Robust Practical Binary Optimization at Run-time using LLVM

Alexis Engelke
Technical University of Munich
Munich, Germany
engelke@in.tum.de

Martin Schulz
Technical University of Munich
Munich, Germany
schulzm@in.tum.de

***Abstract*—In High Performance Computing (HPC) the performance of applications is paramount, which is has led to a wide body of work on optimizing compilers. However, compilers alone are naturally limited in their potential, as they cannot exploit run-time information to specialize generated code. For this reason, run-time components to perform such specializations are gaining traction. In particular, application or use-case driven specializations have a high potential for optimizations, as they can benefit from explicit guidance from the application or the user, which allows for targeted changes incorporating specific knowledge. Because full compilers and sources are usually unavailable during computation, specializations have to be done at machine code level. However, existing libraries for binary specialization face structural limitations of separating optimization and code generation, in addition to numerous implementation gaps.**

In this paper, we describe *BinOpt*, a novel and robust library for performing application-driven binary optimization and specialization using LLVM. A machine code function is lifted directly to LLVM-IR, optimized in LLVM while making use of application-specified information, used to generate a new specialization for a function, and integrated back to the original code. We apply this technique to existing optimized code and show that significant performance improvements can be observed with only a small optimization time overhead.

***Index Terms*—High Performance Computing, Dynamic Code Generation, Run-Time Optimization, Binary Translation, LLVM**

I. Introduction

Performance and efficiency are very important properties of computational applications, especially in High Performance Computing (HPC). Therefore, modern optimizing compilers support a large set of code transformations and optimizations to achieve good usage of modern computing platforms. However, the traditional approach of a strictly separate compilation step before execution has a major drawback: not all information is available at compile-time and this limits optimization possibilities. For example, high-level programming models typically determine their data layout at run-time, causing an additional level of indirection and frequent redundant computations of offsets and indices. Also, many applications have an initial setup phase, where the configuration is processed. Resulting information like matrix sizes is only available at run-time, preventing the elimination of function dispatchers and limiting vectorization possibilities.

While it is theoretically possible to generate several variants during compilation, this is highly impractical due to constraints in code size and compilation time and is also infeasible, if the possible value range is too big to cover all likely variants. The latter is further aggravated by increasing complexity of architectures and system design choices.

As a consequence, it is becoming more and more essential to integrate a run-time component to achieve significant performance increases. Such a component can specialize performance sensitive code in order to exploit the availability of parameters and design options at run-time. However, as it is generally hard to automatically predict which information is actually run-time constant and what their corresponding performance impact is, such specialization is most beneficial if it is guided explicitly by the application itself, an approach that we refer to as "application-guided optimization".

This approach, however, comes with several challenges, in particular in HPC environments. For example, simply triggering a re-compilation is generally not possible, as compilers on compute resource may not be accessible and, even if they were, the source code may not available while the application is running. Further, the time needed for compilation is a limiting factor for many applications, which can easily negate any potential performance gain. To avoid these problems, optimizations must, therefore, be done directly at machine code level, which also has some additional benefits: first, it makes the optimization approach independent from the compiler used, allowing the use of a closed-source vendor-supplied compiler, and second, it enables optimizations across boundaries of programming language and even optimization of functions from pre-compiled shared libraries.

This approach of application guided optimization has been previously explored, e.g., by DBrew [1], which is a library for binary optimization at run-time. It provides a C API for applications, where functions can be specialized for known parameters and associated memory regions. During optimization, instructions that use constant input operands are replaced with the computational result, loops are unrolled completely if possible, and invoked functions are unconditionally inlined. This approach is very fast, but also very specific and surgical and hence, unfortunately, provides very limited optimization possibilities and may lead to a significant size of newly generated code. With DBrew-LLVM [2], the DBrew-optimized code was lifted to LLVM, where further optimizations were performed. This resulted in higher-quality machine code,

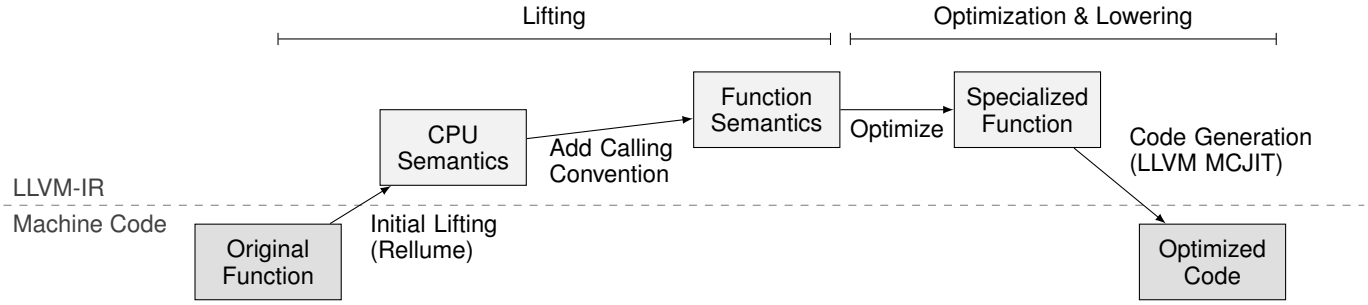


Fig. 1: Overview of the lifting, optimization and code generation flow for run-time optimizations. The original function is lifted from machine code to LLVM-IR for optimizations. The newly generated machine code is placed in the same address space and can be used as drop-in replacement by the application.

but also suffers from other drawbacks in addition to the limitations of DBrew: it is not possible to resolve indirect calls or jumps at LLVM-IR level due to the strict separation between DBrew's initial optimization and the LLVM-based post-processing. Thus, DBrew-LLVM can only be applied to a very limited range of code, which is further restricted by the small instruction coverage.

