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
  – Only few code changes required
  – Developers know optimization potential best
• Targeted optimizations: no profiling or –
  – Only few code changes required
  – Developers know optimization potential best

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
int res = fn(r, 2); dbrew_rewrite(r);
// 4. Use fn2 instead of fn
int res = fn2(42, ptr);

DBrew [WB16]

General Approach
• Decode original machine code as needed
  (partial evaluation, emulate and capture)
• If all operands for an instruction are known, the instruction is emulated, otherwise captured and kept
• Remove jumps with known condition
• Duplicate block if known values differ

Code Representation
Decoded x86 instructions

Currently Performed Optimizations
Constant folding, dead branch elimination, full loop unrolling, aggressive inlining

Missing Optimizations
Elimination of dead instructions, expression folding, limited loop unrolling, register re-allocation

Generated Code Example

```
sub rbp, 8
push rsi, rax
push rdi, rdx
mov rax, rcx
add rax, -1
mov rmx, [rax]
```

Benchmark Results

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>DBrew</th>
<th>Drob</th>
<th>DBrew-LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd Stencil Point</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.99</td>
<td>0.85</td>
<td>0.44</td>
<td>0.44</td>
</tr>
</tbody>
</table>

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