINOPT_MEM_CONST);

  int(*nfn)(size_t, const int*) = binopt_spec_create(bc);

  // Call new version, uses array values from above instead
  int res = nfn(2, (int[]) {3, 1});
}
```

Listing 6.2: *Example showing BinOpt usage for specifying a constant memory region.*

API Design

The function `binopt_cfg_new(handle, func)` *creates a new configuration object for the purpose of rewriting within the context of a* `handle` *for the function* `func`*, which will be referred to as* rewriting target*.*

Rewriter Implementation

A rewriter is allowed to already decode and analyze the function at this point.

Limitations of the C language require the presence of a way to specify the signature of the function explicitly — this is necessary because arguments and return values may be stored in different registers or the stack, depending on the parameter type. Currently the BinOpt API only supports simple types (e.g., integers, floating-point numbers, pointers), structure or union types must be transformed manually to the correct simple types according to ABI rules. In future, it may be possible to provide a C++ API which uses metaprogramming mechanisms or reflection to extract relevant information. Additionally, ongoing efforts in the C community to improve type-generic programming [Gus21] may allow for use of lambda expressions and automatic inference of the function prototype.

API Design

The function `binopt_cfg_type(cfg, nparam, retty, paramtys...)` *sets the type of the rewriting target, specifying the number of parameters, the return type, and the types of the parameters.*

Rewriter Implementation

A more sophisticated rewriter may also fetch this information from debug information sections like DWARF [DWA17] or Compact C Type Format (CTF) [14] and therefore not require this explicit configuration call or use it to warn if there is a type mismatch. As debug information sections are not necessarily present, the API function is strictly necessary.

An important source of optimization potential are constant memory regions. By default, all memory ranges that are mapped as read-only in the virtual address space of the program are treated as constant data. Other memory regions can be marked as constant explicitly in a configuration. If the automatic detection of read-only regions is not desired for some memory regions, for example due to read-only shared file mappings, regions can also be explicitly configured as dynamic.

API Design

The function `binopt_cfg_mem(cfg, addr, len, flags)` *configures a memory region, where* `flags` *is either* `BINOPT_MEM_CONST` *for constant memory,* `BINOPT_MEM_DEFAULT` *to depend on the page mapping provided by the kernel, or* `BINOPT_MEM_DYNAMIC` *to forcefully treat a region as volatile. Specified memory regions must not overlap. Initially, all memory is configured as* default.

Rewriter Implementation

A rewriter may use additional information about constant memory regions as soon as they are available.

Another source of constant data are parameters, which can be given a constant value or address after the function signature has been specified.

API Design

The function `binopt_cfg_param(cfg, idx, valptr)` *sets a parameter at a given index to a constant value, with* `valptr` *being a pointer to the constant value. Convenience functions are provided to set integer parameters directly and to associate a constant memory region directly to a pointer parameter.*

Rewriter Implementation

A rewriter may propagate information about constant parameters immediately after configuration for incremental optimizations.

Rewriters may support further configuration options, for example allowing the optimization of floating-point arithmetic (*fast math*), a threshold value for loop unrolling, and verbosity of debug information. Such options can be configured using rewriter-specific configuration options, which are exposed in a single API function.

API Design

The function `binopt_cfg_set(cfg, flag, value)` *sets a rewriter-specific option.* `value` *may also be a pointer to a more complex structure.*

Rewriter Implementation

By default, all configuration options must be set to the most restrictive supported option. This allows a rewriter to apply this information immediately for incremental optimizations. Unsupported options must be ignored.

Once the configuration is finished, a new specialization of the function can be generated, which has the same signature as the original function. In case of a

configuration or rewriting error, either a less optimized function may be generated, or the original function is returned directly.

API Design

The function `binopt_spec_create(cfg)` *returns a pointer to an optimized specialization of the function, or a pointer to the original function.*
The function `binopt_spec_delete(handle, spec)` *may delete a specialized function, or does nothing if the given function is not generated by the rewriter. Care must be taken when recursively specializing functions, as rewriting failure results in the returned "original" function being still generated by the rewriter, which may cause* double-free *errors.*

Rewriter Implementation

The rewriter should attempt to create an optimized specialization, but in unsupported or unexpected circumstances, it is always safe to return the original function.
Actually deleting unused specializations is not required, as this might be non-trivial to do, for example, if several code variants share the same memory region.

A configuration can be *cloned*, simplifying programming if multiple similar specializations of a function are required and at the same time allowing for further optimizations in the rewriting process.

API Design

The function `binopt_cfg_clone(cfg)` *creates a new independent configuration with the same state as the original.*

Rewriter Implementation

A rewriter is free to just clone the configuration, to clone an internal analysis/optimization state of the function, or even temporarily generate new machine code from the current configuration state.

Trivial implementation: Null rewriter The design criteria of the BinOpt API mandates that a trivial, yet fully-functional implementation must be possible. With the API described above, such an implementation is possible, even without any memory allocations: the *handle* is unused and the *configuration pointer* is a pointer to the rewriting target. All calls that modify the configuration are ignored. When a new specialization is requested, the pointer to the original function is returned without performing any modifications.

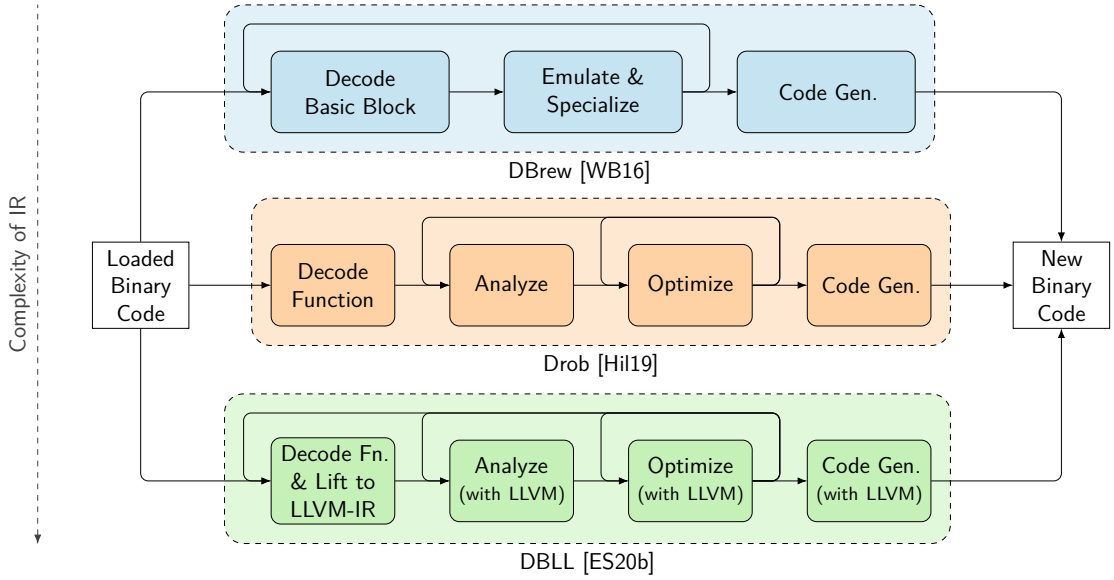


Figure 6.2: *Overview of different stages from the three rewriting approaches. Some stages may be executed multiple times, for example, when further code is discovered or previous analysis results are invalidated during optimization.*

6.7 Rewriting Strategies

The general strategy for run-time binary optimization consists of four main stages: first, the original machine code has to be *decoded* into a code representation that is used for analyses and transformations. Second, the decoded program is *analyzed* for transformation possibilities and optimization potential. Third, based on the analysis results *transformations* are applied. And fourth, new *machine code is generated* for the transformed program.

These stages are typically not completely separable, though: as a result of applying transformations, prior analyses of the second step may be invalidated, requiring the rewriter to redo some analyses throughout the transformation of the program. Further, these stages may also be interleaved, for example, when decoding only occurs for code that turns out to be reachable during analysis of previously decoded fragments.

The key factor for analyses and code transformations is the Intermediate Representation (IR) used to internally represent the program code: low-level IRs allow for fast transformations and code generation but make analyses more expensive and complex transformations more involved. Due to the lack of abstraction from machine code, transformations are rather low-level. High-level IRs, in contrast, simplify program analysis and enable more impactful transformations with low effort, but require more time for lowering to efficient machine code.

In order to understand the trade-offs between different IRs and rewriting strategies,

a direct comparison of different optimizers implementing such approaches is necessary. This section will describe three different approaches for performing function-level binary optimization. First, Section 6.7.1 will describe DBrew [WB16], a tracing binary optimizer, which performs strictly-local analyses and transformations on machine code similar to a tracing binary rewriter. Second, Section 6.7.2 will cover Drob [Hil19], a binary optimizer which performs function-level analyses while still being close to the original code for fast code generation. Finally, Section 6.7.3 will detail DBLL [ES20b], which lifts machine code to LLVM-IR [LA04] using Rellume (see Chapter 3) and re-uses the entire LLVM infrastructure for analysis, optimization, and JIT-compilation. Table 6.1 shows an overview of these approaches and Figure 6.2 depicts an overview of the rewriting stages.

6.7.1 Tracing Binary Rewriting: DBrew

DBrew [WB16; EW17] is a prototypical tracing binary rewriter for x86-64. A function is optimized by tracing through possible execution paths. On the way, instructions with all-constant input operands and conditional branches with a known branch decision are removed. This approach is inspired from tracing binary optimization and translation tools with the difference that the function and all functions called therein are processed completely ahead-of-time.

Publication Information

The information given in this sub-section is based on [WB16; EW17] and the DBrew sources.

6.7.1.1 Rewriting Approach

The rewriter steps through a function in basic block granularity, starting with the basic block at the entry address of the function. For each basic block, the rewriter keeps track of a *known-world state*, which stores the *value state* of registers and the stack frame, including possibly constant values. There are five possible value states:

- *dead:* this state is used for uninitialized values, for example non-parameter registers.
- *dynamic:* this state implies that no constant value is known at rewriting time, for example for unknown parameters or other values that are computed in the function with a possibly non-constant value.
- *static:* this state marks values that have a known constant value, which is stored in the known-world state. This is used, for example, for immediate operands or constant parameters.

Table 6.1: *Overview of rewriting approaches.*

	DBrew [WB16; EW17]	**Drob** [Hil19]	**DBLL** [ES20b]
Code Repr.	x86-64 Instructions	x86-64 Instructions	LLVM-IR
Scope	Instruction-local	Whole Function	Whole Function
Approach	• Decode machine code as needed (partial evaluation, emulate and capture) • All operands known ⇒ instruction is emulated, otherwise captured and kept • Remove jumps with known condition • Duplicate block if known values differ	• Decode function • Reconstruct full CFG • Lift instruction to Drob IR, keep register and instruction information • Analyze stack frame and register usage, tracking known constant values • Run optimization passes, updating analysis data when needed	• Lift whole function to LLVM-IR (Rellume) • Iteratively discover further code • Apply `-O3` LLVM optimizations with extra constant propagation from memory • Use LLVM-MCJIT to generate new code
Key Optimizations	• Constant folding • Full loop unrolling • Inlining	• Constant folding • Dead code elimination • Simple loop unrolling	• Constant folding • Loop transformations • Vectorization • Inlining • ...
Information Kept	• Instruction selection • Register allocation • Stack frame layout	• Instruction selection • Register allocation • Stack frame layout • Basic block ordering	• Stack frame layout • Instruction semantics only

- *static2:* this is a special state for pointers. Values loaded from an address derived from a pointer with the state *static2* are marked again as *static2*, enabling a simple way to mark memory regions as constant transitively.

- *stack-relative:* to keep track of offsets relative to the stack frame of the function, this state indicates that the known-world states stores a constant offset to the start of the stack frame.

When processing a basic block, the rewriter decodes all instructions and iterates over these. If all input operands of an instruction are constant, the values for the output operands are computed at rewriting-time, stored as *static* in the known-world state, and the instruction is removed. Likewise, if the instruction loads from a constant memory region or dereferences a pointer with state *static2*, the actual memory access is performed and the value is again treated as constant. Otherwise, an instruction is *captured* and emitted in the rewritten code again. If such a captured instruction depends on values emitted by previously removed instructions, the known constant value is either encoded as immediate operand or, if this is not possible, restored immediately before the captured instruction.

Function calls are simply treated as *push/jmp* combination, with the addition that the return address is stored in a shadow stack. When encountering a nested return instruction, the continuation address is taken from the shadow stack without further verifications. As indirect jumps or calls with a dynamic address are not supported, DBrew can eliminate the *push/jmp* sequence in many cases.

At the end of a basic block, the successors are determined from the last instruction. Basic blocks ending with a conditional jump generally have two possible successors (branch taken vs. fallthrough), unless the jump condition is known. In case of an unconditional jump or the absence of an instruction modifying the control flow (fallthrough), there is just one successor. A return instruction of the top-level function indicates the end of the function; a nested return instruction has one successor, which is determined by the address stored on the shadow stack by a previous call instruction. If a succeeding basic block was not yet processed *with the same known-world state*, it is added to the processing queue of basic blocks. Consequently, basic blocks are duplicated whenever a tracked constant value is modified.

After all basic blocks have been processed, the generator produces new machine code from the captured instructions for each basic block. As all transformations only emit encodable x86-64 instructions, machine code generation only consists of encoding the instruction in binary form. The generated machine code blocks are then connected using appropriate jump instructions, taking care of changed offsets.

```
1  sum_array: // rdi=len, rsi=ptr
2    xor eax, eax // accumulator
3    xor ecx, ecx // loop ctr.
4    jmp .Lloopcond
5  .Lloopbody:
6    add eax, [rsi + 4 * rcx]
7    add rcx, 1
8  .Lloopcond:
9    cmp rcx, rdi
10   jb .Lloopbody
11   ret
```

$\xrightarrow{\texttt{rdi}=3}$

```
1  sum_array_3:
2    mov eax, [rsi]
3    add eax, [rsi + 0x4]
4    add eax, [rsi + 0x8]
5    ret
```

(a) *Specialization of a function calculating the sum of all values in an integer array with a constant length 3. The loop is unrolled completely, the value of the then constant loop counter is propagated and the addition with zero is removed.*

```
1  acc_array: // rdi=len, rsi=ptr
2             // edx=accumulator
3    mov eax, edx // accumulator
4    xor ecx, ecx // loop ctr.
5    jmp .Lloopcond
6  .Lloopbody:
7    add eax, [rsi + 4 * rcx]
8    add rcr, 1
9  .Lloopcond:
10   cmp rcx, rdi
11   jb .Lloopbody
12   ret
```

$\xrightarrow{\substack{\texttt{rdi}=3 \\ \texttt{rsi}=\&\{4,5,6\}}}$

```
1  acc_array_456:
2    mov eax, edx
3    add eax, 4
4    add eax, 5
5    add eax, 6
6    ret
```

(b) *Specialization of a function accumulating the sum of all values of a constant integer array with a third parameter. The loop is unrolled completely and the memory accesses to the constant values are removed. This example shows the use of the* static2 *state: a pointer is passed as parameter and all subsequent dereferences are removed transitively. Expressions are not re-ordered, resulting in a series of addition operations, which could be folded to* `add eax, 15`*.*

```
1  foo: // edi, esi, edx are int
2    mov eax,esi
3    imul eax,edx // comp. edx*esi
4    cmp edi,0    // but if edi>=0
5    jl .Lret
6    mov eax,edi  // ..return edi
7  .Lret:
8    ret
```

$\xrightarrow{\texttt{edi}=6}$