Based on these previous experiences, we propose a new approach that directly transforms native machine code to LLVM-IR without the need for an intermediate step, like it was necessary in DBrew. This approach is implemented in our library named *BinOpt*. For the actual lifting of machine code semantics, we use Rellume [3], a library designed for producing performant LLVM-IR from x86-64 machine code, providing almost complete coverage of architectural x86-64 instructions including SSE/SSE2. As there is no strict separation between lifting and optimization, further code can be lifted as needed, e.g., when target addresses of indirect jumps become known. Further, with only minor modifications of the lifted IR code created by Rellume, it is possible to support cases where jump targets stay unknown during optimization by continuing at the original code.

The main contributions of this paper are the following:

- We re-use the LLVM compiler infrastructure for practical and efficient run-time optimization of machine code.
- We offer a robust integration into existing code, including proper handling of unsupported corner cases.
- We show first experiences with applying our run-time optimization techniques to real-world code.

Our results show that significant performance gains can be achieved when specializing code for sparse data structures. Moreover, we show that applying BinOpt to the optimized and widely used image processing library GEGL [4] allows for a performance improvements up to 33%.

The remainder of this paper is structured as follows: in the next section, we will give an overview on the general procedure for our run-time binary optimization approach as well as the usage in application code. In Section III, we will describe our approach to lifting x86-64 machine code to LLVM-IR. Afterwards, in Section IV, we show our optimization procedure and in Section V we explain our support for special cases found in production codes. Then, in Section VI, we describe our benchmarks used to evaluate our approach and show our findings. Finally, we cover related work in Section VII and conclude with ideas for further improvements in Section VIII.

II. Binary Optimization Procedure

To enable application-guided optimization of binary code, we provide a library where applications can specify a code section along with parameters that are constant in the particular execution. This library is named *BinOpt*[1]. The high-level approach is similar to DBrew [1]: the application specifies a function, which is specialized and optimized for a given configuration, e.g., constant parameters or the target architecture. The new function created by the optimization library has the same interface as the original function and therefore can be used instead of the original input function.

Optimization at function granularity has many benefits: 1) functions have a clearly defined boundary and ABI; 2) this allows for a simple adaption to other programming concepts like C++ lambdas and even other programming languages; 3) providing an optimized replacement with the same interface allows for an easy replacement in the application code; and 4), in case of an error condition, it is always possible to return a pointer to the original function, so that the application does not have to handle any kind of rewriting errors.

Our current implementation builds on top of the LLVM framework and lifts binary code to its intermediate representation. LLVM offers a widely used and stable infrastructure and opens BinOpt to a wide range of optimization passes. At this moment, it only supports the x86-64 architecture on Linux with functions following the System V ABI [5]. The general concepts, however, apply to other systems and architectures as well.

A. Workflow for LLVM-based optimization

The general procedure for doing optimization of binary code using LLVM is the following (see also Figure 1):

1) Initially, the function specified by the application is lifted to LLVM-IR using Rellume [3]. This results in a single

[1]https://github.com/aengelke/binopt

```
1  int func(int a, int b) { return a - b; }
2  int main(void) {
3    // Create a new handle into the library
4    BinoptHandle h = binopt_init();
5    // Create a configuration for func
6    BinoptCfgRef bc = binopt_cfg_new(h, func);
7    // Specify signature, two parameters
8    binopt_cfg_type(bc, 2, BINOPT_TY_INT32,
9                BINOPT_TY_INT32, BINOPT_TY_INT32);
10   // Specify second parameter as constant 42
11   binopt_cfg_set_parami(bc, 1, 42);
12
13   int(*nfn)(int, int) = binopt_spec_create(bc);
14
15   // Call new version, uses 42 instead of 16
16   int res = nfn(48, 16);
17 }
```

Fig. 2: Basic usage of the binary optimization library. The library calls are added by the application developer to specialize a function for constant values.

LLVM-IR function that represents the semantics of the machine code.

2) Then, the function is wrapped to apply the calling convention, so that the LLVM function has a signature matching the original function. Parameter registers and return values are correctly wired to the registers in the lifted machine code semantics. At this point, constant parameters are propagated.
3) This function is then optimized, during which accesses to constant memory regions are also replaced with their constant values. Whenever target addresses of calls or indirect jumps become known during optimization, these code fragments are lifted as well and this step is recursively repeated until no further code can be discovered.
4) After optimization, the LLVM-IR code is compiled to a new machine-executable function using the MCJIT [6] compiler provided by LLVM.

