Optimizing Performance at Runtime Using Binary Rewriting

Alexis Engelke, David Hildenbrand, Martin Schulz
engelke@in.tum.de,hildenbr@in.tum.de,schulzm@in.tum.de
Department of Informatics, Technical University of Munich, Munich, Germany

1 INTRODUCTION AND MOTIVATION

In addition to scalability, performance of sequential code in applications is an important factor in HPC. Typically, programs are compiled once, at which time optimizations are applied, and are then run several times. However, not all information relevant for performance optimizations are available at compile-time and therefore restrict optimization possibilities. Such information may come from configuration options, results of previous computation, or even abstract programming models, where the exact data layout and distribution is defined at runtime. In such scenarios, the generation of specialized code at runtime allows for further optimizations based on additional runtime-only information.

Performing such specialization on binary code rather than regenerating machine code from a high-level language has several advantages: first, it is independent from the programming language or compiler infrastructure used and hence naturally also supports new languages. Second, also libraries without available source-code can be optimized as well. Finally, initial code can be generated at compile-time and only the relevant parts have to be rewritten at runtime, reducing the overhead of instruction selection at runtime. The latter is even further emphasized, if any kind of optimization is driven by the application, library or a runtime on itself, rather than being blindly applied.

2 APPLICATION-GUIDED BINARY REWRITING

To enable targeted optimizations and an effective use of known runtime information, the rewriting process needs to be guided by the application itself. This involves exploiting information only known to the developer (such as runtime constants) and avoids overhead through profiling or rewriting of code without beneficial optimizations. The rewriter is exposed as a library, where an application can configure information about constant memory regions or parameters. Any optimization is applied at function granularity as functions have clearly defined boundaries and their interface in the binary is well defined by the calling convention.

The next sections describe three different approaches we developed for runtime binary rewriting at different code abstraction levels, as illustrated in Figure 1, and we will discuss the respective pros and cons.

2.1 DBrew

DBrew [4] is a prototype of a tracing binary rewriter for the x86-64 architecture. The rewriter steps through the function instruction by instruction, keeping track of known values in a *known-world state* attached to each instruction. If all operands are known, the result is stored in the known-world state and the instruction is removed. Otherwise, an instruction is captured for emission in the rewritten

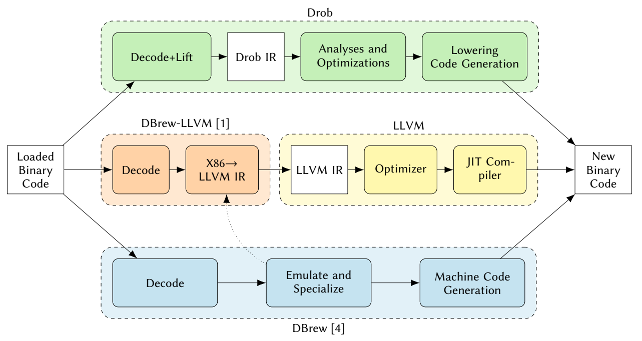


Figure 1: Overview of different approaches for runtime binary rewriting.

function, eventually replacing known operands with immediate encodings. If the next instruction has not been processed yet or was processed with a different known-world state, it is added to the queue for instructions to be handled. For conditional branches with an unknown decision, both possible targets are enqueued.

With this approach, DBrew efficiently implements three optimizations: first, all function calls are *inlined* if the target address is known. Second, the function is *partially evaluated* based on constant parameters and memory regions, removing instructions with known output values. Third, all loops with a known iteration range are *unrolled* completely, because the branch condition is always known and the loop counter is constant in the known-world state, causing the loop body to be replicated for each iteration.

However, due to the simple rewriting approach, DBrew misses several optimizations, for example, the removal of unused instructions, and due to aggressive inlining and loop unrolling, DBrew may generate a sizable amount of code.

2.2 Drob

Addressing the shortcomings of DBrew's strictly local transformations, Drob [2] decodes the whole function, reconstructing the control flow graph, and lifts the machine code to its intermediate representation—Drob IR; the IR consists of architecture-specific, abstracted machine instructions and architecture-specific registers. Afterwards, an optimization pass pipeline is run, performing and updating the analysis of the stack frame and the register usage when required.

Currently, Drob performs the following optimizations: simple loops are unrolled up to 10 times; constants are encoded as immediate operands where possible; instructions which are never executed or with unused results are removed; and the layout of the transformed code is improved to avoid branch instructions.

Drob does not implement advanced optimizations (e.g., expression folding and general loop transformations), yet; however, it provides a framework to implement such optimizations. The lifting to the IR and the lowering from the IR are efficient as information from the original binary code (e.g., register assignment, rough instruction selection) is maintained.

