# Optimizing Performance at Runtime Using Binary Rewriting

**Alexis Engelke, David Hildenbrand, Martin Schulz**

Chair of Computer Architecture and Parallel Systems,
Department of Informatics, Technichal University of Munich, Germany
{engelke,hildenbr,schulzm}@in.tum.de

## Why Runtime Code Generation?

### Why at Runtime?

- Exploit information only available at runtime
  - E.g., configuration, previous computation results, data distribution, CPU features
- Reduce overhead caused by abstractions, for example, from higher-level programming models
- Optimize code in shared libraries for actual use

### Why from Binary Code?

- Independent of language and compiler
- Implies support for new frameworks/languages
- No extra language constructs or extensions needed
- Reuse work performed by the static compiler for faster rewriting
- Enables optimization of closed-source libraries

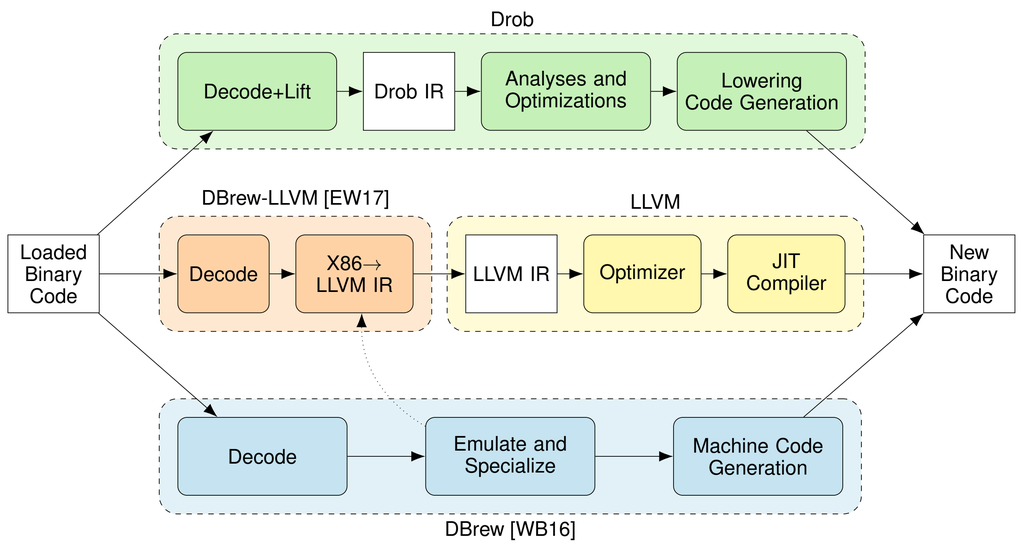
### Potential Applications

- Elimination of overhead of configuration checking in application kernels and libraries
- Specialization of MPI for data types, sizes, topology
- Optimization of performance abstraction libraries, e.g., Kokkos and Raja
- Abstract programming models, e.g., PGAS

## Contrasting Rewriting Approaches

### Application-guided Binary Rewriting

- Control rewriting from the application itself
  - Targeted optimizations: no profiling or unnecessary rewriting
  - Developers know optimization potential best
- Application specifies known values/params
- Rewriting of entire functions
  - Clearly defined boundaries and interface
- Easy replacement: No func. signature change
  - Only few code changes required
  - Allows safe fall-back to original code




### Library Usage

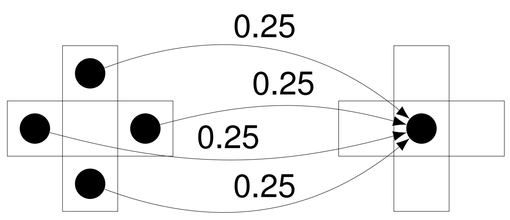
```
// 1. create new rewriter for fn
Rewriter* r = dbrew_new(fn);
// 2. configure rewriter
//  - set parameter as constant
dbrew_setpar(r, 0, 42);
//  - fix memory region
dbrew_setmem(r, ptr, len);
// 3. rewrite function
fn_t fn2 = (fn_t)dbrew_rewrite(r);
// 4. Use fn2 instead of fn
int res = fn2(42, ptr);
```