If target addresses of indirect jumps or calls cannot be determined during optimization, the register state is restored and execution continues in the original code at that point.

B. Triggering BinOpt from the Application

The BinOpt library provides a C API for generating specializations of a function. A simple usage example of the library is shown in Figure 2.

At the very beginning, a new handle into the library has to be created (Line 4), which allows using multiple handles in different threads. Then, a new rewriting configuration for a specified function is created (Line 6). Currently, the library requires the signature to be explicitly specified via a library call so that the calling convention can be applied correctly. In future, this information may also be fetched from debug information sections, like DWARF [7] or CTF [8].

At this point, further configuration for optimizations can be added. This includes setting constant values for parameters, marking specific memory regions as constant and enabling other transformations like floating-point math optimizations.

After configuration, a new specialization can be generated from the configuration (Line 13). The new function has the same parameters and return values as the original function. In this example, after successful optimization the value for `res` (Line 16) will be 6[2].

III. Lifting x86-64 to LLVM-IR

The key challenge of using LLVM for run-time binary optimization is lifting machine code to LLVM-IR code. The generated LLVM-IR code should not only have the same semantics, but also be structured in a way that it interacts well with the existing optimization and code generation infrastructure, such that the generated machine code is also performant. Further, not only the performance of the final code is important, but also the optimization time, as rewriting happens at run-time.

In this section, we describe our approach to lifting machine code to LLVM-IR and our handling of calls to other functions, indirect jumps and calling conventions.

A. Instruction Semantics and Main Control Flow

For lifting instruction semantics and the control flow of machine code to LLVM-IR, we opted for Rellume [3], a library designed for generating efficient LLVM-IR code in the context of dynamic binary instrumentation. As such, the library also focuses on the performance of the lifting process and provides a very good coverage of x86-64 instructions, making it suitable for our purpose of run-time optimization. In the following, we will summarize the general lifting procedure of Rellume and important properties of the produced LLVM-IR code.

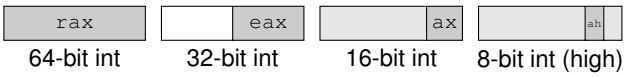
When lifting machine code, Rellume creates a single LLVM-IR function for a set of instructions. Control flow between the instructions is setup properly. The function returns if either the next instruction was not lifted, or the address of the next instruction is entirely unknown because of an indirect jump, call or return instruction.

A lifted LLVM-IR function takes a single parameter: a pointer to the *CPU state* structure. This structure contains the state of all registers, including the instruction pointer (`rip`) and status flags (`rflags`). Before the function returns, the updated values are written back to the structure. As only the instruction semantics is lifted, Rellume does not take care of lifting calling conventions and also does not assume that a `call` returns properly[3].

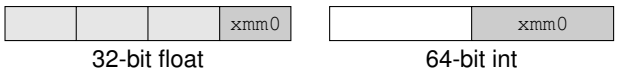
The lifting process itself is done in three stages: 1) instructions are decoded, including following both targets of conditional jumps, and a control flow graph of basic blocks is constructed; 2) the instructions are lifted individually into a new LLVM-IR function; 3) branches between LLVM-IR basic blocks are added and the PHI nodes, which merge values from

[2]When optimization fails, the original function is executed and the result will be 32.

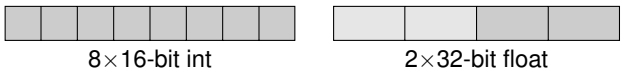
[3]This indeed happens in ordinary C/C++ code with exceptions or `setjmp`/`longjmp`.



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



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



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

Fig. 3: Examples for facets as used by Rellume during lifting [3]. Registers can be accessed in different data types, which are propagated separately to reduce the number of conversions and insert/extract operations.

preceding blocks, for propagating register values across basic blocks are filled.

One of the key aspects in Rellume to ensure that the created LLVM-IR code is highly optimizable by subsequent optimization passes, is the fact that it handles registers of different types separately, depending on the actual instructions. For example, a general-purpose register on x86-64 may be accessed as full 64-bit register, but also as 32-bit or 8-bit register, as depicted in Figure 3. Rellume avoids frequent casts and insert/extract operations by propagating such individual facets of a register separately, and—most importantly—also across basic blocks. This is a very relevant improvement, as LLVM's standard optimization passes rarely change types of PHI nodes and therefore prevent elimination of many casts and insertions/extractions, which in turn prevents constant propagation as well as further analyses and optimizations.

B. Calls and Indirect Jumps

One problem during lifting occurs, when the address of the next instruction is unknown, for example because of an indirect jump. For these cases, the previously described way of lifting functions would result in a return of the lifted function. This behavior is unsuitable for our use case, where the lifted function should only return if the function *itself* is finished. We also do not want to always inline calls, as this may lead to code explosion and is problematic with recursive code. Instead, we want to retain call semantics at LLVM-IR level.

Thus, we modified the lifting process as follows: function calls (direct or indirect) are lifted to a call of a helper function in LLVM-IR. We now assume that a function entered with a `call` instruction returns ordinarily. Indirect jumps are lifted to a tail call of a helper function. The only remaining possibility where the lifted function returns is on a `ret` instruction, which we assume to imply an ordinary return of the original function.