2.3 DBrew-LLVM

A different approach to perform runtime optimizations and code generation is to use an existing compiler infrastructure. Compilers include a wide variety of different optimizations and have highly tuned machine code generators. The LLVM [3] compiler infrastructure provides a highly configurable optimization pass pipeline operating on a target-independent intermediate representation (LLVM IR). Further, the framework also includes a JIT compiler, which generates machine code for execution in the same address space at runtime.

DBrew-LLVM [1] uses LLVM for runtime binary rewriting. This is achieved by decoding the whole function, reconstruction the control flow graph, and lifting the x86-64 instructions to the LLVM IR. As registers can contain different data types, the lifter models registers in different facets (e.g., a scalar float or a vector of integers for SSE registers) to avoid casts where possible. Special care also is required for pointer arithmetic, which is performed via integer arithmetic in hardware, but requires special semantics for effective optimizations in compilers. Specialization of parameters is realized by inlining the appropriate function call, leaving all constant folding and other code transformations to the optimizer.

Consequentially, even complex transformations such as vectorization, loop transformations, and expression optimization can be performed at runtime. However, an optimization which cannot be performed easily is the specialization of constant memory regions. Further, the optimization and code generation of LLVM is a time consuming process, even for simple functions. Moreover, parts of the work done by the compiler of the original code cannot be preserved during lifting, including register allocation, the stack frame, and the instruction selection, and must be redone during lowering.

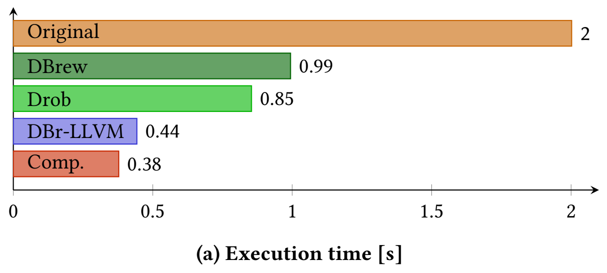
3 RESULTS

We compare the three described approaches to specialize a generic computation of a 2D stencil. In our kernel, the stencil is applied in a Jacobi-style iteration on a matrix. The data structure for the stencil is a list of triples containing an x-offset, a y-offset, and the weight factor. For our benchmarks, we use a simple 4-point stencil:

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

The benchmark runs this kernel with 20,000 iterations on a matrix of size 649×649. We compare the three transformations with the following two variants: first, the unmodified generic code (baseline); and second, a code where the stencil is known at compile-time and is manually specialized at compile-time (best case).

The rewriters and the benchmark were compiled with GCC 5.4.0 and run on a machine with two Intel Xeon Silver 4116 clocked at 2.1 GHz with 96 GiB memory running Ubuntu 16.04.6 with Linux kernel 4.4. We disabled AVX for the original benchmark code and the rewritten codes due to lacking support in the transformation.



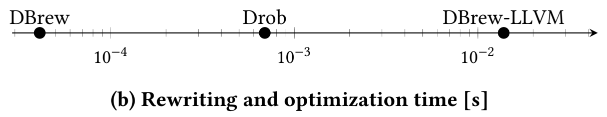


Figure 2: Performance results on the stencil benchmark, comparing the three runtime binary rewriting approaches with the original and a compile-time optimized code.

Results. The runtime and the time required for rewriting are shown in Figure 2. Compared to the original code, all three binary rewriting systems can achieve significant performance improvements. With the approach of analyzing and transforming the entire function, the performance of the code specialized by Drob is slightly faster than the code of DBrew. The LLVM-based rewriter achieved much more performant code with a better expression optimization and vectorization. The difference between the code optimized at compile-time to the known stencil is caused by a better vectorization produced by GCC compared to LLVM.

When taking the time required for analyzing, optimizing, and generating binary code into account, the cost of abstracted code representation and complex code transformations becomes visible: while DBrew can efficiently rewrite the function in $41\mu s$, Drob requires significantly more time ($0.7ms$) for the code analysis. DBrew-LLVM is another order of magnitude slower and requires $14ms$ for optimizations, spending most of the time analyzing/optimizing the code and generating new machine instructions.

4 CONCLUSIONS & FUTURE WORK

Runtime binary rewriting opens new optimization possibilities to specialize code for runtime-only information, as our rewriting frameworks demonstrate. However, we also expose that a higher level of abstraction allows for wider optimization at the cost of significantly higher rewriting time.

In the future, support for more complex architectural features, e.g., AVX and indirect jumps, can be added to support transformation of a wider range of codes. Also other architectures relevant for HPC, such as Power or ARM, can be targeted. Another opportunity to improve rewriting performance is to only analyze code that is likely to be executed. Finally, a hybrid rewriting system combining the three approaches can reduce the trade-off between rewriting time and code performance.

REFERENCES

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

[2] David Hildenbrand. 2019. *An Optimized Intermediate Representation for Binary Rewriting at Runtime.* Master's thesis. Technical University of Munich, Munich, Germany.

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

[4] Josef Weidendorfer and Jens Breitbart. 2016. 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)*. 376–385.