```
1  foo_spec:
2    mov eax, esi
3    imul eax, edx
4    mov rax, 6
5    ret
```

(c) *Branches with a known constant condition are removed. The compare instruction is also removed because all operands are known constants. However, the now-unused preceding computational operations are not eliminated.*

Figure 6.3: *Examples of specializations generated using DBrew.*

6.7.1.2 Implemented Optimizations

With this approach, DBrew implements several simple, yet performance-critical optimizations.

Instructions with only constant input operands are removed and constants are propagated into instructions where possible. For some instructions, even further simplifications are applied: for example, an addition with the constant zero does not change the value and can be replaced with a simple move instruction (as shown in Figure 6.3a).

Loop unrolling is performed whenever the iteration range of a loop is constant. As the constant loop counter will be tracked in the known-world state, this state is different in every loop iteration, resulting in all basic blocks of the loop to be copied for every value in the iteration range. Examples are shown in Figures 6.3a and 6.3b.

Constant values from memory are *propagated* when a memory range is either marked as constant or the address has the state *static2*. For integer values, this causes memory accesses that result in a known constant value to be removed and replaced with a constant value encoded directly in the generated instruction stream, as shown in Figure 6.3b. For floating-point constants, no such optimization is implemented as x86-64 does not support immediate floating-point operands [Int20a].

Conditional branches where the jump decision can be determined from statically available information are removed and the only succeeding basic block is merged, as exemplified in Figure 6.3c. Due to the lazy processing strategy, code fragments that are known to be never executed are never added to the processing queue and therefore are not even decoded.

As function calls are essentially treated only as unconditional jumps, all function calls will be eliminated and *inlined* unconditionally. This aggressive inlining strategy strictly avoids overhead from call or return instructions, but may also significantly increase the code size and is problematic for recursive functions — rewriting recursive functions with possibly unbounded depth unconditionally results in a rewriting failure.

6.7.1.3 Limitations and Performance Considerations

DBrew implements a simple and straight-forward approach to apply simple and performance-effective optimizations. Due to the strictly local nature of the implemented transformations, no complex analyses are necessary, making the optimization process very efficient. As all instructions remain valid x86-64 instructions during all transformations, no further effort for re-allocating registers or selecting instructions is required, resulting in a low overhead for code generation.

This simplicity is, however, also the biggest drawback of DBrew: the lack of an overview on the whole function prevents an impact analysis of several transformations. For example, the strategy of complete loop unrolling may lead to a massive increase

of code size when the constant iterations range is large, and aggressive inlining additionally imposes limitations for recursive functions. A large amount of highly redundant generated code not only increases rewriting time, but also has a negative impact on the execution time of the generated code due to cache constraints. Limiting this code duplication would require merging known-world states with different constants, which may be non-trivial if the loop iteration cannot be identified easily.

Another missed optimization opportunity results from the absence of a liveness analysis of computed values. Instruction may become unused due to branch elimination or constant propagation; but whenever these instructions have not *entirely constant* input operands, they are not removed, as illustrated in Figure 6.3c. Thus, the optimization result may still contain superfluous instructions, which are not relevant for the behavior of the function.

Recombining partially constant expressions (see Figure 6.3b) require a more abstract code representation that allows for more flexible recombination of operands compared to x86-64 instructions.

Finally, deficiencies in the implementation and the limited coverage of x86-64 instructions further limit the applicability of DBrew for more general applications.

6.7.2 Whole-function Binary Rewriting: Drob

Drob [Hil19] is a binary optimizer that performs analyses and transformations at function scope while still focusing on low rewriting times. This rewriter addresses the shortcomings of the strictly-local transformations performed by DBrew by analyzing the entire scope of the optimization target. At the same time, Drob strives to avoid expensive parts of code generation, including instruction selection and register allocation: as these parts are already during done before compile-time, the rewriter attempts to re-use the original code as much as possible, only changing code fragments that actually benefit from additional run-time information.

Publication Information

The information given in this sub-section is based on [Hil19] and the Drob sources.

6.7.2.1 Rewriting Approach

Initially, the function that is specified for optimization and all known targets of function calls contained therein are decoded. With this information, a so-called Interprocedural Control Flow Graph (ICFG) is constructed, which represents the entire rewriting scope and the relation between functions contained in it. All further analyses and optimizations, however, operate at function scope; inter-procedural optimizations, like inlining, are not implemented. In the following, the initial lifting

step, the analyses, and the transformations are described in their main execution order.

Function Representation A function is decoded into superblocks, which have a single entry point, but can have several exits. This is intended to allow for a code representation close to the original machine code layout and reduces the overall number of blocks. To simplify code generation, superblocks can be *chained* to indicate a preferred ordering and enable the omission of a terminating branch instruction by *falling through* to the next block. The superblocks of a function represent its CFG.

Within a superblock, instructions are stored in a target-dependent format, closely following the instruction and register representation of the underlying architecture. Instructions can have a *predicate* for conditional execution, which is used on x86-64 for conditional jumps and moves. Drob also supports handling *unmodeled* instructions, for which no semantics have been specified: such an instruction is conservatively assumed to read and write all registers and modify arbitrary memory locations. Consequently, in this rather conservative rewriting mode, the optimization scope is severely limited.

Register Liveness Analysis For several optimization passes, information about the usage of written registers is helpful, allowing Drob to detect unused effects of instructions, which can therefore be ignored, to identify entirely unused instruction sequences, which can be removed, and to find cyclic chains of unused values.

Starting at the return instructions, information about used registers is propagated backwards through the instructions of all superblocks of the function until the liveness data no longer changes. For each instruction, Drob stores the set of registers which is potentially used afterwards. When further information about conditional execution or eventually written registers is available (e.g., from a previous analysis run propagating constant values), such information is incorporated as well.

Stack Analysis The core analysis of Drob propagates information about the state of registers and stack slots — hence referred to as *stack analysis* — through the program, making use of dynamically available information about specified constants. However, the stack analysis is aborted whenever a (partial) pointer to stack memory escapes the analysis, for example when a pointer to a local variable is written to non-stack memory or when encountering an unmodeled instruction.

The analysis attaches a *program state* to each superblock, which shadows the state of the CPU registers and the stack frame, both in byte-wise granularity. Each shadowed byte has one of the following states:

- *dead:* the value is undefined.

- *unknown:* the value depends on state unknown during the analysis, e.g., unknown parameters or non-constant memory regions.
- *immediate:* the value is a known constant.
- *usrptr:* the value is a constant offset from a pointer parameter, used to associate information with parameters, e.g., alias information or `const`-ness.
- *returnptr:* the return address on the stack.
- *stackptr:* the value is a constant offset to the stack pointer; used to track the address of the stack pointer.
- *tainted:* usage of the value for memory access invalidates stack analysis, e.g., pointers that might point to the stack or are computed from stack pointer fragments.
- *tail/stackptrtail:* as pointers have a size of 8 bytes, the seven bytes following a *usrptr*/*returnptr*/*stackptr* are marked with this state. Access to such values have the state *unknown* or *tainted*.

The analysis starts at the entry block with an initial program state constructed from the function parameters. Within a block, the program state is updated with the effect of the instructions, eventually propagating constant values and emulating instructions with constant input operands. At this point, information about constant values is also attached to the instruction operands for later specialization. At the ends of the superblock, the program state is propagated to the succeeding blocks, which are analyzed as well.

It may happen that a block is analyzed multiple times, for example within loops. In such cases, the program states have to be merged, where different states are updated to a generalized state. For example, merging two different *immediate* values result in *unknown* and merging a *stackptr* and a non-*stackptr* value results in *tainted*. When the program state of the block changes, it has to be examined again with the new, more general state. This process repeats until the analysis data converged to a stable point for all reachable blocks.

Optimizations After the initial analyses, a pre-defined sequence of optimization passes (see below) is executed. Before each optimization pass, Drob checks whether the analysis data still reflects the current state of the code, running the analysis passes again as needed. Consequently, the two analysis passes are executed several times throughout the optimization step. A strategy for partially updating the analysis results with only changed code parts is not implemented as it is expected to be less precise, potentially causing Drob to miss optimization opportunities.

Code Generation After applying the optimization passes, Drob generates new machine code using a two-stage strategy: in the first stage, the code size of all superblocks is computed, using the longest possible encoding for branch instructions. This information is used to select the shortest possible encoding for branches. Chained blocks are laid out in order to avoid branch instructions. Only in the second stage actual machine code is generated. During code generation, the encoder tries to re-use the original encoding if the instruction was unmodified during the optimization process.

As computed constants for floating-point values or vector registers cannot be encoded directly in the instruction stream, they are stored in a separate constant pool located after the generated code.

6.7.2.2 Implemented Optimizations

Drob implements several optimization passes, which are applied in a predefined order. The first optimization pass performs a *simple unrolling of loops* that consist of a single superblock. If a superblock contains a conditional branch to itself, the part of the block up to the conditional branch is duplicated ten times[3] with the conditional branch being copied. If the condition of that branch turns out to be constant, the branch will be removed during specialization. Examples of the loop unrolling pass can be seen in Figures 6.4a, 6.4b, and 6.4d.

The *dead code elimination* pass is based on the stack analysis and removes unreachable superblocks and instructions whose result is never used. In particular, unnecessarily unrolled loop iterations are removed. Similarly, the *dead register write elimination* pass removes unused instructions, but only requires a register liveness analysis, thereby avoiding the creation of an updated stack analysis. The effect of these passes can be easily identified in Figure 6.4c.

When information about constant operands is available, instructions can eventually be *specialized* to make use of this information. If the output values of an instruction are known, these can typically be folded into move instructions, as shown in Figure 6.4c. Constant input operands are folded into immediate operands or references to the constant pool where possible. Note that instructions that become unused in the course of specialization get removed by a later run of the dead code elimination pass.

A special case is the *optimization of memory operands*: if the address is known, it may point to a memory region that was specified as constant. In this case, the memory access can either be removed entirely, for example as in Figure 6.4b, or the constant value gets moved to the constant pool, improving cache usage by exploiting spatial locality. However, even if the memory access does not result in a known value, it is still possible to make use of constant address fragments. Several

[3]The *unroll count* is a configuration option with the default value 10.

```
1  sum_array: // rdi=len, rsi=ptr
2    xor eax, eax // accumulator
3    xor ecx, ecx // loop ctr.
4    jmp .Lloopcond
5  .Lloopbody:
6    add eax, [rsi + 4 * rcx]
7    add rcx, 1
8  .Lloopcond:
9    cmp rcx, rdi
10   jb .Lloopbody
11   ret
```

$\xrightarrow{\texttt{rdi=3}}$

```
1  sum_array_3:
2    xor eax, eax
3    add eax, [rsi]
4    add eax, [rsi+0x4]
5    add eax, [rsi+0x8]
6    ret
```

(a) *Specialization of a function summing an integer array for a constant length 3. The loop is unrolled ten times, but iterations that are never executed are removed again. The value of the then constant loop counter is propagated and folded into the address operand. The addition with zero is not removed as this optimization is only implemented for 64-bit additions.*

```
1  acc_array: // rdi=len, rsi=ptr
2             // rdx=accumulator
3    mov rax, rdx // accumulator
4    xor ecx, ecx // loop ctr.
5    jmp 2f
6  1:add rax, [rsi + 8 * rcx]
7    add rcx, 1
8  2:cmp rcx, rdi
9    jb 1b
10   ret
```

$\xrightarrow{\texttt{rdi=3}\ \texttt{rsi=\&}\{4,5,6\}}$

```
1  acc_array_456:
2    mov rax, rdx
3    add rax, 4
4    add rax, 5
5    add rax, 6
6    ret
```

(b) *Specialization of a function adding up the values of a constant integer array and an integer parameter. The loop is unrolled completely and the memory accesses to the constant values are removed. Expressions are not re-ordered, resulting in a series of addition operations, which could be folded to* ***`add eax, 15`****.*

```
1  foo: // edi, esi, edx are int
2    mov eax,esi
3    imul eax,edx // comp. edx*esi
4    cmp edi,0    // but if edi>=0
5    jl 1f
6    mov eax,edi  // ..return edi
7  1:ret
```

$\xrightarrow{\texttt{edi=6}}$

```
1  foo_spec:
2    mov rax, 6
3    ret
```

(c) *Branches with a known constant condition are removed together with dead code and instructions that do not affect the program behavior. The constant result is written with a newly created* ***`mov`*** *instruction in the destination register.*

```
1  compzero: // edi is int
2    mov eax, edi
3  1:shr eax, 1
4    jnz 1b // ret if eax=0
5    ret
```

$\xrightarrow{\textit{no spec.}}$

```
1  compzero_drob:
2    mov eax, edi
3    shr eax, 0x1
4    jz 2f
5    shr eax, 0x1
6    jz 2f // 8 more times
7  1:shr eax, 0x1
8    jnz 1b
9  2:ret
```

(d) *Rewriting of a function that always returns zero with no further information for specialization. Drob unrolls all simple loops ten times, even if no useful information is available.*

Figure 6.4: *Examples of specializations generated using Drob.*

architectures support more complex addressing modes, for example x86-64 with a base, a scaled index and a constant displacement [Int20a]. Thus, a constant scaled index can also be folded into the displacement, as exemplified in Figure 6.4a.

Finally, Drob reduces the number of branch instructions between superblocks by chaining them, resulting in a *fall-through* edge in the machine code, or merging them, if possible. This is implemented in the *block layout optimization* pass, which is executed after passes that modify the CFG of the function.

6.7.2.3 Limitations and Performance Considerations

Drob combines a function-level analysis with local transformations, applying rather simple transformations in a safe but effective way. Using function-level analyses, information about the usage of constant values is extracted and resulting unused instructions, laying the ground for optimizations to leverage this information to eliminate computational instructions and dead branches.

Code is modified only if an actual optimization is taking place, while all other parts are left unchanged, allowing to safely apply Drob on code having only few spots with optimization potential. As the original machine code instructions are never abstracted away, the original code can be re-used in places without modifications, reducing the overall time needed for machine code generation.

The extensive analyses, in particular the stack analysis, however, are rather expensive, especially as they have to be performed multiple times during the rewriting process. Due to their iterative nature, the analysis time may grow super-linear with the complexity of the control flow.

While Drob provides a code representation and analysis facilities to *enable* even complex transformations, potential of performance improvements is currently limited by the set of *available* optimizations. Especially the lack of more generalized loop transformations prevents significant performance improvements of non-trivial code, where loop bodies typically involve a non-linear control flow, e.g., for conditionals.

One of the biggest drawback is the code representation in machine code instructions, which not only imposes constraints for transformations to always produce encodable instructions, but also increases the difficulty of folding arithmetic expressions, modifying the register allocation, or re-combining instructions.

Ultimately, also the very limited coverage of fully described instruction semantics and specialization opportunities limits applicability on a larger set of applications; and further expansion of the instruction coverage requires non-trivial code modifications.

6.7.3 LLVM-based Binary Rewriting: DBLL

Instead of re-implementing specific optimization on machine code level, a different approach consists of re-using analyses and transformations implemented in compilers.

DBLL[4] [ES20b] makes use of standard compiler optimizations provided by LLVM for the case of binary optimization, transforming binary code at the level of an architecture-independent LLVM-IR (see Section 2.4.2 for a brief description).

Publication Information

This approach was previously published as [ES20b], which in turn is based on work published in [EW17].

6.7.3.1 Rewriting Approach

To make use of the LLVM optimizer and code generator, the first step is to lift the machine code semantics of the function designated for rewriting to the LLVM-IR. This function is then optimized. If during optimization further code is discovered, for example the target of an indirect function call, this code is lifted as well and the optimization process is repeated until no further code is discovered. Finally, the LLVM MCJIT [LLV20c] compiler is used to compile the code and prepare it for execution.

Initial Lifting The initial step of lifting the compiled function to LLVM-IR is done using Rellume (see Chapter 3) with the call-return optimization described in Section 3.3.7.1. This results in a single LLVM-IR function describing the machine code semantics, which takes a pointer to the CPU state as single parameter. The function returns whenever the original function returns; nested function calls and indirect jumps are represented as placeholder calls.

As LLVM does not permit direct access to the stack, at the beginning of the function a fixed-size virtual stack is allocated on the stack using the LLVM `alloca` instruction. The size of the stack defaults to 4 kiB, but can be configured using the *rewriter-specific* configuration API.

Next, the function is adjusted so that the function signature in LLVM-IR matches the parameter and return types specified using the BinOpt API. Inside the function, the CPU state is allocated on the stack at the beginning and parameters are stored in the corresponding virtual registers or virtual stack according to the System V ABI [Mat+14]. If a parameter is marked as constant, the constant value is stored instead, causing the actual parameter to be ignored. Further, every return statement is modified to return the correct values from the CPU state.

[4]Historical note: the approach to use LLVM for specialization was first described in [EW17] as "LLVM fixation". The work was later split into the stand-alone lifting library Rellume [ES20a] (see Chapter 3) and the rewriting approach presented together with the BinOpt library [ES20b], which is named DBLL and described here. DBrew-LLVM [EW17] refers to the approach of post-processing machine code generated by DBrew [WB16; EW17] using LLVM; this direct bridge was removed due to increasing differences in storing decoded x86-64 instructions.

Iterative Code Discovery and Optimization During the lifting process of Rellume, only code within a function is considered. Code reached via `call` instructions (direct and indirect) and code reached via indirect jump instructions is not lifted. Especially resolving indirect call or jump targets is an important run-time optimization, allowing inlining and further constant propagation. However, such targets usually only become known *during* the optimization process. This leads to an iterative procedure of lifting and optimization, which ends when no further code for lifting can be discovered.

Therefore, the overall optimization process is the following: after the initial lifting, some light optimization passes are applied. If any of the placeholder calls for (indirect) calls/jumps is found to have a constant address, code from these addresses if lifted, the placeholder call is replaced with a call to the newly lifted function, or, in case of indirect jumps, the following code fragment is inlined, and the optimization process repeats. Otherwise, the full optimization pipeline is applied. If at this point any placeholder calls have a known address, code is lifted and the process repeats. Otherwise, there is no code left to discover, and all lifted code is fully optimized.

A special case for lifting functions are stubs in the Procedure Linkage Table (PLT): these contain a single indirect jump to a library function, whose address is stored in the Global Offset Table (GOT), or to the dynamic linker if the function is resolved for the first time. However, the GOT is usually mapped as writable to permit lazy binding of libraries, implying that the library function generally cannot be resolved during optimization, even if the address is *probably* known. To avoid this problem and the problem of avoiding to accidentially lift the dynamic linker itself, PLT stubs are detected and not lifted.

Moreover, optimizing into library functions is generally problematic from the performance perspective: many functions like `malloc` or `fopen` provide very little optimization potential based on constant data, but are usually complex and therefore require a large amount of time for lifting, "optimization" and code generation. Other frequently called functions, like `memset` or `strlen`, usually employ highly-tuned implementations making use of assembly language, where performance gains are rather unlikely if the code is re-compiled using LLVM.

Code Generation After optimization, the LLVM-IR module is compiled to machine code using the LLVM JIT compiler. The newly compiled function resides in the same address space and can be used as drop-in replacement, because the previous LLVM-IR function has the same signature as the original function.

However, care has to be taken when some indirect calls or jumps remain after optimization. This case is handled by restoring all registers, including the stack pointer, to their virtual state and continuing at the original code. This is achieved using a combination of inline assembly in LLVM-IR and (ab-)using the `iretq` instruction to set the instruction pointer and the stack pointer [ES20b].