Both helper functions take a pointer to the CPU state as parameter. As an optimization, the functions also receive the address of the target and the current virtual stack pointer as separate parameters, although the information is also contained in the CPU state. This allows for an easy detection when the jump address becomes constant and also simplifies the computation of the stack offset. An example of such a helper call is shown in Figure 4a. The helper functions are either replaced with more lifted code if the address becomes constant during optimization (see Section III-D) or lowered at the end to an indirect jump to the native code (see Section V).

C. Calling Convention and Constant Parameters

To create a function replacement with the same signature, an LLVM-IR with a corresponding signature has to be created. This function creates a stack-allocated CPU state structure and initializes the parameter registers according to the calling convention with the parameters from the function, or the specified constant of the configuration.

Additionally, a virtual stack with a 16-byte alignment is allocated on the stack. Parameters which are passed on the stack are copied into the virtual stack, and the virtual stack pointer is set to the end of the argument region. As with other systems, like DBrew-LLVM [2], the virtual stack can only have a fixed size, which can be configured by the application.

Next, a call to the Rellume-lifted function is generated, which is inlined immediately afterwards. Finally, the return values are read from the CPU state according to the calling convention and the specified signature and returned from the LLVM-IR function.

We note that during optimization, the CPU state structure will most likely be removed by the scalar-replacement-of-aggregates (SROA) pass. In many cases, the virtual stack will also be eliminated, resulting in more optimizable and idiomatic LLVM-IR code.

D. Code Discovery

Although the targeted function is user-specified and all code reachable via direct and conditional branches is lifted by Rellume, it can happen that more code becomes reachable during the optimization. Examples include callback functions or dispatch functions, which select the function for the actual work depending on input parameters. In both cases, the call target is generally not predictable during lifting and only gets known during optimization. However, as indirect calls not only have a high latency, but also require some overhead to follow a calling convention, it is in many cases beneficial to inline such function calls.

We solved this problem using a fixed-point algorithm: after lifting, we optimize the LLVM-IR code. Then we check whether there are calls to helper functions with a constant target address. If so, code from that address is lifted to separate

functions; which is either called directly for calls or inlined afterwards for indirect jumps. Otherwise, all reachable code has been lifted and there is no further code.

As it may take several iterations until all code is discovered, a significant amount of time is spent for optimization. Therefore, we not always run a full optimization pipeline but only run a small subset of light optimization passes first and check if this yields more discoverable code. Only if this is not the case, a full pipeline is run.

At the end, no calls to helper functions with a known address remain, as all code with a known address is lifted to LLVM-IR and is either replaced with a direct call at LLVM-IR level or inlined immediately.

A special case are function stubs in the Procedure Linkage Table (PLT), which are used for linking against shared libraries. These stubs usually only contain an indirect jump to the implementation in the shared library using the Global Offset Table (GOT) – or to the dynamic linker, if the function has not been resolved so far. This is problematic for two reasons: first, we do not want to lift or transform the code of the dynamic linker, as it is executed at most once. Second, the GOT is usually mapped as partially writable to enable lazy binding, meaning that the target address cannot be resolved at all during optimization. Thus, we do not lift functions that look like PLT stubs. As a side-effect, this also improves performance as the overall number of different indirect jumps is reduced, leading to a better branch target prediction. We also note that we also want to be more selective regarding which library functions to optimize: a C standard library like *glibc* provides complex functions like `malloc`, which usually have a very low optimization potential, and highly optimized routines with hand-written assembly code like `memcpy`, where LLVM is unlikely to produce a performant re-compilation.

IV. Extended Optimization Pipeline

In this section we describe, how we adapt the existing standard optimization pass sequence to the use case of optimizing LLVM-IR derived from machine code.

For the main optimization run, we use a slightly modified pass sequence of the default `-O3` pipeline.

A. Improved Alias Analysis

First, we add a new plug-in for the memory alias analysis infrastructure to improve alias analysis when pointers to the CPU state structure are involved. Accurate aliasing information regarding this structure is critical for optimization, because otherwise the structure may not be optimized away, causing all registers values to be frequently stored on the stack. On the other side, the lifted code contains many `inttoptr` instructions for memory accesses, which are problematic for alias analysis, as usually no further information about the resulting pointer is available.

However, we know that the original machine code is unable to access that structure and thus that any `inttoptr` instructions can never point to the CPU state. Even further, due to our way of lifting, we know that a pointer to the CPU state is never stored in memory and therefore only pointers derived from the original allocation may point to the structure.

Consequentially, we add a new function analysis, which returns *no alias* for all queries where exactly one pointer is based on the stack allocation of the CPU state. Therefore, if the CPU state is not required for other non-inlined functions, it will be optimized away by the scalar-replacement-of-aggregates (SROA) pass.

B. Propagating Constants from Memory

In a usual compiler setting, loads from constant addresses are rare, e.g., for accessing memory-mapped I/O devices or other hardware functionalities. As LLVM has no way to reason about such accesses, they are not modified further during optimization.

