Building a Fast Back-End for LLVM

Tobias Schwarz Alexis Engelke
{tobias.schwarz,engelke}@tum.de

Chair of Data Science and Engineering
Technical University of Munich

Eighth LLVM Performance Workshop at CGO
02.03.2024

Motivation

- For non-optimized codegen, compilation latency usually important (e.g. JIT, developer productivity)
- LLVM is widely used in compilers and can easily produce optimized code
- O0 back-end relatively slow

Can we do better?

Transforming LLVM-IR to Machine Code

General Strategy

- Build symbol table
- Generate data and relocations for globals and resolve aliases
- Per Function:
  - Instruction selection
  - Register/Stack allocation and PHI elimination
  - Unwind/Exception information
  - Relocations
- Either lay out in memory directly or
  generate object file with appropriate sections

Transforming LLVM-IR to Machine Code – Example

Globals

- Can be simple:

```
@.str.6 = private unnamed_addr
  constant [23 x i8] c"old␣output␣scalar␣file\00", align 1
```

- Or more difficult:

```
@_ZTI20cOutputScalarManager = linkonce_odr dso_local
  constant { ptr, ptr, ptr } {
      ptr getelementptr inbounds (
        ptr,
        ptr @_ZTVN10__cxxabiv120__si_class_type_infoE,
        i64 2),
      ptr @_ZTS20cOutputScalarManager,
      ptr @_ZTI7cObject
    }, comdat, align 8
```

- Need to generate appropriate section contents and relocations

Transforming LLVM-IR to Machine Code – Example

Function

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

```
define i32 @foo(int)(i32 noundef %n) {
  entry:
    %tobool.not6 = icmp eq i32 %n, 0
    br i1 %tobool.not6, label %while.end, label %while.body

  while.body:
    %res.08 = phi i32 [ %mul1, %while.body ], [ 1, %entry ]
    %n.addr.07 = phi i32 [ %sub, %while.body ], [ %n, %entry ]
    %mul = mul nsw i32 %n.addr.07, %n.addr.07
    %mul1 = mul nsw i32 %mul, %res.08
    %sub = add nsw i32 %n.addr.07, -1
    %tobool.not = icmp eq i32 %sub, 0
    br i1 %tobool.not, label %while.end, label %while.body

  while.end:
    %res.0.lcssa = phi i32 [ 1, %entry ], [ %mul1, %while.body ]
    ret i32 %res.0.lcssa
}
```

Transforming LLVM-IR to Machine Code – Example

Function

```
entry:
%0:gr32 = COPY $edi
%1:gr32 = MOV32ri 1
TEST32rr %0:gr32, %5:gr32
JE %while.end
JMP %while.body
```

```
while.body:
  %2:gr32 = PHI %1:gr32, %entry, %5:gr32, %while.body
  %3:gr32 = PHI %0:gr32, %entry, %6:gr32, %while.body
  %4:gr32 = IMUL32rr %3:gr32, %3:gr32
  %5:gr32 = IMUL32rr %4:gr32, %2:gr32
  %6:gr32 = DEC32r %3:gr32
  JNE %while.body
  JMP %while.end

while.end:
  %7:gr32 = PHI %1:gr32, %entry, %2:gr32, %while.body
  $eax = COPY %7:gr32
  RET
```

Transforming LLVM-IR to Machine Code – Example

Function

```
entry:
  liveins: $edi
  $eax = MOV32ri 1
  TEST32rr $edi, $edi
  JE %while.end
```

```
while.body:
  liveins: $eax, $edi
  $ecx = COPY $edi
  $ecx = IMUL32rr $ecx, $edi
  $eax = IMUL32rr $eax, $ecx
  $edi = DEC32r $edi
  JNE %while.body

while.end:
  liveins: $eax
  RET
```

Transforming LLVM-IR to Machine Code – Example

Function

```
0:
  PUSHr $rbp
  $rbp = COPY $rsp

entry:
  liveins: $edi
  $eax = MOV32ri 1
  TEST32rr $edi, $edi
  JE %while.end
```

```
while.body:
  liveins: $eax, $edi
  $ecx = COPY $edi
  $ecx = IMUL32rr $ecx, $edi
  $eax = IMUL32rr $eax, $ecx
  $edi = DEC32r $edi
  JNE %while.body

while.end:
  liveins: $eax
  $rsp = COPY $rbp
  POPr $rbp
  RET
```