```
1  sum_array: // rdi=len, rsi=ptr
2    xor eax, eax // accumulator
3    xor ecx, ecx // loop ctr.
4    jmp .Lloopcond
5  .Lloopbody:
6    add eax, [rsi + 4 * rcx]
7    add rcx, 1
8  .Lloopcond:
9    cmp rcx, rdi
10   jb .Lloopbody
11   ret
```

$\xrightarrow{\texttt{rdi}=3}$

```
1  sum_array_3:
2    mov eax, [rsi+0x4]
3    add eax, [rsi]
4    add eax, [rsi+0x8]
5    ret
```

(a) *Specialization of a function adding up the values an integer array for a constant length 3. The loop is unrolled, the initial addition with zero is replaced with a load from the* second *element.*

```
1  acc_array: // rdi=len, rsi=ptr
2             // rdx=accumulator
3    mov rax, rdx // accumulator
4    xor ecx, ecx // loop ctr.
5    jmp .Lloopcond
6  .Lloopbody:
7    add rax, [rsi + 8 * rcx]
8    add rcx, 1
9  .Lloopcond:
10   cmp rcx, rdi
11   jb .Lloopbody
12   ret
```

$\xrightarrow{\substack{\texttt{rdi}=3\\ \texttt{rsi}=\&\{4,5,6\}}}$

```
1  acc_array_456:
2    lea rax, [rdx+0xf]
3    ret
```

(b) *Specialization of a function adding up the values of a constant integer array and a third (dynamic) integer parameter. The loop is unrolled and all memory accesses are removed. The constant values are combined and optimized into a x86-64 complex memory operand.*

```
1  div: // edi, esi are int
2    mov eax, edi
3    xor edx, edx
4    div esi
5    ret
```

$\xrightarrow{\texttt{edi}=5}$

```
1  div5:
2    mov ecx, edi
3    mov eax, 0xcccccccd
4    imul rax, rcx
5    shr rax, 0x22
6    ret
```

(c) *Division operations with a constant divisor are lowered to a more efficient combination of a multiplication with a logical shift.*

```
1  compzero: // edi is int
2    push rbx
3    mov eax, edi
4  .Lloop:
5    shr eax, 1
6    jnz .Lloop // ret if eax=0
7    pop rbx
8    ret
```

$\xrightarrow{\textit{no spec.}}$