In our setting, however, such kinds of accesses may occur significantly more often, for example when accessing global variables or constant pointers that get propagated from parameters. We also have more information available, as an application can mark specific memory ranges as constant. Additionally, we can get information from the memory mapping[4] about constant memory regions mapped from the executable file itself or other shared libraries.

As such cases are not covered by the existing LLVM passes, we add a new pass to the pipeline, which iterates over all load instructions with a constant address operand. For each such instruction, the value of the address operand is evaluated. If the load is identified to only access data marked as constant, the value of the load instruction is replaced by the constant value.

This process is scheduled several times in the pass pipeline, currently by using the *peephole* callback of the pass builder.

C. Folding Integer-Pointer Casts

In rare cases when propagating constant pointer parameters, sequences of `ptrtoint–add–inttoptr` can occur, which will not always be folded by standard optimizations, blocking further optimizations due to worse alias analysis results. To improve the optimization in these cases, we add a pass, which folds such sequences to a `getelementptr` instruction for pointer arithmetic.

V. Handling Indirect Jumps

In some cases, indirect jumps or calls cannot be resolved during optimization. There are three possible ways to handle such cases: 1) the rewriting process can abort with an error, simply returning the original function. This, however, prevents optimizations up to the indirect jump, which may even never be executed; 2) it might be possible to trigger a new rewriting process during function execution once the jump address is known, which allows for more optimizations, but has unpredictable implications on the time required for rewriting, especially if there are many possible jump targets; and 3) (which we apply here) we can restore all registers and

[4] `/proc/self/maps` on Linux, if available.

```
; ...
; Indirect jump to address %next_rip
call void @native_cont(i8* %cpu_state,
                       i64* %rsp, i64 %next_rip)
; ... load %rax from %cpu_state
ret i64 %rax
```

(a) LLVM-IR Code after lifting. The helper function conceptually modifies the CPU state and is replaced with inline assembly during lowering. If `%next_rip` becomes a constant, the call is removed and further lifted code is added.

```
// ... register set by inline asm constraints;
//     rax/rcx/rdx stored in CPU state

// Setup iretq area on top of stack
movq   [rsp], rax        // next_rip
movq   [rsp+0x8], 0x33   // cs
movq   [rsp+0x10], 0x202 // eflags
movq   [rsp+0x18], rcx   // virt. stack
movq   [rsp+0x20], 0x2b  // ss
// Store "our" rsp in a global variable
movabs rdx, glob_slot
movq   [rdx], rsp
// Change return address of virtual stack
// so that the code returns to "cont" below
leaq   rdx, [rip+cont]
movq   [rcx], rdx
// Restore remaining registers from CPU state
movq   rax, [rsp+0x28]
movq   rcx, [rsp+0x30]
movq   rdx, [rsp+0x38]
// No need to change rsp, iretq data is
// already at the top of the stack
iretq

// The code we jumped into returns here.
// Registers with return values are set.
// RSP still points to the user stack.
cont:
  // Restore "our" rsp, stored above.
  movabs rsp, glob_slot
  movq   rsp, [rsp]

  // ... Clean up stack frame
  ret
```

(b) Machine code after lowering. The code jumps to unknown target using an interrupt return, ensuring that *all* registers including the stack pointer are set correctly. After the unknown target returns, the stack pointer is restored and the function exits ordinarily.

Fig. 4: Example for handling an indirect jump during lifting/optimization and machine code generation.

continue at the original code, offering a good trade-off between optimizations and predicable execution and rewriting times.

For the latter option, to continue the execution at the original code location, the full register state has to be moved into the corresponding architectural registers. Restoring *all* registers is not directly possible with LLVM, because the stack frame of the native function is actually *part of* the stack frame of the new function. This leads to two challenges: first, the stack pointer `rsp` needs to set to the address of the top of the virtual stack; and second, after the native code finished (i.e., returned using `ret`), the LLVM function must clean up the real stack and therefore needs the previous stack pointer address in the register `rsp`.

We solve the first problem using an inline assembly fragment, which is based on the x86-64 instruction `iretq`. This instruction is designed for an interrupt return and takes the value for the stack pointer, the instruction pointer and the flags register from the stack. To make use of this instruction, a sufficiently sized temporary buffer is allocated and populated appropriately. Most registers, mainly except for `rsp` and `rip`, are set up using inline assembly constraints with values loaded from the CPU state. Inside the inline assembly fragment, the stack pointer is set to the temporary buffer and `iretq` is executed. This enables us to continue the native code with all registers having proper values.

To solve the second problem, we have to change the return address on the virtual stack, which will be used by the original code, and restore the original stack pointer. Therefore, the original stack pointer is stored in a global variable, whose address is a constant at link-time. At the new return point for the native function, the stack pointer is loaded again from that variable. Finally, the new register values are accessed using appropriate inline assembly constraints and are written back to the CPU state. The resulting machine code after optimization is shown in Figure 4b.

For indirect calls, the procedure is very similar, with one exception: instead of the return address of the current function the return address of the callee has to be changed, which resides at the top of the stack because of the call instruction.