| | DBrew [WB16] | Drob | DBrew-LLVM [EW17] |
|---|---|---|---|
| **General Approach** | • Decode original machine code as needed (partial evaluation, emulate and capture)<br>• If all operands for an instruction are known, the instruction is emulated, otherwise captured and kept<br>• Remove jumps with known condition<br>• Duplicate block if known values differ | • Decode function and reconstruct CFG<br>• Lift instruction to Drob IR, keeping register and instruction information<br>• Analyze stack frame and register usage, tracking known constant values<br>• Run optimization passes, updating analysis data when needed | • Decode whole function and reconstruct CFG<br>• Lift individual instructions to LLVM-IR<br>• Constant arguments forwarded via inlining<br>• Run existing LLVM optimization passes and JIT compiler |
| **Code Representation** | Decoded x86 instructions | Abstracted x86 instructions | Architecture-independent LLVM-IR |
| **Currently Performed Optimizations** | Constant folding, dead branch elimination, full loop unrolling, aggressive inlining | Dead code elimination, constant folding, simple loop unrolling, instr. specialization | Loop transformations, vectorization, DCE, constant and expression folding, . . . |
| **Missing Optimizations** | Elimination of dead instructions, expression folding, limited loop unrolling, register re-allocation | Expression folding, general loop transformations, register re-allocation | Specialization for known memory regions |
| **Generated Code Example** | *Prologue*<br>`sub rsp,8`<br>`pxor xmm1,xmm1`<br>`pxor xmm1,xmm1`<br>*1st Stencil Point*<br>`mov rax,rcx`<br>`add rax,-1`<br>`movsd xmm0, [rsi+8*rax]`<br>`mulsd xmm0, [0x632048]`<br>`addsd xmm1,xmm0`<br>*2nd Stencil Point*<br>`mov rax,rcx`<br>`add rax,1`<br>`movsd xmm0, [rsi+8*rax]`<br>`mulsd xmm0, [0x632060]`<br>`addsd xmm1,xmm0`<br>. . . | *Prologue*<br>`pxor xmm1,xmm1`<br>*1st Stencil Point*<br>`mov rax,rcx`<br>`add rax,-1`<br>`movsd xmm0, [rsi+8*rax]`<br>`mulsd xmm0, [rip+0x7f4bb2cf6ff8]`<br>`addsd xmm1,xmm0`<br>*2nd Stencil Point*<br>`mov rax,rcx`<br>`add rax,1`<br>`movsd xmm0, [rsi+8*rax]`<br>`mulsd xmm0, [rip+0x7f4bb2cf6ff8]`<br>`addsd xmm1,xmm0`<br>. . . | *1st Stencil Point*<br>`movsd xmm0,[rsi+8*rcx+0x8]`<br>*2nd Stencil Point*<br>`addsd xmm0,[rsi+8*rcx-0x8]`<br>*3rd Stencil Point*<br>`addsd xmm0,[rsi+8*rcx-0x1448]`<br>*4th Stencil Point*<br>`addsd xmm0,[rsi+8*rcx+0x1448]`<br>*Multiplication*<br>`mulsd xmm0, [rip+0x7f53e498c000]`<br>`movsd [rdx+8*rcx],xmm0`<br>*Epilogue*<br>`ret` |

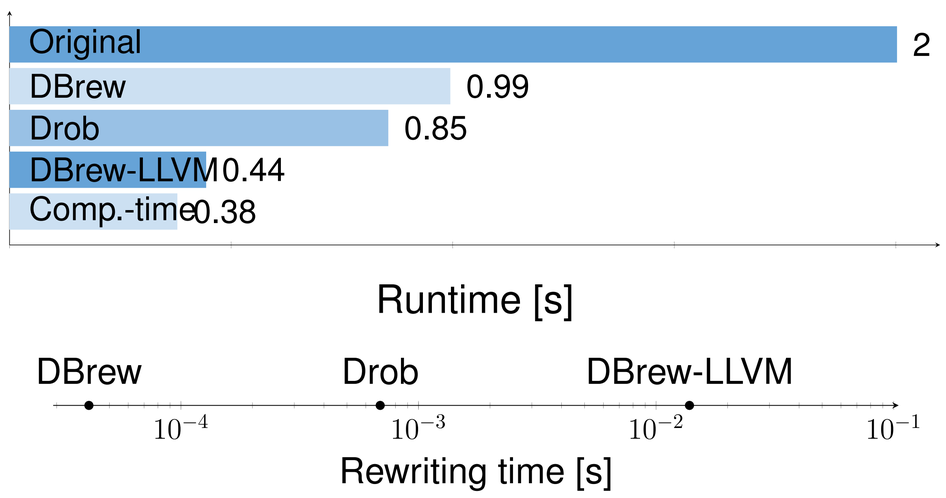
## Benchmark Results

- Benchmark: stencil given as flat structure

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




- Specialize for stencil only known at runtime
  - 20k Jacobi-style iterations on $649 \times 649$ matrix
- Measure rewriting and execution times
- Comparison: compile-time specialized code




Compiled with GCC 5.4.0; run on a machine with two Intel Xeon Silver 4116 @ 2.1 GHz, 96 GiB memory, Ubuntu 16.04.6, Linux kernel 4.4; DBrew-LLVM uses LLVM 6.

## Future Work

- Support for more architecture features: AVX, jump tables
- Other architectures: ARM, Power9, RISC-V
- API: automatic signature detection
- Lazy transformation/optimization
- More optimizations for DBrew/Drob: register renaming, expression optimization
- Hybrid rewriter combining the approaches
- Optimization parallel to execution

[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.

[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.

**SC’19**, International Conference for High Performance Computing, Networking, Storage and Analysis 2019, Denver, Colorado, USA