```
1  compzero_dbll:
2    xor eax, eax
3    ret
```

(d) *The LLVM optimizer detects that the function will always return zero and the unnecessary register spill to the stack is eliminated.*

Figure 6.5: *Examples of specializations generated using DBLL.*

6.7.3.2 Performed Optimizations

Due to the use of the LLVM framework, the huge range of optimizations in standard compilers is available. As a consequence of the high abstraction over the underlying machine code, analyses and transformations are not only greatly simplified, but also some optimizations found in the previously described rewriting approaches, for example moving constants into immediate operands, are not explicitly necessary, because such modifications are performed automatically during the lowering process. In general, as all information about the register allocation, instruction selection, and block ordering is not preserved during lifting, such low-level characteristics of the machine code are implicitly improved. Figure 6.5c shows an example of this, where a division with constant divisor is transparently lowered to a multiplication.

For the main optimization, the generic `-O3` pass pipeline is used, extended with only three new passes specifically tailored to machine code rewriting. The generic pass sequence provided by LLVM includes passes for instruction combination, inlining, vectorization, loop unrolling (see, e.g., Figures 6.5a and 6.5b) and other (non-trivial) loop transformations, and propagating memory accesses to SSA registers where safely possible (see, e.g., Figure 6.5d). The following passes are specifically designed for DBLL.

Improved Alias Analysis When the lifted machine code makes heavy use of integer-pointer conversions, the basic analysis of potential memory aliasing has to be conservative and prevents transformations regarding loads/stores. In combination with the stack-allocated CPU state, however, this needlessly limits important optimizations: the original program has no access to that structure and therefore it is known to not alias with any other pointers used by the original code.

This information is supplied to the alias analysis infrastructure using a newly added plug-in, which indicates that any pointer not derived from the CPU state can never alias with a pointer derived from the CPU state. This pass ensures that the CPU state structure will be eliminated by the scalar replacement of aggregates (SROA) pass.

Constant Memory Regions When compiling from higher-level languages, the case of constant addresses (besides `null`) occurs rather rarely, for example, with memory mapped I/O, but in general does not require any special handling. Thus, LLVM simply treats such constant addresses like any other pointer.

In contrast to the compile-time setting, there *is* more information about constant addresses in the run-time setting: load operations from read-only memory regions can be folded into constants. Such information can either be derived from the virtual memory mapping, e.g., `/proc/self/maps` on Linux, or can be explicitly specified during configuration.

To make use of this knowledge, an additional pass is scheduled multiple times in the pipeline. This pass checks for all load instructions with a constant address whether the entire memory access has a known value, and if this is the case, the load is replaced by a constant value. Such constants are propagated during later passes, for example with instruction combination. An example of the application of this pass is shown in Figure 6.5b.

Folding Integer-Pointer Casts As integer and pointer arithmetic is indistinguishable in machine code, but uses separate instruction in the LLVM-IR, the lifted code may contain several occurrences of `inttoptr` instructions. When a pointer parameter is specialized, this frequently leads to sequences of `ptrtoint` followed by arithmetic followed by `inttoptr`. Such sequences are currently not folded with the standard optimization passes, preventing further propagation constant pointers.

This problem is addressed by adding a simple pass which folds such sequences of `ptrtoint/add` to a `getelementptr` instruction, which is dedicated for pointer arithmetic.

6.7.3.3 Limitations and Performance Considerations

The described LLVM-based rewriting approach provides access to a wide range of transformations and analyses used by traditional compilers for the purpose of runtime binary optimization. The abstraction of low-level details with the architecture-independent LLVM-IR having the SSA properties enables complex transformations, which, combined with the highly-tuned code generation infrastructure, allows for heavily optimized rewritten code.

These abstractions and high-level optimizations, naturally, imply a significant cost: several time-expensive analysis passes are required for more involved transformations, and code generation encompasses several complex lowering steps. As information about the original selection of machine instructions or registers is discarded during lifting, these steps have to be done again, even if specific code parts did not meaningfully changed.

Furthermore, while the optimizing code generation back-end (SelectionDAG) generally performs a high-quality selection of instructions, it is also known to have a significant performance impact, especially due to its own internal IR. A faster replacement back-end named GlobalISel is under active development (initially with focus on AArch64), but is not ready for wider use. [LLV21]

A different limitation regarding transformation opportunities arises from a lack of meta-information. During compilation from source, many meta-information on the generated machine code is available, for example, information about the stack frame layout, array bounds, data types, and calling conventions. Most of this information is usually lost during compilation and also not stored in the debug information. The lack of such information can limit transformations, e.g., reorganizing the stack

frame or optimizing calls to unknown functions. For example, unspilling variables from the stack is generally impossible, unless it can be proven that no other code is able to access that location. Consequently, there may still be discrepancies between the run-time optimized code and a code specialized at compile-time, even if the same underlying optimizations are applied.

6.8 Evaluation of Approaches

To evaluate the applicability, performance, and effectiveness of the three previously described rewriting systems, these systems will be applied on several micro-benchmarks. The benchmarks were selected to cover different subsets of optimizable patters and operations. In addition to evaluating the achieved performance gain and the time needed for the optimization itself, this evaluation also seeks to discover practical limitations.

6.8.1 Setup

All experiments described in the following were performed on a system equipped with an Intel Xeon Bronze 3106 CPU (Skylake), clocked at 1.7 GHz with disabled TurboBoost, 11 MiB L3 cache, and 80 GB main memory, running Ubuntu 18.04.3 with Linux kernel 4.15.0-70 in 64-bit mode. All code was compiled with GCC 9.2.0 with the `-Ofast` option and linked against glibc 2.27. For DBrew, commit `10a202f`[5] was used; for Drob commit `90570d4`[6] was used; and for BinOpt, which includes DBLL in its source tree, commit `cf7a9a6`[7] was used together with LLVM 11.0.1. Due to lack of meaningful support of instructions from newer x86-64 ISA extensions, all compilers and code generators were restricted to only use instructions from the base x86-64 instruction, including SSE/SSE2, but without newer extensions.

For each micro-benchmark, the three rewriting systems DBrew, Drob, and DBLL are applied to generate a specialized and optimized version of the kernel function. The observed execution times are compared against the execution of the original, unoptimized function, which will serve as baseline for the run-time optimizations. Moreover, the performance is also compared against a specialization obtained at compile-time using the inlining and constant-propagation optimizations, which therefore can be considered as an optimal specialization.

To ensure stable time measurements, even for short time spans, the rewritten kernel of the micro-benchmark are repeated sufficiently often to achieve time spans larger than a second; and all measurements are repeated five times, and in case of high variance, more runs are performed. In this thesis only the average times

[5] `https://github.com/caps-tum/dbrew/tree/10a202f`
[6] `https://github.com/aengelke/drob/tree/90570d4`
[7] `https://github.com/aengelke/binopt/tree/cf7a9a6`

are shown; the depiction of (always rather short) error bars was omitted for better readability.

6.8.2 Micro-Benchmarks & Results

In this section, the individual benchmarks, the specialization configuration when applying the rewriting systems, and the performance results of the newly generated code will be described. The results of the execution times of the optimized kernels are shown in Figure 6.6a.

6.8.2.1 Stencil

Benchmark The problem of applying a stencil unknown during compilation to a matrix is a frequent problem, for which reason a resulting benchmark was previously used for evaluating DBrew [WB16; EW17], Drob [Hil19], and DBLL [ES20b], but also for other dynamic optimization techniques [Gra+99; DeV+13]. To avoid unnecessarily frequent indirect function calls and allow for the use of vector units, the kernel function used here performs the two-dimensional stencil operation on all elements of a given matrix with a run-time-specified stencil. The matrix elements are single-precision floating-point values. The stencil is specified as a sparse data structure, being an array of triples consisting of an x-offset, a y-offset, and a multiplication factor.

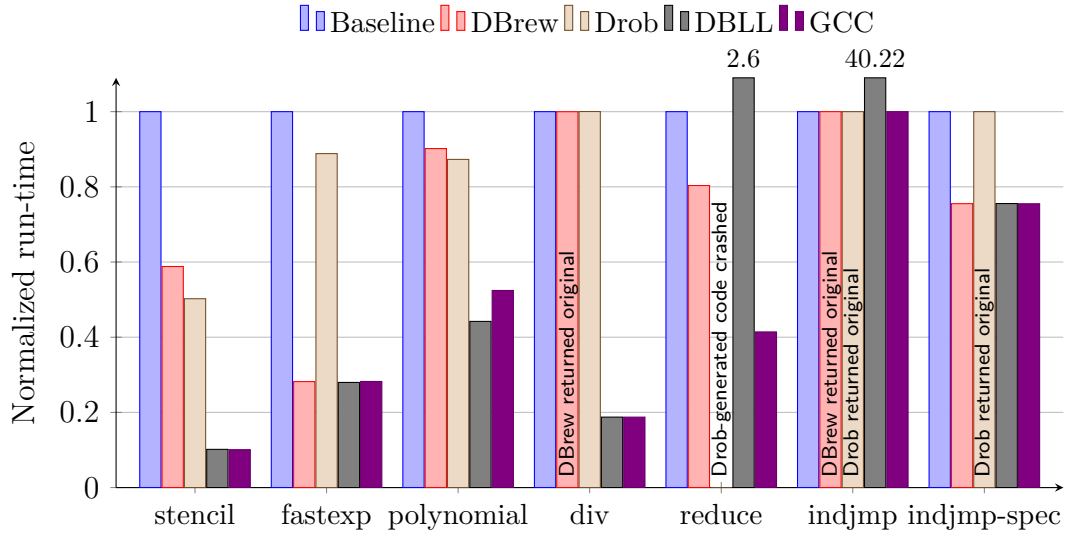
Specialization The stencil is specialized for a four-point stencil, computing the average of all four immediately neighbored matrix elements. Further, the size of the matrix is also marked as constant and fixed as 649×649. Additionally, floating-point optimizations were allowed to allow regrouping of computations, e.g. by factoring out common multipliers.

To work around the aggressive unrolling behavior of DBrew, for DBrew a slightly modified kernel had to be used[8]: the inner specialization target that applies the kernel to a single element had to be spliced out into a different non-inlinable function. This way, for the outer function the *force-unknown* mode could be applied, preventing the loop indices from being marked as constant.[9]

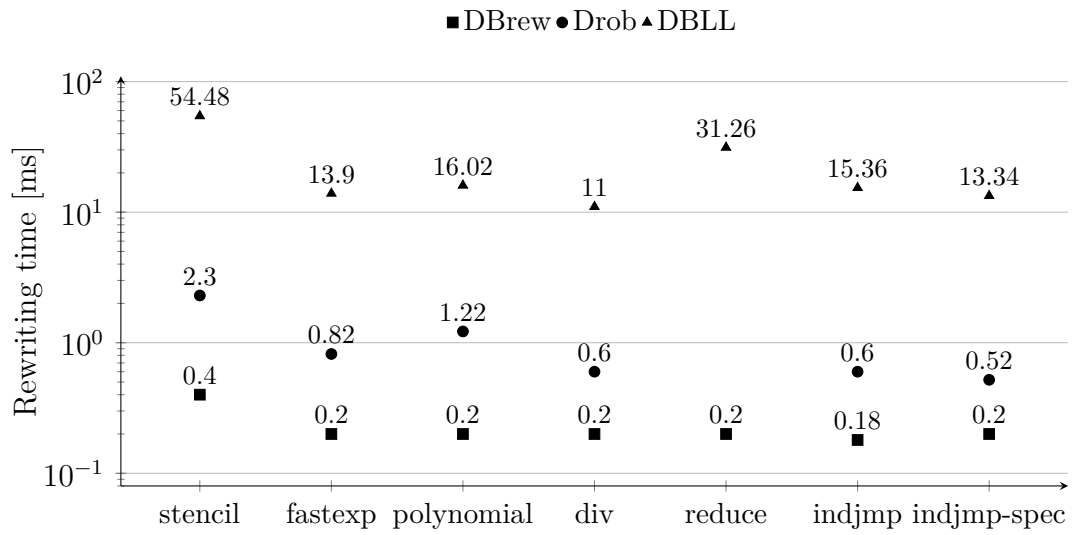
Results DBrew achieves a performance improvement of 42% by unrolling the loop iterating over the stencil points and propagating constant offsets from memory into the instruction stream. The code specialized by Drob is slighly faster with 50%

[8] Otherwise, DBrew failed due to buffer size limitations.

[9] Not specifying the matrix size as constant value led to *infinite unrolling*, as the static loop counter caused state duplication, which never ended as the iteration bound is dynamic. Increasing the buffer sizes (or, alternatively, reducing the matrix size) caused the rewriting process to succeed, but the generated code was non-functional, as actually needed instructions were removed.



(a) *Run times normalized to the unoptimized code, excluding rewriting times, comparing the code generated by the three binary optimizers with an implementation specialized and optimized at compile-time.*



(b) *Rewriting times. Note that the Y axis is shown in logarithmic scale.*

Figure 6.6: *Performance results comparing the three described approaches for run-time binary optimization on micro-benchmarks.*

overall improvement by additionally removing dead instructions and folding constants deeper into existing instructions, e.g. as immediate operands or displacements in memory operands. The performance improvement of DBLL is 90% and thus at the same level as the compile-time specialized kernel, performing vectorization and moving the common multiplication factor into a single multiplication operation.

6.8.2.2 Fastexp

Benchmark In this benchmark, the kernel function performs the square-and-multiply algorithm for exponentiation with an integer exponent on a floating-point base. This benchmark was chosen to evaluate the support for optimizing loops with a non-trivial control flow in the loop body. This benchmark is inspired from an example of using ‘C [Pol99].

Specialization The integer exponent is specialized to the constant 76, which leads to a sufficiently complex operation (in contrast to choosing a power of two). With the constant exponent, all control flow decisions and other integer arithmetic become obsolete, leaving only the essential floating-point computations.

To work around a limitation of DBrew when applying calling conventions, the function parameters had to be ordered in way such that constant parameters are sorted before floating-point parameters.

Results Both, DBrew and DBLL, reach the performance of the compile-time specialized code. Drob, however, is unable to effectively optimize the function due to a missing pass for unrolling loops consisting of more than one superblock.

6.8.2.3 Polynomial

Benchmark The kernel function of this benchmark evaluates a polynomial, which is given as an array of coefficients, at a specified point. For example, to evaluate the polynomial $2x^3 + 5x - 3$, the coefficients are given as $[-3, 5, 0, 2]$. For all values, double-precision floating-point arithmetic is used. With this benchmark, optimization of floating-point arithmetic is evaluated.

Specialization For specialization, the coefficients are fixed to a constant array of length 11, which leads to a sufficiently complex operation, containing several entries with values zero and one. Possible optimizations are loop unrolling, consequent elimination of integer arithmetic instructions, removal of terms with a coefficient value zero and omission of multiplications with a constant value one.

Results DBLL achieves a performance improvement of 56% over the generic kernel. In fact, the LLVM-generated code is even slightly faster than the code generated by GCC, due to a more efficient mapping of the arithmetic operations to machine code instructions. Due to a lack of arithmetic optimization, DBrew and Drob yield a specialized version, which improves performance by only 10% and 13%, respectively, over the generic variant.

6.8.2.4 Division

Benchmark This benchmark performs an integer division. It was originally selected to evaluate optimizations of complex integer operations. As DBrew and Drob currently do not model the semantics of the division instruction, this benchmark actually shows handling of unsupported instructions for these rewriting systems.

Specialization The function is specialized for the constant divisor 11, which is non-trivial to optimize as it is not a power of two. This allows to replace the rather expensive integer division operation with a much more efficient multiplication with a constant *magic number* combined with a shift [War13].

Results DBrew aborts rewriting and returns the original function as the integer division instructions is not supported. Drob also does not support the division instruction, but enters a more restrictive rewriting mode and therefore still succeeds in rewriting without performing any actual modifications. As optimizing division operations with a constant divisor is a common compiler optimization, this is also implemented by LLVM and therefore the LLVM-based rewriter produces the same code as GCC.

6.8.2.5 Reduce

Benchmark In this benchmark, a function is sequentially applied on all elements of an array of double-precision floating-point values, with the current array element and the result of the previous function call (or the first element in case of the first call) as arguments. Such an operation is commonly referred to as *reduction* and can be used, for example, to compute the minimum or sum of the array. This benchmark was created to evaluate the capability of inlining an indirect function call.

Specialization The kernel function is specialized with reduction operation which compute the sum of the two specified parameters. The resulting optimization possibility consists of inlining the specified function and removing instructions required for maintaining ABI compliance.

Results DBrew is the only rewriting system that successfully optimizes this benchmark and inlines the function, yielding an improved function with a performance improvement of 20%. However, note that no ABI was specified for the passed function, implying that the performed optimization was actually unsafe. In particular, modern compilers may choose to modify ABI of strictly internal functions, which may cause unexpected problems when rewriting arbitrary code.

The code generated by Drob is non-functional, as Drob stops decoding when encountering the indirect function call and therefore ignores the function epilogue, ignoring the fact that the indirectly called function will return.

DBLL fails to inline the function and replaced the indirect function call with its own strategy to continue at original code, causing a slowdown of factor 2.6x. As no ABI information about the specified function is present and the indirect function call is performed in a loop, the LLVM optimizer cannot prove that the callee-saved register holding the address of the function is not modified by the function itself. Thus, the indirect function call is maintained for correctness.

6.8.2.6 Indjmp & Indjmp-Spec

Benchmark These benchmarks are the most simplest used for this evaluation: the kernel function takes a function pointer as single parameter. The specified function is called and its result is returned. As a consequence of tail-call optimization, the resulting machine code consists of a single indirect jump instruction. This benchmark was designed to evaluate handling of indirect jumps during the rewriting process.

Specialization This benchmark was evaluated in two configurations: in the first specialization configuration, no additional information was specified, implying that the indirect jump cannot be removed. In the second configuration, a function pointer was specified as constant parameter, in which case the indirect jump can be removed and the specified function can be inlined.

Results Neither DBrew nor Drob can handle indirect jumps with an unknown address and aborted their rewriting process by return the original function. DBLL is able to rewrite the first configuration, but the resulting code was 40x slower as a consequence of using the expensive and serializing `iretq` instruction.

In the second configuration with a constant jump target, DBrew and DBLL successfully inline the called function; Drob, in contrast, is unable to handle indirect jump, even when the target becomes dynamically known.

Table 6.2: *Relation between optimization effect, rewriting time, and the number of kernel calls needed to amortize the rewriting time. The base time is the execution time of the unoptimized kernel. All execution times refer to one execution of the rewritten function and do not include rewriting times.*

	Base time	Exec. time	Diff.	Rew. time	#calls
Stencil/DBrew	3.03ms	1.78ms	−1.25ms	0.40ms	1
Stencil/Drob	3.03ms	1.52ms	−1.51ms	2.30ms	2
Stencil/DBLL	3.03ms	0.31ms	−2.73ms	54.48ms	20
Fastexp/DBrew	15.18μs	4.28μs	−10.90μs	200μs	19
Fastexp/Drob	15.18μs	13.48μs	−1.69μs	820μs	486
Fastexp/DBLL	15.18μs	4.24μs	−10.93μs	13900μs	1272
Polynomial/DBrew	13.84μs	12.48μs	−1.35μs	200μs	149
Polynomial/Drob	13.84μs	12.08μs	−1.75μs	1220μs	698
Polynomial/DBLL	13.84μs	6.11μs	−7.72μs	16020μs	2076

6.8.3 Rewriting Times

The measured times spent for performing the actual rewriting and optimization is shown in Figure 6.6b. Two main causes for varying transformation times can be observed: the first factor is the complexity of the rewritten code itself. From the described micro-benchmarks, the *stencil* benchmark is the most complex with three nested loops. This complexity immediately results in an increase in rewriting time by a factor of 2–4 compared to benchmarks with a single loop or no loop at all.

The second, and much more important, factor is the complexity of the performed analyses and transformations itself. DBrew performs only very simply and strictly local operations and therefore is also the fastest approach on all benchmarks (unless DBrew performs massive loop unrolling, which is not the case on any of the benchmarks shown here). Drob requires 3–7 times more rewriting time compared to DBrew due to its whole-function analyses. DBLL, in contrast, takes 13–26 times more rewriting time than Drob and 55–135 times more time than DBrew. This is not only caused by the more extensive analyses performed at LLVM-IR-level, but also the optimized machine code generation using the SelectionDAG back-end is more complex and therefore requires more time.

Table 6.2 compares the rewriting time with the absolute performance improvement of the performed optimization. It can be observed that in all shown cases the faster rewriting approach requires a lower number of execution times to amortize the cost of the optimization process; in case of the *stencil* benchmark, DBrew's optimization already pays off with a single execution of the function. When the transformed code is executed more often, the use of the heavy rewriter DBLL becomes beneficial.

6.8.4 Discussion

The experiments and results above clearly show differences between the quality and applicability of the rewriting systems. These differences are caused by a different quality of the applied optimizations, but also the rewriting time has an impact.

Optimization quality DBrew has a simple yet effective optimization strategy, which can achieve significant performance improvements by rather simple optimizations. Still, the architectural limitation of strictly-local optimizations structurally prevents advanced transformations like vectorization. Moreover, also implementation issues become apparent, most notably from missing instructions and other bugs — for example, even for rather simple kernels as in the *stencil* benchmark, several workarounds had to be applied.