This method provides a fully functional way to continue at arbitrary machine code from an LLVM-IR function. However, we note that the use `iretq` incurs a performance penalty and may also lead to a mismatch of return addresses in the return address stack of the branch target predictor inside the processor. Thus, the execution of indirect calls or jumps is best avoided inside the lifted code.

VI. Benchmark Results

To evaluate the use and performance gains of our optimization technique, we optimize different kernels using our library. We optimize a stencil kernel for comparison with previous work [1], [2] and also optimize two real-world image processing operations. We compare the run-time of the optimized code, including the time spent for optimization itself, against the same code without optimizations applied (i.e., no rewriting occurs). We also tried comparing against DBrew [1] and Drob [9], but these libraries failed on all benchmarks due to several unsupported instructions or buffer size limitations for the generated code, which—at least — highlights the general applicability of our new approach.

A. Benchmarks

Micro benchmarks: In the *stencil kernel* micro-benchmark case, a function that applies a generic 2d stencil to a matrix is specialized using the run-time configuration describing the stencil itself and the matrix size. Such a kernel with an unknown stencil is frequently used in HPC and image processing applications, as well as artificial neural networks codes. In this

$$\{(1, 0, 0.25), (0, -1, 0.25), (0, 1, 0.25), (-1, 0, 0.25)\}$$

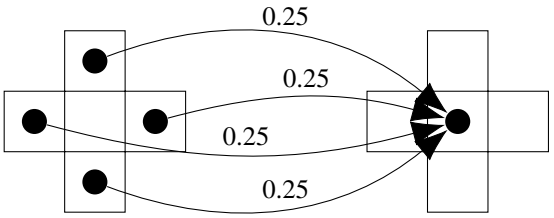


Fig. 5: Stencil used for the *stencil* micro-benchmark. The stencil is specified as an array of triples of offsets and weight.

case, the stencil is given as a list of triples consisting of Δx, Δy, and the multiplication factor. We specialize the kernel on a 4-point sparse stencil of size 3×3, which computes the average of the four directly neighbored matrix entries (cf. Figure 5). In our input configuration, we use a matrix of size 649×649 and run 10,000 iterations. In each iteration, the result is written into a separate matrix, resulting in a Jacobi-style iteration. We compare the performance of the code optimized from binaries at run-time against the original code and also against a code specialized at compile-time with the particular stencil.

GEGL kernels: To evaluate the performance impact on optimized real-world code, we adapt two kernels from the GEGL image processing library [4] to use our binary optimization library on its respective main computations. The general process for the adaption is the following: we splice out the part of the code we wanted to specialize into a separate function, passing required variables, configuration options and results from the algorithm setup as parameters. Then we generate a specialized version of the function by setting configuration parameters and associated memory regions as constant. A call to the returned function finally replaces the code previously spliced out.

The first operation we adapt is *gblur-1d*, where we optimize the finite impulse response filter of the Gaussian 1d blur filter. This is essentially a 1d stencil of a variable size, applied to the all channels of every pixel of the image with single-precision floating-point values. The kernel for specialization consists of 2 nested loops, one for the dimension of the stencil and one for the color channels. The kernel is called separately for each pixel. As input, we use an image of size 9000×4923 pixels, which is internally processed with four channels (RGBA) as single-precision floating-point numbers. The standard deviation is set to 1.5, resulting in a dense stencil size of 11 elements.

The second operation is the *bilateral-filter*, which applies a variably-sized 2d gaussian-weight stencil on all channels for every pixel of the image. We first adjust and replace the call to the *exp* function with a corresponding polynomial approximation to avoid external function calls. The kernel for specialization itself consists of 5 nested loops, two for the dimension of the image, two for the dimension of the stencil

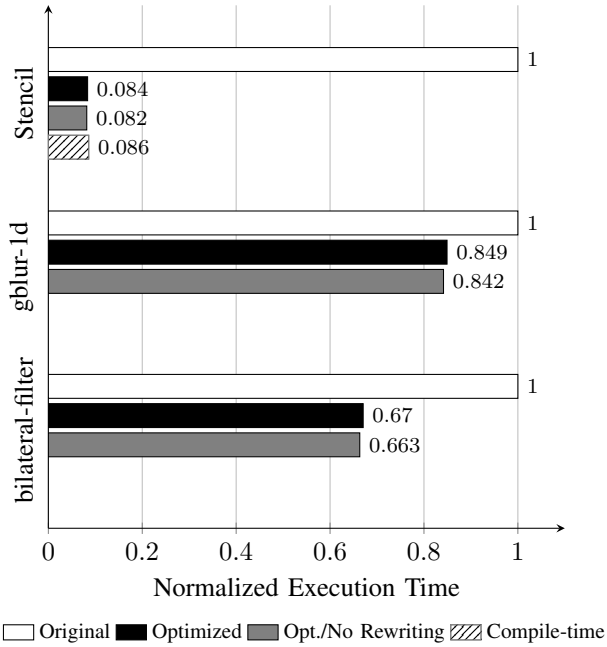


Fig. 6: Benchmark run-times normalized against the execution time of the original code, shown for the run-time optimized code with and without time needed for optimization.

and one for the color channels. The kernel is called once for processing the entire image, As input, we use an image of size 3000×1641 pixels, which is internally processed with four channels (RGBA) as single-precision floating-point numbers. The blur radius is set to 4.0, resulting in a dense stencil with a size of 9×9 elements.