Drob performs further analyses on the entire function and therefore can perform further optimizations. However, there are currently few optimizations implemented that make use of these capabilities, so that only small improvements over DBrew's local approach can be achieved. Further, the lack of a general loop unrolling transformation, which can handle loops consisting of more than one superblock, strongly limits optimization effects on larger optimization targets. Finally, also Drob only covers a small subset of x86-64 instructions. As Drob enters a restrictive mode when encountering such instructions, no further optimizations are performed afterwards. While this allows Drob to technically succeed "optimizing", in such cases it actually does not optimize code but rather copies the existing code without modifications.

DBLL is the only rewriter which is able to rewrite all evaluated micro-benchmarks. In most cases it is further able to achieve a performance-level similar to a compile-time specialized code, mostly as a consequence of re-using compiler techniques. This indicates that lifting machine code to well-optimizable LLVM-IR code is generally possible. Nonetheless, whenever non-resolvable indirect calls or jumps are involved, the "optimization" was not beneficial and in fact resulted in a significant slowdown. Due to lack of knowledge about ABI semantics of function calls and to ensure strict correctness of transformations, indirect functions in loops cannot be resolved — solving this problem would require either more ABI information to be specified or some kind of speculative optimization to determine whether some transformation *will turn out* to be valid. Although DBLL is the only rewriter to support unresolvable indirect jumps, the use of an interrupt return instruction for handling this case in the rewritten code adds considerable overhead. Thus, all kinds of indirect calls or jumps are currently best avoided for all three rewriting systems.

Transformation time The rewriting time has an immediate effect on the applicability of the run-time optimization: a fast rewriter like DBrew can also be used effectively for optimizations where the optimized function has only a short overall

run-time or is used rarely, despite the simplicity of the actually performed optimizations. In contrast, an expensive optimizer like DBLL usually generates more efficient code, but due to high transformation times, it also needs a sufficiently large timespan to amortize the costs.

This motivates the design of *hybrid* rewriters, which perform execution and optimization in parallel on different processor cores with multiple optimizers. Such a system could initially start with a fast-generated but not thoroughly optimized specialization and replace the the rewritten kernel with more optimized specializations once they become available from another rewriter running on a different core. Further research in this direction is left as future work.

Conclusions The previous results and observations show that DBLL is currently the most robust rewriting system with the widest support of instructions and optimization possibilities. While both DBrew and Drob implement approaches to also optimize transformation time and therefore allow for a faster pay-off, implementational issues such as the small set of supported instructions (DBrew and Drob), missing optimization passes (Drob), or other bugs (DBrew) prevent the general applicability.

6.9 Evaluation on Real-world Code

In the previous section, the performance and applicability of the different rewriting systems was evaluated using micro-benchmarks. To also evaluate the overall approach of a library for application-guided run-time binary optimization, the library is applied to performance-sensitive real-world code. In particular, in this section, image processing kernels will be specialized, which generally have a large number of configuration options controlling various parts of the algorithm and therefore make them hard to optimize statically at compile-time.

> ***Publication Information***
>
> *The application of run-time binary optimization on the benchmarks described in the following was previously published in [ES20b], which was developed as part of this work. The evaluation presented here is more detailed and covers more configuration options and comparisons.*

6.9.1 Benchmarks

For this evaluation, two operations from the image processing library Generic Graphics Library (GEGL) [Kol20] are optimized by applying the library BinOpt. The general procedure for adapting the operations is the following: first, the main operation kernel and relevant configuration options are identified. In particular, from

the entire process routine, non-critical set-up code is separated from performance-critical code which actually operates on the image data. Second, this operation is outlined into a separate function with the relevant configuration options and other important run-time information as function parameters, for example, the dimensions of the image. Third, the library calls for creating a BinOpt configuration and specialization of the function previously outlined are added and the specialized function is used instead. Within the configuration, parameters and associated memory regions with known run-time values are marked as constant. Finally, a new configuration option to disable the run-time optimization is added for the purpose of comparison — if disabled, the outlined function is called directly.

Gblur-1d Filter The first optimized operation is the *gblur-1d* filter. This operations implements a one-dimensional Gaussian blur filter with a configurable standard deviation and orientation (horizontal or vertical). A two-dimensional Gaussian blur filter, which is usually used for two-dimensional images, is realized as two applications of the one-dimensional filter with different orientations. For this evaluation, only the Finite Impulse Response (FIR) filter is optimized as this filter provides better numerical stability over the Infinite Impulse Response (IIR) filter. The operation supports images with 1–4 color channels and uses single-precision floating-point values to represent a value of a color channel.

Internally, the operations works as follows (see also Figure 6.7). Initially, a one-dimensional weight matrix is computed. The size of the matrix depends on the configured standard deviation σ and the resulting number of rows/columns is $6.5 \cdot \sigma$, rounded upwards to the next odd integer number to enable symmetric application on a pixel. After this setup step, the actual operation begins. The individual rows or columns of the image — depending on the orientation of the filter — are copied into a temporary buffer with a user-configurable policy for handling corners of the image. Then, the weight matrix is applied to each pixel and each color channel separately and the results of the computation are written back into a different buffer. Once all columns or rows are finished, the new result is copied back into the image. As a consequence of the copying step, the actual filter operation does not depend on the orientation. However, as the image is has a row-major order in memory, using a vertical orientation leads to more scattered memory access and therefore uses processor caches less efficiently.

To apply BinOpt for specialization, the innermost part, which applies the matrix to a given point, is outlined into a separate function for specialization. This function is then specialized for the size of the matrix, the matrix values itself, and the number of color channels.

Bilateral Filter The second optimized operation is the *bilateral filter*, which is a two-dimensional blur filter. While it is conceptually similar to the Gaussian blur

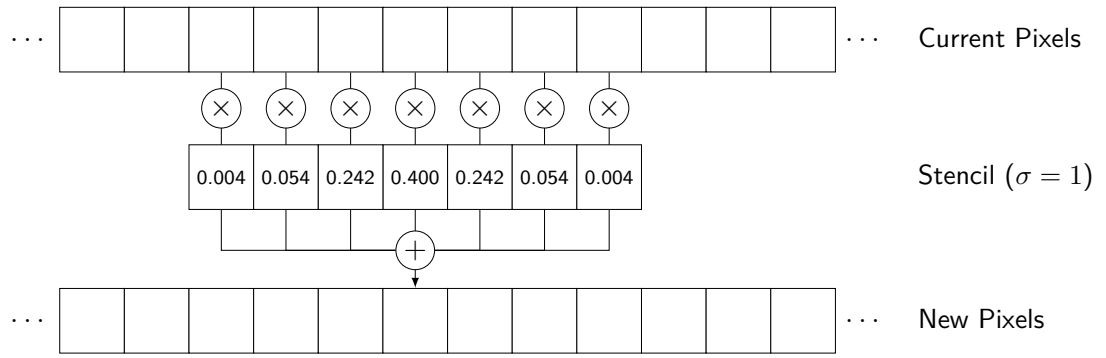


Figure 6.7: *Illustration of the Gblur-1d filter with $\sigma = 1$. The one-dimensional stencil covers 7 pixels, whose values are weighted, summed up, and stored as new pixel value. This process is applied for all pixels of the image.*

filter, the impact of neighbored pixels not only depends on their distance, but also on their per-channel color difference. Nearby pixels with a larger color difference therefore have less impact than pixels with a similar color. The operation in GEGL only supports a single color format, which is the 4-channel RGBA format with single-precision floating-point data types.

Initially, the operation computes the two-dimensional distance-weight matrix, whose size depends on the user-configured blur radius. For a specified radius r, the weight matrix has the size $(2 \cdot r + 1) \times (2 \cdot r + 1)$. Then, in the main part of the operation, all pixels of the image are processed. For each pixel, all neighbored pixels in the specified radius are considered; their values are accumulated weighted by the distance-weight from the matrix computed previously and the inverse color difference; and the result is written to the new image. To avoid accessing the pixel values of pixels beyond the border of the source images, additional bounds checks are required.

To apply BinOpt for specialization, the main part that performs the actual operation after the computation of the weight matrix is outlined to a separate function for specialization. This function is then specialized for the blur radius, which indicates the matrix size, the matrix values themselves, as well as the dimensions of the image. In the original kernel, the computing the color distance involves the use of the exponential function from the standard library. To avoid indirect function calls, this call was replaced by a polynomial approximation in both, the specialized kernel and the reference baseline.

6.9.2 Setup

The setup of the hardware and software stack is identical to the setup described in Section 6.8.1. Additionally, GEGL 0.4.28 and Babl 0.1.84 are used. For image

verification, ImageMagick 6.9.7.4 was used. However, as the compiled kernels use several instructions that are currently unsupported by DBrew and Drob, only the DBLL rewriter is used for this evaluation. All codes are compiled with the `-Ofast` compiler option, which is also used by GEGL, and for the run-time optimized code, fast-math optimizations are enabled.

The images used for this evaluation are dynamically generated using GEGL's checkerboard render operation, cropped to a specified size. Note that for the operations considered here, the values of the individual pixels have no impact on the actually performed operations.

The kernels are applied on different configurations of the standard deviation or blur radius. To determine the required workload size where the rewriting time is amortized, additionally images of different sizes are used. As the *gblur-1d* filter supports images with a different number of color channels, for that benchmark the image is explicitly re-formatted to a specified image format, and the different color formats grayscale (Y), grayscale with transparency (YA), RGB color (RGB), and RGB color with transparency (RGBA) are evaluated as well.

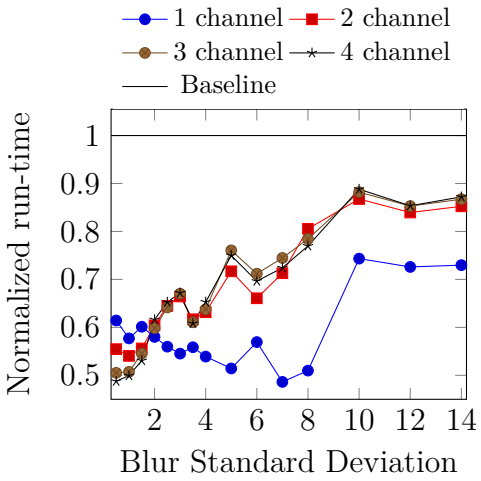
To evaluate the effectiveness of run-time optimizations, of the *bilateral filter* benchmark also source-code-level specializations are generated, which incorporate the same information as specified to the run-time optimization library. This code is compiled with GCC 9.2.0 and also with Clang 11, which uses the same LLVM framework as the run-time optimizer.

All produced output files are verified for correctness using the `compare` tool of ImageMagick. It is verified that the resulting images are either identical to the baseline without any optimizations or have a low mean absolute error (i.e., at most a few pixels diverge by insignificant values, which could result from differences in floating-point rounding).

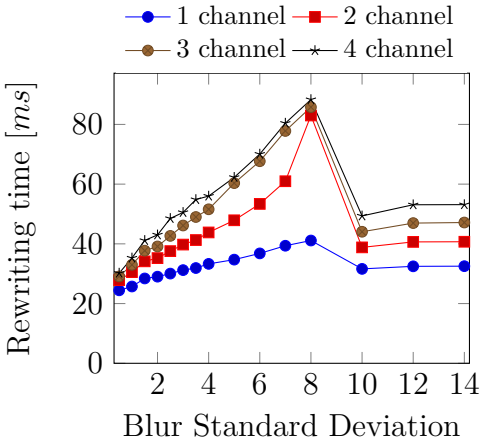
6.9.3 Results

Gblur-1d Filter The results of the evaluation on the *gblur-1d* filter are shown in Figure 6.8. Overall, the application of the described LLVM-based binary optimization technique leads to performance improvements of 51%, as shown in Figure 6.8a. For a reasonably small blur radius and images with multiple color channels, the performance improvement is in the range of 25–40%. For filters with a standard deviation less than 10 (corresponding matrix size: 65 elements), the loop that iterates over the neighbored points is unrolled, eliminating the loop overhead and allowing for more optimized use of vector instructions. If the matrix is larger, unrolling is no longer considered as beneficial by the heuristics built into LLVM. Further optimizations include unrolling the loop over the color channels and elimination of associated offset computations.

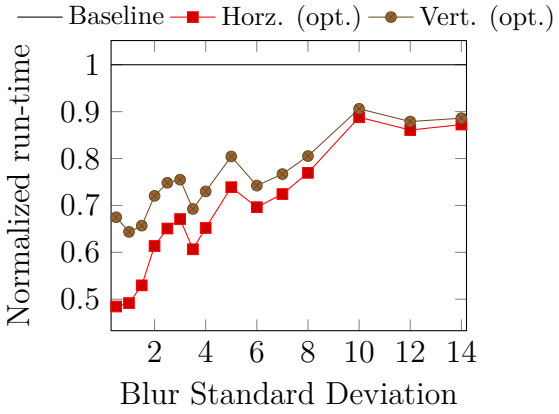
A significant difference in the performance improvement originates in the number of color channels: images with a single color channel can use SIMD extensions much



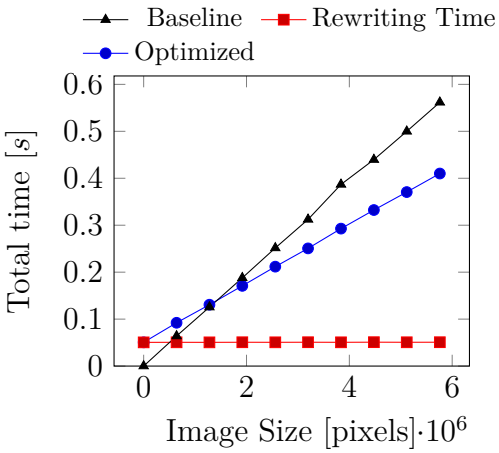
(a) *Execution time of the dynamically optimized kernel normalized to the unoptimized kernel, with images having different numbers of color channels. When only one color channel is present, de-interleaving of data is not necessary, resulting in smaller code more efficient usage of vector processing. Image size: 5000×3000, horizontal orientation.*



(b) *Rewriting time for the dynamically optimized kernel normalized to the unoptimized kernel, with images having different numbers of color channels. Optimization times correspond with the code size, which grows with the stencil size and is smaller for single-channel operation as no de-interleaving of data is required.*

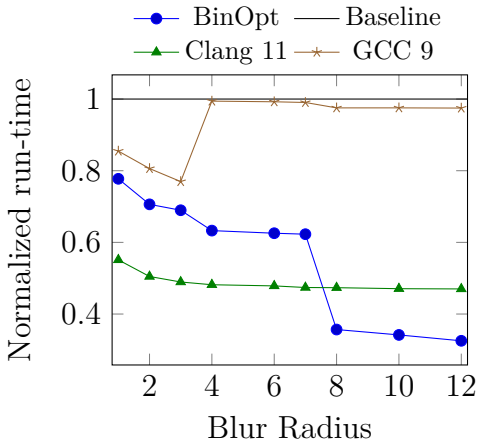


(c) *Execution time of the dynamically optimized kernel normalized to the unoptimized kernel, applied with horizontal or vertical application. The orientation has no impact on the run-time optimized part and thus, absolute performance differences are similar; however, relative performance improvements are smaller due unfavorable memory access patterns. Image size: 5000×3000, 4 color channels.*

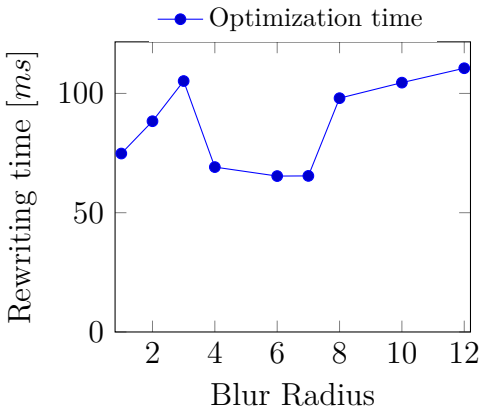


(d) *Overall execution time for different image sizes with blur standard deviation 3. Images have a square size with the specified number of pixels and four color channels. The run times are approximately equal for images of size* 1200×1200 *pixels.*

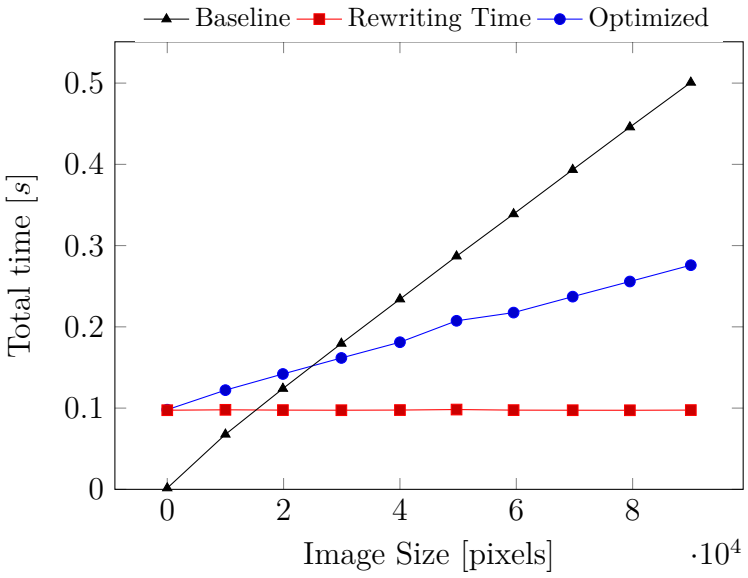
Figure 6.8: *Performance results evaluating the dynamically optimized GEGL gblur-1d kernel. Execution times include rewriting times, the baseline is the kernel with any dynamic transformations disabled.*



(a) *Overall execution time of the dynamically optimized kernel and compile-time specialized kernels normalized to the unoptimized kernel. Performance differences are caused by different unrolling and vectorization strategies. Image size: 5000×3000.*



(b) *Rewriting time for the dynamically optimized kernel normalized to the unoptimized kernel. The rewriting time is higher when loop unrolling is performed.*



(c) *Overall execution time for different image sizes with blur radius 8. Images are have a square size with the specified number of pixels. The run times are approximately equal for images of size* 180×180 *pixels.*

Figure 6.9: *Performance results evaluating the dynamically optimized GEGL bilateral-filter kernel. Execution times include rewriting times, the baseline is the kernel with any dynamic transformations disabled. The kernel internally only supports images with four color channels.*

more effectively, because neighbored pixel values are stored immediately next to each other and therefore no de-interleaving of color channels is required. When multiple color channels are present, a series of move and shuffle instructions is required.

The rewriting time, shown in Figure 6.8b, initially grows linear with the size of the matrix, but once loop unrolling is no longer performed, the rewriting time drops significantly due to the smaller size of the code. The number of color channels also has an impact on the rewriting times; the most significant difference is in case of a single color channel rewriting is faster as shuffling values is not required.

As expected, the orientation of the filter has no impact on the absolute performance improvement (cf. Figure 6.8c). However, the relative performance improvement compared to the unoptimized version is smaller for vertical kernels, as more time is spent for copying the columns of the image into the temporary buffer.

Due to the rather small performance improvement of 25–40% and rewriting times up to 90ms, larger image sizes are needed to amortize rewriting times. As shown in Figure 6.8d, with a standard deviation of 3 (corresponding matrix size: 21) and four color channels, a size of $\sim 1200 \times 1200$ ($\sim 1.4M$ pixels) is required to achieve a faster overall performance when run-time optimizations are performed.

Bilateral Filter The performance results for the *bilateral filter* are shown in Figure 6.9. Applying BinOpt/DBLL on this kernel leads to performance improvements of up to 67%, as shown in Figure 6.9a. For a blur radius smaller than 8 pixels, the improvement is slightly lower with around 30–40%. These performance improvements have a variety of sources: first, for a small radius (≤ 3), the loops iterating over the weight matrix are unrolled and therefore the loop overhead is eliminated. Additionally, the bounds checks and index computations can be simplified. However, no vectorization is performed. Loop unrolling is no longer considered as beneficial by the LLVM heuristic for a radius larger than 4, but vectorization is still not performed. Only with a blur radius of 8 pixels and larger, the LLVM vectorizer considers use of SIMD processing as beneficial: now, always four neighbored pixels are processed in parallel, with the four color channels distributed over separate vectors. Additionally, the matrix loop is now unrolled again.

When specifying the same information as constants for GCC to generate an operation for a fixed blur radius and image size, only loop unrolling is performed for small matrices, but besides this, no vectorization or other optimizations take place. Consequently, GCC is unable to leverage the additional information even when explicitly specified. In contrast, Clang pursues a different vectorization strategy and always puts the four color channels of a pixel into a vector. While this approach is faster compared to code without vectorization, at least for larger matrices a vectorization schema that parallizes over pixels turns out to be more effective. It should be noted that both Clang and BinOpt/DBLL are based on the same LLVM version for optimization.

The rewriting time, as shown in Figure 6.9b, roughly corresponds to the size of the generated code. When loop unrolling is performed (blur radius < 4 or ≥ 8), the rewriting time grows due to increased effort needed for analyses and code generation.

Due to the large performance improvement, the high rewriting time in the range of 60–100ms amortizes already for small images. As it can be seen in Figure 6.9c, with a blur radius of 8 pixels, a size of $\sim 180 \times 180$ ($\sim 32k$ pixels) is already sufficient to achieve a faster overall performance.

6.9.4 Discussion

The results show that, even for optimized real-world code, significant performance improvements over statically optimized code can be achieved. The observed speed-up was only possible due to exploitation of concrete values from dynamic configuration options. Although it is possible to generate specialized kernels for all possible configuration values, such an approach is highly impractical compared to the relatively minor changes necessary to use the BinOpt API.

When comparing with compile-time specialized code with the same specialization data, the run-time optimized version derived from prior machine code could even outperform two state-of-the-art C compilers. This is an indicator that starting optimization from machine code is no large optimization hindrance. Especially, the fact that GCC is unable to leverage the additional information for specialization is a clear indication that basing a dynamic binary optimization system on LLVM has more potential for effective optimizations. However, the results also imply that both compiler infrastructures, GCC and LLVM, still have substantial problems in finding a good vectorization strategy.

The time needed for the optimization process itself strongly depends on the performed optimization and the complexity of the code. Still, the high rewriting time for the LLVM-based binary rewriting approach can be amortized already with a single execution of the optimized kernel. Further improvements in this regard would be possible by re-using specialized kernels for several kernel executions, for example, when processing multiple images with the same operation sequence.

Nonetheless, the technique is not in all cases beneficial — especially for small workloads — and therefore the time overhead has to be balanced with the benefits. Moreover, it does require programmer effort to identify suitable optimization targets and related code modifications to separate this code to have function boundaries, so that a function-level specialization can be applied. Thus, the approach of application-guided run-time binary optimization can only be reasonably used for performance-critical code sections when optimization potential can be derived from run-time data in a sufficiently compact and isolatable code region.

6.10 Related Work: Post-compilation Optimization

The idea of using run-time-only information for generating a further optimized program during its execution has a long history. Approaches for realizing this optimization step can be classified into three categories:

- Libraries that simplify generation of machine code at run-time;
- Language constructs, language extensions, or Domain-specific Languages (DSLs) created for designating code regions for run-time compilation or optimization; and
- Dynamic binary optimizers, which improve performance during program execution without explicit knowledge or control of the application itself and in many cases automatically without any external input.

In all these categories, research gained significant traction in the mid-1990s to the early-2000s. Since then, computer architecture and compiler development have seen radical changes in development: not only did Dennard scaling [Den+74] end, forcing CPU manufacturers to increase the number of processor cores and do other per-core optimizations to improve the overall performance of processors [Esm+11], but also compilers have seen major improvements with regard to optimization, most notably with profile-guided optimization [Fre01], which is widely available nowadays, and improved strategies for instruction selection [Bli16].

It can be assumed, that for these reasons the performance impact of many dynamic optimization techniques became lower, thereby causing overall research in these areas to decline. While in all categories notable improvements have been published in recent years, only library-based approaches have received significant developments and wider adoption over the past decades.

Classification of BinOpt The approach of application-guided run-time binary optimization described previously in this chapter can be classified as a hybrid approach of a library, the use of language constructs, and dynamic binary optimization. From a purely technical perspective, it is a library, which a program can link against to use provided functionality. From the perspective of a user, this is not only a library, it also allows to re-use the same programming language and language constructs as the rest of the code. Internally, the library is a dynamic binary optimizer, which — in contrast to many other systems — has explicit knowledge about run-time data which it is worth optimizing for.

Structure The rest of this section on related work is structured as follows: in the next three sub-sections, important and recent approaches in each of the three categories will be described. Finally, Section 6.10.4 will cover approaches regarding feedback-directed optimization and static binary optimization.

6.10.1 Run-time Code Generation Libraries

Numerous libraries and frameworks that implement or supplement JIT compilation or run-time code generation in general and work without compiler modifications have been proposed. Thus, in the following, only a small selection of relevant or highly related libraries will be covered.

LLVM [LA04] is a widely used run-time code generation and optimization infrastructure, which is not only used in static compilers, like Clang [LLV20a], but also for JIT code compilation. The LLVM-IR code can be constructed using API calls from the application. Afterwards, optimizations or other transformations can be applied and, finally, the JIT infrastructure can be used to compile the created code into the same address space.

libgccjit [Mal20] is a library that exposes the optimization and code generation part of GCC as library for JIT compilation. Internally, this library uses GCC to create a temporary shared object file, which is then compiled using standard tools and afterwards loaded into the process using the dynamic linker (`dlopen()`). In contrast to LLVM, libgccjit requires an assembler and a linker to be available on the execution machine.

QBE [Car] is a generic code generation library, which aims to be more flexible and provide faster code generation by avoiding expensive transformations. Generally, the approach attempts to balance overall complexity of the compilation process with the performance of the emitted machine code. MIR [Mak20] is a more recent library with similar ideas, aiming to act as JIT compiler for CRuby and replacing currently used LLVM and libgccjit.

KART [Noa+17] is a library that provides the infrastructure for run-time compilation of C/C++/Fortran source files. Specializations can be specified using preprocessor defines given as command-line parameters to the compiler. The code is compiled into a shared library and loaded into the same program using the dynamic linker (`dlopen()`).

6.10.2 Run-time Optimization with Compiler Support

Several approaches for exposing dynamic code generation through specific constructs integrated into an existing programming language, through compiler extensions, or through the use of Domain-specific Languages (DSLs), have been implemented.

The only widely available “language construct” can be found in dynamic languages like Python [Pyt21b] or ECMAScript [Ecm21] (commonly known as JavaScript): the `eval()` function. Such a function takes a dynamically constructed string as parameter and evaluates it using the semantics of the programming language itself. As the provided expression can also describe a function, `eval()` can be used to convert dynamically generated functions into natively callable functions. When the execution engine of the dynamic language supports JIT compilation, for example like

V8 [Tit15] for JavaScript, this dynamically evaluated function is turned into machine code. For example, CPython uses this technique to create efficient `namedtuple` implementations [Pyt21a].

Extensions of the C/C++ ‘C [EHK96; Pol+99] (Tick C) extends the C language [ISO17] with additional syntax to enable application-controlled run-time optimizations. In particular, they add operators to mark code regions to be compiled at run-time and incorporate data at run-time into dynamically generated code as well as type specifiers to simplify handling of dynamically generated functions. During compilation, code regions marked for run-time compilation are compiled to “code-generating functions”, which at run-time create the actual machine code incorporating dynamic values.

Tempo [Noe+98; CN96] is an approach for run-time specialization using partial evaluation based on the C language, which performs further analyses of side effects and is implemented by generating small templates during compilation, which are linked together at run-time.

DyC [Gra+99; Gra+00] is a different C extension for supporting dynamic optimizations, where constant data is marked only using a special function call. During compilation, a set of code templates and an associated dynamic code generator are produced. During execution, the code generator fills in static values and concatenates the templates to the final function.

Easy::Jit [CG18] is based on modified C++ compiler (Clang) and allows to dynamically optimize functions. During compilation, all LLVM-IR code is embedded into the binary so that it is accessible at run-time. When the JIT-compilation is triggered using a call into the run-time library, the LLVM-IR code is optimized for the actual parameters, compiled and executed. This project is closest to the BinOpt approach described in this chapter. However, in contrast to requiring a specific compiler and C++ as language, the BinOpt approach works with arbitrary compilers as long as the function follows the C ABI. Further, BinOpt allows to explicitly specify additional memory regions as constant or parameters as dynamic, exposing more optimization potential while allowing for better re-use of already created specializations.

ClangJIT [Fin+19] is the most recent approach based on C++ attributes and the Clang/LLVM infrastructure. For annotated functions, the abstract syntax tree produced by the compiler and the generated LLVM-IR code are embedded into the executable file. At run-time such functions are then optimized for the specified parameters. In addition to CPUs, also optimization of CUDA code is supported.

OpenCL [Khr19] code is generally compiled at run-time for a chosen host architectures, usually CPUs or Graphics Processing Units (GPUs). During compilation of a module at run-time, additional specialization constraints can be specified using API calls, which can be used by the back-end for further optimizations.

Other languages LLVM [LA04] IR code cannot only be generated using API calls, but can also be pre-generated from higher-level languages using existing compilers and then be dynamically loaded, specialized, and compiled at run-time.

Terra [DeV+13] is a low-level language designed for producing specialized implementations embedded in Lua code. Terra code can be produced dynamically and is JIT-compiled to machine code using LLVM for execution.

DeGoal [Cha+14] is an approach to produce fast code generators from a simple assembly-like language. At run-time, constants and other data is filled inserted at specified places and machine code is generated.

6.10.3 Dynamic Binary Optimization

Dynamic binary optimizers strive to improve performance of a currently running program by modifying its machine code. Such systems usually operate transparently to the program that is optimized. Many of these systems operate automatically without further guidance and therefore have no prior knowledge about the application, for which reason they have to detect frequently executed ("hot") code fragments where improvements are possible during the program execution.

One of the first transparent binary optimizers was Dynamo [BDB99; BDB00], which optimizes code on the PA-RISC architecture. The system initially runs the program in an interpreter, where it identifies hot traces of actually executed code by software profiling and optimizes such traces in native code by combining included code blocks, improving locality. DynamoRIO [BGA03] started as a variant of Dynamo targeting the transparent optimization of x86 code, but afterwards shifted its focus towards generic code modification [Bru04].

Mojo [Che+00] is a similar transparent optimization system for x86 with support for multi-threading, but in fact increases execution times on several benchmarks.

Wiggins/Redstone [DGR99] do transparent binary optimizations for Alpha, where hot code fragments are re-organized into traces and specialized for concrete run-time values; the system uses hardware performance counters to identify relevant hot code fragments. Adore [Lu+04] uses a similar technique for Itanium and additionally inserts prefetch instructions to reduce memory latency. COBRA [KHY07] also targets Itanium and further extends these ideas by using separate threads to monitor performance counters and optimizing memory accesses by inserted prefetch instructions and hints for concurrent memory access, reducing cache misses in multi-threaded programs.

Kistler and Franz [KF01] propose a dynamic optimization system for PowerPC which permanently profiles the application, performs data layout and trace optimizations, and generates a new binary, which replaces the loaded executable in memory during execution. However, they do not optimize machine code directly, but instead start from an architecture-independent low-level code representation (Slim Binary [FK97]).

More recent dynamic binary optimizers are more specialized to a specific type of optimization. Selftrans [NMO11] performs run-time vectorization of existing binary code by analyzing the machine code for specific patterns that are found to benefit from vectorization and replacing occurences with an improved implementation that makes better use of available SIMD extensions.

Padrone [Rio+14] is a framework which allows to profile and optimize machine code running in a different process. Although the framework does not implement any optimizations itself, it has been used to implement an optimization tool which increases the length of vector operations from 128-bit SSE to 256-bit AVX instructions [Hal+15].

ExanaDBT [SYE17] is an optimization system that lifts profiles code during execution and performs polyhedral optimizations on identified code fragments. Selected code fragments are lifted to LLVM-IR using McSema [Tra21], where Polly [GGL12] is applied to perform the actual transformation.

These approaches differ from the BinOpt approach, which not only avoids the overhead of profiling and transparently dispatching to optimized code, but also allows for highly targeted transformations and better exploitation of available information.

6.10.4 Static and Feedback-directed Optimization

Besides approaches to exploit run-time information directly during execution, a different strategy consists of a three stages, which work without code modifications: in the first stage, the program is initially compiled, optionally with profiling instrumentation. In the second stage, this program is then executed on different, common workloads and profiling information about actual values or control flow decisions is gathered. In the third stage, the program is the re-compiled or otherwise optimized based on this profiling information.

As this strategy only allows for optimizing on *likely* values instead of *guaranteed* values, the optimized program still has to operate properly for all sets of possible inputs. Thus, such optimizations can be considered as complimentary to run-time specializations.

Link-time Optimization Originally, link-time optimizers operated on machine code emitted in the prior compilation step. For example, PLTO [Sch+01] is a static binary rewriter that optimizes compiler-produced object files during linking, eventually including profile data from a previous compilation with instrumentation.

However, with the integration of link-time optimization into compilers [GCC05; LA04], which relies on embedding the compiler intermediate code representation in the object file and using these for enhanced inter-procedural optimizations during linking, link-time optimizers operating on machine code became obsolete.

Profile-guided Optimization Classic profile-guided optimization is widely available with standard compilers [Fre01]. Code is first compiled with instrumentation for profiling, then executed on some workloads, and finally compiled again where the previously acquired profiling data is employed to guide optimizations.

AutoFDO [CML16] is a system which instead profiles applications running in production using system-level profilers to account for changing workloads. The gathered data is then used for steering optimization when the next build is released for production use.

BOLT [Pan+19] is a static binary optimizer which uses collected profiling data to perform optimizations on binary level, allowing more precise use of gathered data.

Auto-tuning Frameworks Tools for automatic performance tuning are generally faced by a variety of tuning parameters, of which they strive to find a performance-effective combination. Therefore, typically a subset of possible combinations is evaluated, from which a result is derived. Such systems can tune for selections of compiler options [PE06], but also for application-specific parameters [TCH02].

The type of performed optimizations is orthogonal to the approach described here, as such frameworks do not specialize for actual run-time data. Still, the BinOpt approach can be used to optimize auto-tuning frameworks by avoiding expensive compilations and propagating application-specific tuning parameters further inside the application, as outlined previously in Section 6.3.

6.11 Summary

In this chapter, an application-guided technique for run-time binary optimization was presented. In particular, a new library *BinOpt* was introduced to expose such optimizations in a general, flexible, and future-proof manner. For optimization, an application can designate functions and explicitly mark further information to generate specialized versions of a function. Three different approaches and the corresponding implementations were described, which operate at different abstraction levels and therefore have different optimization possibilities and optimization overhead. The results show that an LLVM-based rewriting approach achieves best performance, with an overall performance improvements of up to 67% on real-world applications. In future, a hybrid optimization system which selects the optimization strategy dynamically can reduce rewriting overheads and a better integration into other languages, for example by using meta-programming capabilities of C++, can further ease usage.

7 Conclusions

Efficient and performant execution of program code requires a transformation to native machine code, which is typically achieved by compiling code from programming language. This separate compilation step ties compiled code to a specific processor architecture and prevents subsequent optimization based on information that is only available while the program is running. Dynamic binary rewriting is a widely used technique to address these challenges and also enables in-depth analyses of performance limitations and potential security vulnerabilities. Many of these systems, however, lack high-quality optimizations typically found in compilers, causing a significant slowdown.

Therefore, this thesis introduced Rellume, a performance-focused library to bridge the gap between dynamic binary rewriting systems and the standard compiler framework LLVM by lifting machine code to the IR of LLVM, enabling full use of the optimization and code generation infrastructure. To reduce the performance impact in the context of dynamic compilation, particular emphasis was put on reducing the overhead of the lifting step itself and on optimizing the newly generated code.

Based on Rellume, the Instrew framework for LLVM-based Dynamic Binary Translation and Dynamic Binary Instrumentation was described. The framework implements a novel client–server approach, where the entire translation and instrumentation procedure is split into a separate process to increase flexibility and enable improvements of efficiency in distributed systems. The presented instrumentation API allows for flexible code modifications in the architecture-independent LLVM-IR. Performance results on the SPEC CPU2017 benchmark suite [Sta21] show that Instrew has an average overhead of only 59% over the native execution, which is significantly lower than comparable systems like QEMU [Bel05] with 1044% and Valgrind [NS07b] with 547%.

To enable dynamic optimizations based on additional application-specific information, I proposed a library-based approach implemented in the BinOpt library, where an application can explicitly designate compiled functions for specialization on dynamic configuration data. Further, an API unifying the interfaces of different dynamic optimizers was introduced, providing a way to target multiple rewriting systems in a common, yet flexible manner. For this API, three different rewriting approaches were described, which operate at different instruction levels. The performance results show, that on optimized image processing kernels, an LLVM-based rewriting approach yields a reduction of the overall execution time by up to 67%.

In combination, this work enables the effective use of LLVM for all applications of

dynamic binary rewriting and demonstrated that by providing a high-performance framework for Dynamic Binary Translation and Dynamic Binary Instrumentation as well as a library for application-guided run-time binary optimization.

Future Work

Extending Rellume to support lifting more architectures, for example, complete support of AArch64 or the Power architectures, would increase the portability of the lifter and make the all depending systems described in this thesis usable for more systems. Also, porting popular tools from other instrumentation frameworks to Instrew would enable to fully use the potential of LLVM's optimizations for widely used instrumentation use cases and at the same time lead to improvements to the Instrew tool API, making tool development easier. Moreover, Instrew could also be applied for security purposes, for example, by combining static and dynamic binary analysis to gain insights into possibly unwanted behavior with lower effort.

In regard to the work on dynamic binary optimization, further guidance for application developers on optimization possibilities by tools integrated into static compilers would allow for a more effective use of dynamic optimizations. Such tools could point out missed optimizations due to missing knowledge during the initial compilation. Also, integrating more semantic information about machine code in a portable manner could further reduce the optimization overhead and has the potential to simplify the API for applications even further.

For both, Instrew and BinOpt, further performance optimizations can be achieved by avoiding redundant analyses and code generation. This could be realized by a system daemon, which caches or even speculatively transforms code and can also integrate easily into the Instrew approach with the client–server architecture. Especially in distributed systems and particularly in High-Performance Computing, where many systems and CPU cores execute the same program, such a daemon could substantially reduce the overhead incurred by the dynamic code generation process. A system-level daemon could also open the door towards transparent collection of performance statistics and optimization without requiring any code modifications or user interaction.

The presented techniques can further be used to improve the performance of tools for program analysis and security based on dynamic binary instrumentation. Further, the dynamic optimization techniques allow to reduce the overhead of abstract high-level programming models, especially for distributed systems, and therefore ease programming while maintaining performance at the same time. This aspect of efficiency is not only relevant for High-Performance Computing, but may also gain importance in the context of cloud computing.

A Developed Software

This appendix provides a compact overview over software that was developed for this thesis and described in previous chapters.

Rellume A library for efficiently lifting machine code to performent LLVM-IR.

- Currently supported architectures: x86-64 and RISC-V.
  Support for AArch64 is work-in-progress.
- Currently supports LLVM versions 9–11.
- `https://github.com/aengelke/rellume`, licensed under LGPLv2.1+.

Instrew A performance-focused framework for LLVM-based Dynamic Binary Translation (DBT) and Dynamic Binary Instrumentation (DBI).

- Currently supported guest architectures: x86-64 and RISC-V.
  Support for guest architectures strongly depends on Rellume.
- Currently supported host architectures: x86-64 and AArch64.
- Currently supports LLVM versions 9–11.
- `https://github.com/aengelke/instrew`, licensed under LGPLv2.1+.

BinOpt A library for application-guided optimization and specialization of compiled functions at run-time, providing a unified API for DBrew, Drob, and DBLL (see below).

- Currently supported architecture: x86-64.
- `https://github.com/aengelke/binopt`, licensed under LGPLv2.1+.

BinOpt – DBLL A rewriter implementation for BinOpt based on Rellume and LLVM.

- Currently supported architecture: x86-64.
- `https://github.com/aengelke/binopt/tree/master/rewriter/dbll`, licensed under LGPLv2.1+.

B BinOpt API Description

This appendix describes the C API of BinOpt. Section 6.6 describes the design rationale and implications for implementations of this API.