For both GEGL kernels, we compare our run-time optimized version, including rewriting time, against the same code without run-time optimizations. We did not compare against a compile-time specialized variant, as all additional information entirely depends on the configuration specified by the user.

B. Setup

All benchmarks are compiled using GCC 9.2.1 with the options `-Ofast -mtune=native` and linked against glibc 2.30. For run-time optimization, we use LLVM 9.0 and Rellume commit `be7e2db`. We disable AVX for the benchmarks and also for the code generated by the LLVM at run-time, as the lifter does not yet support AVX. We use Fedora 31 with Linux kernel 5.7.15 in 64-bit mode as the OS and run on an Intel Core i5-8250U CPU clocked at 1.6 GHz (3.4 GHz turbo). All experiments were run five times, we show the average of these runs.

C. Results

The main performance results of our experiments are shown in Figure 6 and further information about the run-time optimization is listed in Table I.

TABLE I: Results of the conducted benchmarks showing performance improvements, properties of the original code, and performed optimizations.

	Stencil	gblur-1d	bilateral-f.
Original run-time	17.712s	2.992s	5.461s
Rewriting time	0.031s	0.023s	0.039s
Opt. run-time (w/rew.)	1.481s	2.540s	3.662s
Overall improvement	−16.231s	−0.452s	−1.799s
Original code size	290 B	363 B	680 B
Rewritten code size	587 B	583 B	567 B
Nested loops	3	2	5
Loop unrolling	Yes	Yes	Yes
Constant propagation	Yes	Yes	Yes
Vectorization	Yes	Yes	No

Stencil benchmark: On the stencil micro-benchmark, we can observe significant performance improvements, leading to a speed-up factor of 11.97. By specializing on the sparse data structure, the innermost loop iterating over the stencil points can be unrolled, allowing a much more efficient computation of the stencil offsets. Unrolling the inner loop also makes vectorization profitable, leading to further performance improvements.

With the previous approach in DBrew-LLVM [2], vectorization was problematic, which caused the run-time optimized code to be 2 times slower than the compile-time optimized code. With our current approach, the run-time optimized code is even slightly faster than the code optimized at compile-time by GCC using the same information, supposedly by minor differences in instruction selection and ordering. We attribute this improvement over the approach from DBrew-LLVM to more information being available as a consequence of the direct lifting and also general improvements to the LLVM vectorization infrastructure.

1d Gaussian Blur: For the *gblur-1d* benchmark the performance improves by 24%. Specialization on the convolution matrix and the number of color channels enables loop unrolling. This not only eliminates loop overhead and index computations, but also allows for more efficient use of vector instructions.

Bilateral Filter: Run-time optimization of the *bilateral-filter* operation improves performance by 33%. Specialization on the convolution matrix allows for a simplification of boundary checks and reduces required integer arithmetic for index and offset calculations. Although the function generally has a higher computational intensity compared to the Gaussian blur filter, the performance gains are higher as the Gaussian Blur filter uses other GEGL functions for handling boundary conditions, which are not optimized.

D. Optimization Time

As it can be seen in Table I, the rewriting times are generally low, so that usual workloads can easily compensate the optimization time on these benchmarks. We observe that the optimization time increases with the complexity of the transformed function. A more complex control flow, for example from nested loops, increases time of analysis and transformations. For example, the kernel of the Gaussian blur operation always processes a single pixel at a time and is called separately for each pixel. This reduces optimization time, as the resulting kernel is very simple. For the bilateral filter operation, however, the entire loop iterating over all pixels is part of the optimized function. This can immediately be seen in the almost doubled rewriting time.

This insight is very relevant for application use: generally, the function selected for optimization preferably should only contain optimizable code portions and therefore have a limited scope. Including other code parts will not result in performance improvements during execution, but may significantly increase rewriting times. Especially if calls to other functions without benefits from specialization are involved, the rewriting scope may grow very quickly.

E. Discussion

Our results show that run-time optimizations allow for further performance increases beyond the traditional workflow of separate compilation, even on already optimized real-world applications. The time needed for actually performing the optimizations is negligible and it can be amortized easily using a sufficiently large workload or if the optimized code can be re-used several times.

The complexity of the code selected for rewriting turned out to be the main force for high rewriting times, for which reason our technique should only be used for selected code parts which actually benefit from the availability of further run-time-only information. Indirect calls to other functions are supported by our library, but currently incur a measurable performance penalty and consequentially are best avoided.

VII. Related Work

DBrew [1] is a prototypical run-time binary optimizer, which introduced the concept of rewriting functions guided by the application itself using an API, specializing on parameters. This approach is limited by the strictly local scope of optimizations, which not only leads to missed optimization but also to a sizable amount of generated code. DBrew-LLVM [2] provided an initial LLVM-based code generation back-end with a performance-oriented x86-to-LLVM lifter. This work is based on our previous experiences, but avoiding problems of a strict separation between optimization and code generation and the tight integration into DBrew. Drob [9] explores the approach of a lower-level code representation to avoid most parts of a costly lowering process, but has currently strong limitations in instruction coverage and supported code transformations.