```
const char* binopt_driver(void);
```

Return the name of the actual rewriter implementation, or the string `Default (no rewriting)` for the default implementation. This can be used to distinguish rewriters when using rewriter-specific configuration options.

```
typedef struct BinoptOpaqueHandle* BinoptHandle;
BinoptHandle binopt_init(void);
```

Create a new handle into the rewriter. A handle must only be used in a single thread, multiple handles can be used in different threads.

```
void binopt_fini(BinoptHandle);
```

Close a handle and free all associated resources, including optimized functions and specialization configurations.

```
typedef struct BinoptCfg* BinoptCfgRef;
typedef void* BinoptFunc;
BinoptCfgRef binopt_cfg_new(BinoptHandle handle, BinoptFunc base_func);
```

Create a new configuration for a given function. Implementations may deduce type information from DWARF or CTF information encoded in the binary, or other sources. If such information is not available, the type must be configured using `binopt_cfg_type`.

```
BinoptCfgRef binopt_cfg_clone(BinoptCfgRef base_cfg);
```

Clone a configuration. This creates a deep clone of the previous configuration and inherits all properties. The new configuration is entirely independent of the old configuration.

```
void binopt_cfg_free(BinoptCfgRef cfg);
```

Free a configuration.

```
typedef enum {
    BINOPT_TY_VOID = 0,
    BINOPT_TY_INT8,
    BINOPT_TY_INT16,
    BINOPT_TY_INT32,
    BINOPT_TY_INT64,
    BINOPT_TY_UINT8,
    BINOPT_TY_UINT16,
    BINOPT_TY_UINT32,
    BINOPT_TY_UINT64,
    BINOPT_TY_FLOAT,
    BINOPT_TY_DOUBLE,
    BINOPT_TY_PTR,
    BINOPT_TY_PTR_NOALIAS,
} BinoptType;
void binopt_cfg_type(BinoptCfgRef cfg, unsigned count, BinoptType ret, ...);
```

Set function signature with a specified number of parameters. Functions with a variable number of arguments are not supported.

```
typedef enum {
    /// Undefined flag value. Do not use.
    BINOPT_F_UNDEF = 0,
    /// Maximum stack size of optimized code.
    BINOPT_F_STACKSZ,
    /// Fast-math optimizations flags.
    BINOPT_F_FASTMATH,
    /// Log level verbosity. 0 = none/quiet (default).
    BINOPT_F_LOGLEVEL,
    // Other flags can be defined by the implementation.
} BinoptOptFlags;
void binopt_cfg_set(BinoptCfgRef cfg, BinoptOptFlags flag, size_t val);
```

Set a configuration flag. Specific rewriters may offer more configuration flags, before these can be safely used, an application has to identify the rewriter using `binopt_driver`. However, unsupported flags shall be ignored.

```
void binopt_cfg_set_param(BinoptCfgRef cfg, unsigned idx, const void* val);
```

Set a parameter to a constant value. Note that this API takes a non-captured pointer to the constant value. The size of the dereferenced memory is inferred from the parameter type. The value is copied into an internal configuration storage. The parameter index must be smaller than the number of configured parameters.

```
void binopt_cfg_set_parami(BinoptCfgRef cfg, unsigned idx, size_t val) {
  binopt_cfg_set_param(cfg, idx, &val);
}
```

Convenience function specializing an integer parameter to a given constant value.

```
typedef enum {
    /// The memory region flags depend on the page mapping -- read-only pages
    /// are assumed to be constant, while unmapped and writable regions are
    /// treated as dynamic.
    BINOPT_MEM_DEFAULT = 0,
    /// The memory region is treated as constant. Behavior if the area is
    /// modified between configuration and the last call of the rewritten code
    /// is undefined.
    BINOPT_MEM_CONST,
    /// The memory region and all regions deduced from pointers (recursively)
    /// loaded from that are assumed to be constant. Some rewriters do not
    /// support detecting nested pointers and treat such regions as regular
    /// constant memory.
    BINOPT_MEM_NESTED_CONST,
    /// The memory region is treated as dynamic.
    BINOPT_MEM_DYNAMIC,
} BinoptMemFlags;
void binopt_cfg_mem(BinoptCfgRef cfg, void* base, size_t size,
                    BinoptMemFlags flags);
```

Explicitly configure a memory region. Constant memory regions must not be modified until the last execution of a function derived from this configuration.

```
void binopt_cfg_set_paramp(BinoptCfgRef cfg, unsigned idx, const void* ptr,
                           size_t size, BinoptMemFlags flags) {
  binopt_cfg_set_param(cfg, idx, &ptr);
  binopt_cfg_mem(cfg, ptr, size, flags);
}
```

Convenience function specializing a pointer parameter to a memory region and attach information about size and constant-ness to this region.

```
BinoptFunc binopt_spec_create(BinoptCfgRef cfg);
```

Create a specialized implementation for a configuration. The ABI of the returned function is identical to the original function. A rewriter is not required to actually return a specialized implementation, it may also return a pointer to the original function.

```
void binopt_spec_delete(BinoptHandle handle, BinoptFunc spec_func);
```

Mark a specialized implementation for deletion. However, a rewriter does not necessarily have to release the associated storage, as this may be impossible due to code reuse.

Acronyms

ABI Application Binary Interface.

AoT Ahead-of-Time.

API Application Programming Interface.

AVX Advanced Vector Extensions.

BLAS Basic Linear Algebra Subprograms.

CFG Control Flow Graph.

CPU Central Processing Unit.

CSR Compressed Sparse Row.

CTF Compact C Type Format.

CUDA Compute Unified Device Architecture.

DBI Dynamic Binary Instrumentation.

DBT Dynamic Binary Translation.

DSL Domain-specific Language.

ELF Executable and Linkable Format.

FIR Finite Impulse Response.

FPGA Field-programmable Gate Array.

FPU Floating-point Unit.

GCC GNU Compiler Collection.

GEGL Generic Graphics Library.

GOT Global Offset Table.

GPU Graphics Processing Unit.

HHVM Hip-Hop Virtual Machine.

HPC High-Performance Computing.

ICFG Interprocedural Control Flow Graph.

IIR Infinite Impulse Response.

IPC Inter-process Communication.

IR Intermediate Representation.

ISA Instruction Set Architecture.

JIT Just-in-Time.

MMU Memory Management Unit.

MPI Message Passing Interface.

OpenMP Open Multi-Processing.

PLT Procedure Linkage Table.

RISC Reduced Instruction Set Computer.

SIMD Single Instruction Multiple Data.

SSA Single-Static Assignment.

SSE Streaming SIMD Extensions.

SSH Secure Shell.

TCG Tiny Code Generator.

List of Figures

List of Tables

Listings

Bibliography

[14] *FreeBSD Manual Pages: CTF(5)*. Sept. 2014.

[Aba+05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity: Principles, implementations, and applications". In: *Proceedings of the 12th ACM conference on Computer and Communications Security*. 2005, pp. 340–353.

[AC71] Frances E. Allen and John Cocke. *A catalogue of optimizing transformations*. 1971.

[Ang06] Marc Angelone. "Approaches for Universal Static Binary Translation". MA thesis. PA, USA: Drexel University, Mar. 2006.

[App21] Apple, Inc. *About the Rosetta Translation Environment*. `https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment`, accessed 2021-03-30. 2021.

[Arm18] Arm Limited. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. Dec. 2018.

[Arm20] Arm Limited. *Arm Compiler for Linux – Arm Developer*. `https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-compiler-for-linux`, accessed 2021-03-25. 2020.

[Ayc03] John Aycock. "A brief history of just-in-time". In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. "Dynamo: A Transparent Dynamic Optimization System". In: *SIGPLAN conference on Programming language design and implementation (PLDI)*. 2000.

[BDB99] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. *Transparent dynamic optimization: The design and implementation of Dynamo*. Tech. rep. HPL-1999-78. Hewlett-Packard Company, June 1999.

[Bel+17] Fabrice Bellard et al. *QEMU Wiki: Features/KVM*. `https://wiki.qemu.org/Features/KVM`, accessed 2021-03-25. Feb. 2017.

[Bel+19] Fabrice Bellard et al. *QEMU Wiki: Documentation/TCG*. `https://wiki.qemu.org/Documentation/TCG`, accessed 2021-03-25. Nov. 2019.

[Bel+20] Fabrice Bellard et al. *QEMU Source: Tiny Code Generator README*. `https://gitlab.com/qemu-project/qemu/-/blob/07ce0b05/tcg/README`, accessed 2021-03-25. Dec. 2020.

[Bel05] Fabrice Bellard. "QEMU, a fast and portable dynamic translator". In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. "An infrastructure for adaptive dynamic optimization". In: *International Symposium on Code Generation and Optimization (CGO)*. 2003, pp. 265–275.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. "An API for Runtime Code Patching". In: *International Journal of High Performance Computing Applications (IJHPCA)* 14.4 (2000), pp. 317–329.

[Bla+02] L. Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R. Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[Bli16] Gabriel Hjort Blindell. *Instruction selection: principles, methods, and applications*. Springer, 2016.

[BM11] Andrew R. Bernat and Barton P. Miller. "Anywhere, Any-time Binary Instrumentation". In: *SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE)*. 2011, pp. 9–16.

[Bou+17] Ahmed Bougacha et al. *Dagger*. `https://github.com/ahmedbougacha/dagger`, accessed 2021-04-01. Dec. 2017.

[Bru04] Derek L. Bruening. "Efficient, Transparent and Comprehensive Runtime Code Manipulation". PhD thesis. Massachusetts Institute of Technology, 2004.

[BZ11] Derek Bruening and Qin Zhao. "Practical memory checking with Dr. Memory". In: *International Symposium on Code Generation and Optimization*. CGO '11. 2011, pp. 213–223.

[Car] Quentin Carbonneaux. *QBE Compiler Backend*. `https://c9x.me/compile/`, accessed 2021-03-04.

[CC10] Vitaly Chipounov and George Candea. *Dynamically Translating x86 to LLVM using QEMU*. 2010.

[CC19] Emilio G. Cota and Luca P. Carloni. "Cross-ISA machine instrumentation using fast and scalable dynamic binary translation". In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2019, pp. 74–87.

[CG18] Juan Manuel Martinez Caamaño and Serge Guelton. "Easy::Jit: compiler assisted library to enable just-in-time compilation in C++ codes". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. 2018, pp. 49–50.

[CH15] Ray S. Chen and Jeffrey K. Hollingsworth. "ANGEL: A Hierarchical Approach to Multi-Objective Online Auto-Tuning". In: *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'15)*. 2015.

[CH17] Michael Clark and Bruce Hoult. "Rv8: a high performance RISC-V to x86 binary translator". In: *1st Workshop on Computer Architecture Research with RISC-V (CARRV)*. Oct. 2017.

[Cha+14] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A Endo, and Rémy Gauguey. "deGoal: a tool to embed dynamic code generators into applications". In: *International Conference on Compiler Construction (CC)*. 2014, pp. 107–112.

[Che+00] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M Gillies. "Mojo: A dynamic optimization system". In: *ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*. 2000, pp. 81–90.

[Cif+01] Cristina Cifuntes, Mike Van Emmerik, Norman Ramsey, Brown Lewis, et al. *The University of Queensland Binary Translator (UQBT) Framework*. Tech. rep. The University of Queensland and Sun Microsystems, Inc., 2001.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. 2011, pp. 265–278.

[Clo16] Félix Cloutier. *Lifting x86 code into LLVM bitcode*. `http://zneak.github.io/fcd/2016/02/16/lifting-x86-code.html`, accessed 2021-04-01. Feb. 2016.

[CLU02] Cristina Cifuentes, Brian Lewis, and David Ung. "Walkabout – A retargetable dynamic binary translation framework". In: *Workshop on Binary Translation*. 2002.

[CML16] Dehao Chen, Tipp Moseley, and David Xinliang Li. "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications". In: *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2016, pp. 12–23.

[CN96] Charles Consel and François Noël. "A general approach for run-time specialization and its application to C". In: *SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1996, pp. 145–156.

[Cor12] Jonathan Corbet. *Yet another new approach to seccomp*. `https://lwn.net/Articles/475043/`, accessed 2021-06-10. Jan. 2012.

[Cor19] Jonathan Corbet. *Comparing GCC and Clang security features*. `https://lwn.net/Articles/798913/`, accessed 2021-06-10. Sept. 2019.

[Cot+17] E. G. Cota, P. Bonzini, A. Bennée, and L. P. Carloni. "Cross-ISA machine emulation for multicores". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 210–220.

[CV00] Cristina Cifuentes and Mike Van Emmerik. "UQBT: Adaptable binary translation at low cost". In: *Computer* 33.3 (2000), pp. 60–66.

[Cyb18] Cyberhaven. *Translating binaries to LLVM with Revgen*. `http://s2e.systems/docs/Tutorials/Revgen/Revgen.html`, accessed 2021-04-02. 2018.

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[Den+74] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits*. Vol. 9. 5. 1974, pp. 256–268.

[DeV+13] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. "Terra: A Multi-Stage Language for High-Performance Computing". In: *SIGPLAN conference on Programming language design and implementation (PLDI)*. Vol. 48. 6. 2013, pp. 105–116.

[DFA18] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. "rev.ng: A multi-architecture framework for reverse engineering and vulnerability discovery". In: *2018 International Carnahan Conference on Security Technology (ICCST)*. 2018, pp. 1–5.

[DGR99] Dean Deaver, Rick Gorton, and Norm Rubin. "Wiggins/Redstone: An on-line program specializer". Presented at the Hot Chips 11 Conference. 1999.

[Din+11] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. "PQEMU: A parallel system emulator based on QEMU". In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 2011, pp. 276–283.

[Dor+20] Noah Dormann, Simon Kammermeier, Johannes Pfannschmidt, Florian Schmidt, and Alexis Engelke. *RIA-JIT: Dynamic Binary Translation (RISC-V → x86)*. `https://github.com/ria-jit/ria-jit`, accessed 2021-03-30. Oct. 2020.

[DWA17] DWARF Debugging Information Committee. *DWARF Debugging Information Format Version 5*. Feb. 2017.

[Ecm21] Ecma International. *ECMAScript 2020 Language Specification*. 2021.

[EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. "'C: A language for high-level, efficient, and machine-independent dynamic code generation". In: *SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1996, pp. 131–144.

[EOS21] Alexis Engelke, Dominik Okwieka, and Martin Schulz. "Efficient LLVM-Based Dynamic Binary Translation". In: *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '21. 2021, pp. 165–171.

[ES20a] Alexis Engelke and Martin Schulz. "Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. 2020, pp. 172–184.

[ES20b] Alexis Engelke and Martin Schulz. "Robust Practical Binary Optimization at Run-time using LLVM". In: *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. LLVM-HPC '20. 2020, pp. 56–64.

[Esm+11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling". In: *38th Annual International Symposium on Computer Architecture (ISCA '11)*. 2011, pp. 365–376.

[EW17] Alexis Engelke and Josef Weidendorfer. "Using LLVM for Optimized Lightweight Binary Re-Writing at Runtime". In: *Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*. 2017, pp. 785–794.

[Fin+19] Hal Finkel, David Poliakoff, Jean-Sylvain Camier, and David F Richards. "ClangJIT: Enhancing C++ with just-in-time compilation". In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019, pp. 82–95.

[FK97] Michael Franz and Thomas Kistler. "Slim binaries". In: *Communications of the ACM* 40.12 (1997), pp. 87–94.

[Fre] Free Software Foundation, Inc. *GNU Compiler Collection (GCC) Internals*. `https://gcc.gnu.org/onlinedocs/gccint/`, accessed 2021-05-12.

[Fre01] Free Software Foundation, Inc. *Infrastructure for Profile Driven Optimizations*. `https://gcc.gnu.org/news/profiledriven.html`, accessed 2021-03-03. Aug. 2001.

[Fre20] Free Software Foundation, Inc. *GNU Bash*. `https://www.gnu.org/software/bash/`, accessed 2021-03-25. Sept. 2020.

[Fre21] Free Software Foundation, Inc. *GCC, the GNU Compiler Collection*. `https://www.gnu.org/software/gcc/`, accessed 2021-03-25. Mar. 2021.

[Gal] Galois, Inc. *APTREE*. `https://galois.com/project/aptree/`, accessed 2021-04-01.

[Gal17] Galois, Inc. *Macaw: Adding instruction Semantics — Overview*. `https://github.com/GaloisInc/macaw/blob/master/doc/AddingInstructionSemantics.org`, accessed 2021-04-01. Nov. 2017.

[Gal21a] Galois, Inc. *Macaw*. `https://github.com/GaloisInc/macaw`, accessed 2021-04-07. Apr. 2021.

[Gal21b] Galois, Inc. *Reopt: A tool for analyzing x86-64 binaries*. `https://github.com/GaloisInc/reopt`, accessed 2021-04-01. Mar. 2021.

[GCC05] GCC Developers. *Link-Time Optimization in GCC: Requirements and High-Level Design*. Nov. 2005.

[GGL12] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly—performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04 (2012).

[Goo21] Google. *The Go programming language – Documentation*. `https://golang.org/doc/`, accessed 2021-03-25. Mar. 2021.

[Gra+00] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J Eggers. "DyC: An expressive annotation-directed dynamic compiler for C". In: *Theoretical Computer Science* 248.1-2 (2000), pp. 147–199.

[Gra+99] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J Eggers. "An evaluation of staged run-time optimizations in DyC". In: *SIGPLAN conference on Programming language design and implementation (PLDI)*. 1999, pp. 293–304.

[Guo+19] Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin. "Transkernel: bridging monolithic kernels to peripheral cores". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 675–692.

[Guo18] Xuan Guo. "Dynamic Binary Translator for RISC-V". PhD thesis. United Kingdom: University of Cambridge, May 2018.

[Gus+19] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. "Performance, correctness, exceptions: Pick three". In: *Binary Analysis Research Workshop*. 2019.

[Gus21] Jens Gustedt. *Improve type generic programming – proposal for C23*. ISO/IEC JTC1/SC22/WG14 N2638. Jan. 2021.

[Hal+15] Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. "Dynamic re-vectorization of binary code". In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE. 2015, pp. 228–237.

[Haw+17] William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. "Zipr: Efficient static binary rewriting for security". In: *International Conference on Dependable Systems and Networks*. DSN '17. 2017, pp. 559–566.

[Hex20] Hex-Rays SA. *IDA Pro*. `https://www.hex-rays.com/products/ida/`, accessed 2021-04-01. 2020.

[Hil19] David Hildenbrand. “An Optimized Intermediate Representation for Binary Rewriting at Runtime”. MA thesis. Munich, Germany: Technical University of Munich, 2019.

[Hon+12] Ding Yong Hong, Chun Chen Hsu, Pen Chung Yew, Jan Jan Wu, Wei Chung Hsu, Pangfeng Liu, Chien Min Wang, and Yeh Ching Chung. “HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores”. In: *International Symposium on Code Generation and Optimization (CGO)*. 2012, pp. 104–113.

[Hsu+11] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, and Wei-Chung Hsu. “LnQ: Building high performance dynamic binary translators with existing compiler backends”. In: *2011 International Conference on Parallel Processing*. IEEE. 2011, pp. 226–234.

[Hsu+15] Chun-Chen Hsu, Ding-Yong Hong, Wei-Chung Hsu, Pangfeng Liu, and Jan-Jan Wu. “A dynamic binary translation system in a client/server environment”. In: *Journal of Systems Architecture* 61.7 (2015), pp. 307–319.

[IBM17] IBM. *Power ISA Version 3.0B*. Mar. 2017.

[IEE08] IEEE. “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (2008).

[Int20a] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Nov. 2020.

[Int20b] Intel Corporation. *Pin 3.18 User Guide*. `https://software.intel.com/sites/landingpage/pintool/docs/98332/Pin/html/`, accessed 2021-06-08. 2020.

[ISO17] ISO/IEC. *Programming languages — C, International Standard 9899:2018*. 2017.

[Jef09] Andrew Jeffery. “Using the LLVM compiler infrastructure for optimised asynchronous dynamic translation in QEMU”. MA thesis. Australia: University of Adelaide, 2009.

[JT08] Ali Jannesari and Walter F. Tichy. “On-the-fly race detection in multi-threaded programs”. In: *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Bebugging*. 2008, pp. 1–10.

[KF01] Thomas Kistler and Michael Franz. “Continuous program optimization: Design and evaluation”. In: *IEEE Transactions on Computers* 50.6 (2001), pp. 549–566.

[Khr19] Khronos OpenCL Working Group. *The OpenCL Specification Version V2.2-11*. July 2019.

[KHY07] Jinpyo Kim, Wei-Chung Hsu, and Pen-Chung Yew. “COBRA: An adaptive runtime binary optimization framework for multithreaded applications”. In: *Proceedings of the International Conference on Parallel Processing (ICPP 2007)*. 2007, pp. 25–25.

[Kir+18] Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel. "PwIN – Pwning Intel piN: Why DBI is Unsuitable for Security Applications". In: *European Symposium on Research in Computer Security 2018 (ESORICS '18)*. 2018, pp. 363–382.

[KMZ17] Jakub Křoustek, Peter Matula, and Petr Zemek. *RetDec: An Open-Source Machine Code Decompiler*. `https://retdec.com/static/publications/retdec-slides-botconf-2017.pdf`, accessed 2021-04-01. Dec. 2017.

[Kol20] Øyvind Kolås. *Generic Graphics Library (GEGL)*. `https://gegl.org`, accessed 2020-09-02. 2020.

[Křo14] Jakub Křoustek. "Retargetable Analysis of Machine Code". PhD thesis. Brno, Czech Republic: Brno University of Technology, Nov. 2014.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization (CGO)*. 2004.

[LLV20a] LLVM Project. *Clang: a C language family frontend for LLVM*. `http://clang.llvm.org/`, accessed 2021-02-04. 2020.

[LLV20b] LLVM Project. *LLVM 11 Documentation: LLVM Language Reference Manual*. `http://releases.llvm.org/11.0.0/docs/LangRef.html`, accessed 2021-03-25. Mar. 2020.

[LLV20c] LLVM Project. *LLVM 12 Documentation: MCJIT Design and Implementaiton*. `https://www.llvm.org/docs/MCJITDesignAndImplementation.html`, accessed 2020-09-04. Sept. 2020.

[LLV21] LLVM Project. *LLVM 12 Documentation: Global Instruction Selection*. `https://llvm.org/docs/GlobalISel/index.html`, accessed 2021-05-14. May 2021.

[Lu+04] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. "Design and implementation of a lightweight dynamic optimization system". In: *Journal of Instruction-Level Parallelism* 6.4 (2004), pp. 332–341.

[Luk+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *SIGPLAN Conference on Programming language design and implementation (PLDI)*. Vol. 40. 6. 2005, pp. 190–200.

[Lyu+14] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. "DBILL: an efficient and retargetable dynamic binary instrumentation framework using LLVM backend". In: *International Conference on Virtual Execution Environments (VEE)*. 2014, pp. 141–152.

[Mak20] Vladimir Makarov. *MIR: A lightweight JIT compiler project*. `https://developers.redhat.com/blog/2020/01/20/mir-a-lightweight-jit-compiler-project/`, accessed 2021-03-04. Jan. 2020.

[Mal+11] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. "An evaluation of vectorizing compilers". In: *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. 2011, pp. 372–382.

[Mal20] David Malcolm. *GCC Wiki: Just-In-Time Compilation (libgccjit.so)*. `https://gcc.gnu.org/wiki/JIT`, accessed 2021-03-04. May 2020.

[Mat+14] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.99.7*. Nov. 2014.

[May87] C. May. "Mimic: A Fast System/370 Simulator". In: *SIGPLAN Symposium on Interpreters and Interpretive Techniques*. 1987, pp. 1–13.

[McK10] Paul E. McKenney. "Memory Barriers: a Hardware View for Software Hackers". In: (June 2010).

[Mes15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. June 2015.

[MFG16] R. Mijaković, M. Firbach, and M. Gerndt. "An architecture for flexible autotuning: The Periscope Tuning Framework 2.0". In: *2016 2nd International Conference on Green High Performance Computing (ICGHPC)*. 2016.

[Mic21] Microsoft, Inc. *How x86 emulation works on ARM*. `https://docs.microsoft.com/en-us/windows/uwp/porting/apps-on-arm-x86-emulation`, accessed 2021-03-30. 2021.

[MM06] Stephen McCamant and Greg Morrisett. "Evaluating SFI for a CISC Architecture." In: *USENIX Security Symposium*. Vol. 10. 2006, pp. 209–224.

[Net04] Nicholas Nethercote. "Dynamic Binary Analysis and Instrumentation, or: Building Tools is Easy". PhD thesis. United Kingdom: Trinity College, University of Cambridge, Nov. 2004.

[NMO11] Takashi Nakamura, Satoshi Miki, and Shuichi Oikawa. "Automatic vectorization by runtime binary translation". In: *Second International Conference on Networking and Computing (NPC 2011)*. 2011, pp. 87–94.

[Noa+17] Matthias Noack, Florian Wende, Georg Zitzlsberger, Michael Klemm, and Thomas Steinke. "KART–a runtime compilation library for improving HPC application performance". In: *International Conference on High Performance Computing*. 2017, pp. 389–403.

[Noe+98] Francois Noel, Luke Hornof, Charles Consel, and Julia L Lawall. "Automatic, template-based run-time specialization: Implementation and experimental study". In: *International Conference on Computer Languages (ICCL)*. 1998, pp. 132–142.

[NS07a] Nicholas Nethercote and Julian Seward. "How to Shadow Every Byte of Memory Used by a Program". In: *Proceedings of the 3rd international conference on Virtual Execution Environments*. VEE '07. 2007, pp. 65–74.

[NS07b] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation". In: *ACM SIGPLAN notices*. Vol. 42. 6. 2007, pp. 89–100.

[Ope20] OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 5.1*. Nov. 2020.

[Pan+19] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. "BOLT: a practical binary optimizer for data centers and beyond". In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, pp. 2–14.

[Pau21] Brian Paul. *MESA Drivers: LLVMpipe*. `https://docs.mesa3d.org/drivers/llvmpipe.html`, accessed 2021-03-25. Mar. 2021.

[PE06] Zhelong Pan and Rudolf Eigenmann. "Fast and effective orchestration of compiler optimizations for automatic performance tuning". In: *International Symposium on Code Generation and Optimization (CGO'06)*. 2006, pp. 332–343.

[Pol+99] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. "'C and tcc: a language and compiler for dynamic code generation". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.2 (1999), pp. 324–369.

[Pol99] Massimiliano A. Poletto. "Language and Compiler Support for Dynamic Code Generation". PhD thesis. Massachusetts Institute of Technology, Sept. 1999.

[Pri19] Soumyakant Priyadarshan. *A study of Binary Instrumentation techniques*. 2019.

[Pyt21a] Python Software Foundation. *Python 3.9 Source — `namedtuple` implementation*. `https://github.com/python/cpython/blob/3.9/Lib/collections/__init__.py#L336-L512`, accessed 2021-03-03. 2021.

[Pyt21b] Python Software Foundation. *Python 3.9.2 documentation*. `https://docs.python.org/3/`, accessed 2021-03-03. 2021.

[Pyt21c] Python Software Foundation. *Python 3.9.2 Documentation – Glossary*. `https://docs.python.org/3/glossary.html`, accessed 2021-03-25. Mar. 2021.

[QEM20a] QEMU Project Developers. *QEMU Documentation*. `https://qemu-project.gitlab.io/qemu/index.html`, accessed 2021-03-25. 2020.

[QEM20b] QEMU Project Developers. *QEMU Source: Translator Internals*. `https://gitlab.com/qemu-project/qemu/-/blob/6fe6d6c9/docs/devel/tcg.rst`, accessed 2021-03-25. Mar. 2020.

[Rio+14] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. "PADRONE: a platform for online profiling, analysis, and optimization". In: *International Workshop on Dynamic Compilation Everywhere (DCE 2014)*. 2014.

[RIS19] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume 1: User-Level ISA, Document Version 20190608-Base-Ratified*. Ed. by Andrew Waterman and Krste Asanović. June 2019.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Global value numbers and redundant computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1988, pp. 12–27.

[Saa00] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. 2nd ed. SIAM, Jan. 2000.

[Sch+01] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. "PLTO: A link-time optimizer for the Intel IA-32 architecture". In: *Proceedings of the 2001 Workshop on Binary Translation (WBT-2001)*. 2001.

[SE94] Amitabh Srivastava and Alan Eustace. "ATOM: A system for building customized program analysis tools". In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. PLDI '94. 1994, pp. 196–205.

[Ser+12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A fast address sanity checker". In: *2012 USENIX Annual Technical Conference*. USENIX ATC '12. 2012, pp. 309–318.

[She+12] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. "LLBT: An LLVM-based Static Binary Translator". In: *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 2012, pp. 51–60.

[SHY14] Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang. "A Retargetable Static Binary Translator for the ARM Architecture". In: *ACM Transactions on Architecture Code Optimization* 11.2 (June 2014).

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: Data race detection in practice". In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WIBA '09. 2009, pp. 62–71.

[Søg20] Jens Kristian Søgaard. *Stack Exchange: How does Rosetta 2 work?* `https://apple.stackexchange.com/a/407733`, accessed 2021-03-30. Dec. 2020.

[Sri+06] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. "HDTrans: an open source, low-level dynamic instrumentation system". In: *International Conference on Virtual Execution Environments (VEE)*. 2006, pp. 175–185.

[SS15] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++". In: *International Symposium on Code Generation and Optimization*. CGO '15. 2015, pp. 46–55.

[Sta21] Standard Performance Evaluation Corporation. *SPEC CPU 2017*. `https://www.spec.org/cpu2017/`, accessed 2021-04-01. Mar. 2021.

[SYE17] Yukinori Sato, Tomoya Yuki, and Toshio Endo. "ExanaDBT: A Dynamic Compilation System for Transparent Polyhedral Optimizations at Runtime". In: *Proceedings of the Computing Frontiers Conference*. CF'17. 2017, pp. 191–200.

[TCH02] Cristian Tapus, I-Hsin Chung, and Jeffrey K Hollingsworth. "Active Harmony: Towards automated performance tuning". In: *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*. 2002, pp. 44–51.

[Tit15] Ben L. Titzer. *V8 Blog: Digging into the TurboFan JIT*. `https://v8.dev/blog/turbofan-jit`, accessed 2021-03-03. July 2015.

[Tra19a] Trail of Bits, Inc. *McSema – Machine code to bitcode: the life of an instruction*. `https://github.com/lifting-bits/mcsema/blob/d9bbd1/docs/LifeOfAnInstruction.md`, accessed 2021-04-01. Oct. 2019.

[Tra19b] Trail of Bits, Inc. *Remill Intrinsics*. `https://github.com/lifting-bits/remill/blob/1579ce/docs/INTRINSICS.md`, accessed 2021-04-01. Oct. 2019.

[Tra20] Trail of Bits, Inc. *Design and architecture of Remill*. `https://github.com/lifting-bits/remill/blob/1579ce/docs/DESIGN.md`, accessed 2021-04-01. Apr. 2020.

[Tra21] Trail of Bits, Inc. *McSema*. `https://github.com/lifting-bits/mcsema`, accessed 2021-03-03. 2021.

[V8 17] V8 Team. *V8 Blog: Launching Ignition and TurboFan*. `https://v8.dev/blog/launching-ignition-and-turbofan`, accessed 2021-03-25. May 2017.

[van07] Michael James van Emmerik. "Single Static Assigment for Decompolation". PhD thesis. University of Queensland, May 2007.

[Wal21] Patrick Walton. *Rust and LLVM in 2021: Progress and Challenges in Code Generation*. LLVM Performance Workshop at CGO'21. Feb. 2021.

[Wan+11] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. "COREMU: a scalable and portable parallel full-system emulator". In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. 2011, pp. 213–222.

[War+12] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. "Securing untrusted code via compiler-agnostic binary rewriting". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. 2012, pp. 299–308.

[War13] Henry S. Warren, Jr. *Hacker's Delight*. 2nd ed. Addison-Wesley, 2013.

[WB16] Josef Weidendorfer and Jens Breitbart. "The Case for Binary Rewriting at Runtime for Efficient Implementation of High-Level Programming Models in HPC". In: *Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*. 2016, pp. 376–385.

[WKT04] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. "A tool suite for simulation based analysis of memory access behavior". In: *4th International Conference on Computational Science*. ICCS '04. 2004, pp. 440–447.

[WLK13] Ryan Whelan, Tim Leek, and David Kaeli. "Architecture-independent dynamic information flow tracking". In: *International Conference on Compiler Construction*. 2013, pp. 144–163.

[You+20] Xin You, Hailong Yang, Zhongzhi Luan, Depei Qian, and Xu Liu. "ZeroSpy: Exploring Software Inefficiency with Redundant Zeros". In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. 2020, pp. 397–410.

[YS19] S. Bharadwaj Yadavalli and Aaron Smith. "Raising Binaries to LLVM IR with MCTOLL (WIP Paper)". In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2019. 2019, pp. 213–218.

[Zha+14] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. "A platform for secure static binary instrumentation". In: *International Conference on Virtual Execution Environments*. VEE '14. 2014, pp. 129–140.

[Zha+15] X. Zhang, Q. Guo, Y. Chen, T. Chen, and W. Hu. "HERMES: A fast cross-ISA binary translator with post-optimization". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 246–256.

[ZS13] Mingwei Zhang and R Sekar. "Control flow integrity for COTS binaries". In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 337–352.