Another possibility to include application knowledge into run-time optimization is using language extensions. This was proposed previously with 'C [10], [11], Tempo [12], [13] and DyC [14], [15], among others. ClangJIT [16] is the most recent proposal for a compiler-integrated system, where the optimizable parts of the program are stored in an intermediate representation (Clang AST) in the compiled binary file. These

approaches require a special compiler and do not allow for optimization across different programming languages or shared libraries, as we support in our approach.

Other dynamic binary optimizers like Dynamo [17], Mojo [18] and ADORE [19] transparently translate and optimize the entire program without any further information from the application. None of these systems are currently under active development and modern processors have shown strong performance improvements in the past decade, significantly reducing the benefit of plain tracing optimizations.

For lifting x86-64 machine code to LLVM-IR, several systems exist. McSema [20] lifts compiled binaries statically to LLVM-IR, reconstructing functions, function types and global variables, with focus on reverse engineering and analyzing unknown executable files. Other static binary analysis systems include Retdec [21] and fcd [22], which both use LLVM as intermediate code representation during decompilation. These systems have in common that they are designed for binary analysis, without focus on performance of the lifted code.

Rellume [3] is a library lifting arbitrary x86-64 machine code to LLVM-IR, originally created for use in a Dynamic Binary Translation framework and therefore also focusing on lifting and optimization performance, matching our needs for this work. DBrew-LLVM [2] focused on lifting for run-time code generation, but suffered from structural and instruction coverage limitations. However, DBrew-LLVM provided initial experiences, which influenced Rellume and also this work.

VIII. Conclusion

In this paper, we have shown our advances on optimizing compiled binary code at run-time using LLVM. In particular, we provide a new way for reconstructing LLVM-IR code from raw machine code with the goal of practical and efficient optimization, which is also able to dynamically lift further code during optimization. With only minor modifications, we were able to re-use the standard optimization infrastructure to provide high-quality code transformations in the context of run-time binary rewriting. Our approach for continuing at the original code on unhandled or error cases makes a robust integration into generic application code possible. Finally, we also showed our first experiences with adapting existing optimized code to our technique and found that some kinds of applications can greatly benefit from exploiting run-time only information with a comparably low cost for optimization itself.

Further guidance for application developers can be provided by tool integration into existing compilers, which point out missed optimizations due to unavailable information and give a performance benefit estimation. The latter is especially relevant as performance characteristics of high-end CPUs are hard to predict. A separate infrastructure for system-wide caching of rewritten code fragments can further reduce optimization times, especially on HPC systems where multiple nodes may do similar transformations.

References

[1] J. Weidendorfer and J. 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.

[2] A. Engelke and J. 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.

[3] A. Engelke and M. 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.

[4] "Generic graphics library (gegl)," https://gegl.org, accessed 2020-09-02.

[5] J. Hubička, A. Jaeger, M. Matz, and M. Mitchell, "System V Application Binary Interface, AMD64 Architecture Processor Supplement," July 2013.

[6] "LLVM 12 documentation: MCJIT design and implementaiton," https://www.llvm.org/docs/MCJITDesignAndImplementation.html, accessed 2020-09-04, Sep. 2020.

[7] DWARF Debugging Information Committee, "DWARF Debugging Information Format Version 5," Feb. 2017.

[8] "FreeBSD Manual Pages: CTF(5)," Sep. 2014.

[9] D. Hildenbrand, "An optimized intermediate representation for binary rewriting at runtime," Master's thesis, Technical University of Munich, Munich, Germany, 2019.

[10] D. R. Engler, W. C. Hsieh, and M. F. 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.

[11] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek, "C and tcc: a language and compiler for dynamic code generation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 2, pp. 324–369, 1999.

[12] F. Noel, L. Hornof, C. Consel, and J. L. Lawall, "Automatic, template-based run-time specialization: Implementation and experimental study," in *International Conference on Computer Languages (ICCL)*, 1998, pp. 132–142.

[13] C. Consel and F. 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.

[14] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. 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.

[15] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "DyC: An expressive annotation-directed dynamic compiler for C," *Theoretical Computer Science*, vol. 248, no. 1-2, pp. 147–199, 2000.

[16] H. Finkel, D. Poliakoff, J.-S. Camier, and D. 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.

[17] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *SIGPLAN conference on Programming language design and implementation (PLDI)*, 2000. [Online]. Available: https://www.complang.tuwien.ac.at/andi/bala.pdf

[18] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies, "Mojo: A dynamic optimization system," in *ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, 2000, pp. 81–90.

[19] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu, "Design and implementation of a lightweight dynamic optimization system," *Journal of Instruction-Level Parallelism*, vol. 6, no. 4, pp. 332–341, 2004.

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

[21] Avast Software, "Retdec," https://retdec.com/, accessed 2020-02-17.

[22] F. Cloutier, "fcd," http://zneak.github.io/fcd/, accessed 2020-02-17.