How to Compile Faster

- IR gets rewritten several times $\rightsquigarrow$ performance cost
- Reduce number of passes over IR from > 60
- LLVM-IR is not very cache-friendly
  $\Rightarrow$ use optimized data structures
- Make strong assumptions about input IR
- For now, focus on `x86_64`

Assumptions About Input IR

- Roughly what clang produces at O0
- Regular Data Types
  - `i1`–`i64`, `i128`
  - `float`, `double`, no x86 `long double`
  - Currently, no vector types
- Aggregate types must fit in two registers, max two members
- No inline assembly
- No special/custom sections
- No thread-local storage
- No garbage collection support
- ...

Compiling Globals

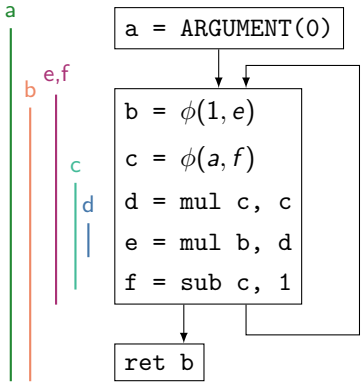
- Create symbols for globals, aliases and functions
- Globals with initializer are layed out sequentially depending on their attributes
  - Distinction between data, rodata, relro
  - special handling for `llvm.global_ctors`/`llvm.global_dtors`
  - Use of constant expressions currently very limited
  - No deduplication of equivalent globals
- Global aliases are resolved after all functions have been compiled

Preprocessing of LLVM-IR

- Assign consecutive ids to values and blocks for faster iteration
- Build lookup arrays to store auxiliary information during compilation
  - Instruction fused? Is value an argument?
  - Precompute storage size + register bank for type
  - Alloca frame offsets
  - Liveness information
  - Stack/Register Assignments
- Remove IR constructs which make compilation more difficult
  - Constant expressions/aggregates
  - Non-constant indices for GEPs, except the first

Analyzer

- Identify loops and place blocks accordingly
- Loops are basis for live intervals of values
- Additional refcounting for better liveness information inside of loops



Generate Globals → LLVM IR Pass → Analyzer

Machine-Code Generation

Overview

- Assume PIC and small code model (i.e. size of code+GOT+PLT $<$ 2GB)
- Prologue with placeholder for frame size and callee-saved register pushes and optional reg-save area for vararg functions
- Iterate over basic blocks and generate code directly into final code buffer
- Fusing is only done in forward direction
  - `icmp+br`
  - `gep+load/store`
  - `...`

Machine-Code Generation

Register Allocation

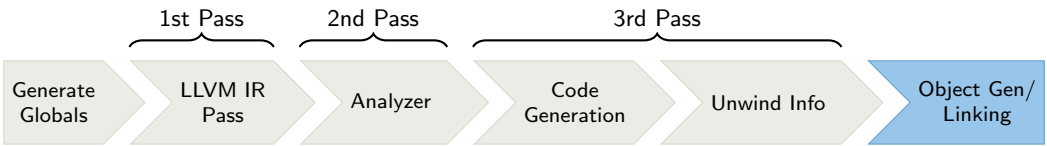
- Register Allocation is done together with instruction selection
- Greedy algorithm
- Assigns fixed registers to values with longer lifetimes
- Does not keep assignments saved for blocks with multiple outgoing edges
- PHI-Handling relatively standard:
  - Only one PHI in target: Move value to stack slot or into fixed register if there is any
  - For multiple PHIs:
    - Sort topologically
    - PHIs without a reader handled as above
    - Remaining ones must be part of a cycle → break with temporary register

Unwind/Exception Info

- Written out while compiling functions
- Frame pointer is always set up $\rightarrow$ only need to emit some info depending on which callee-saved registers were pushed
- For (C++) exceptions, the Language-Specific Data Area (LSDA) can't be linearily generated
  - $\rightsquigarrow$ Collect parts of the LSDA while compiling and build completed LSDA at end of every function

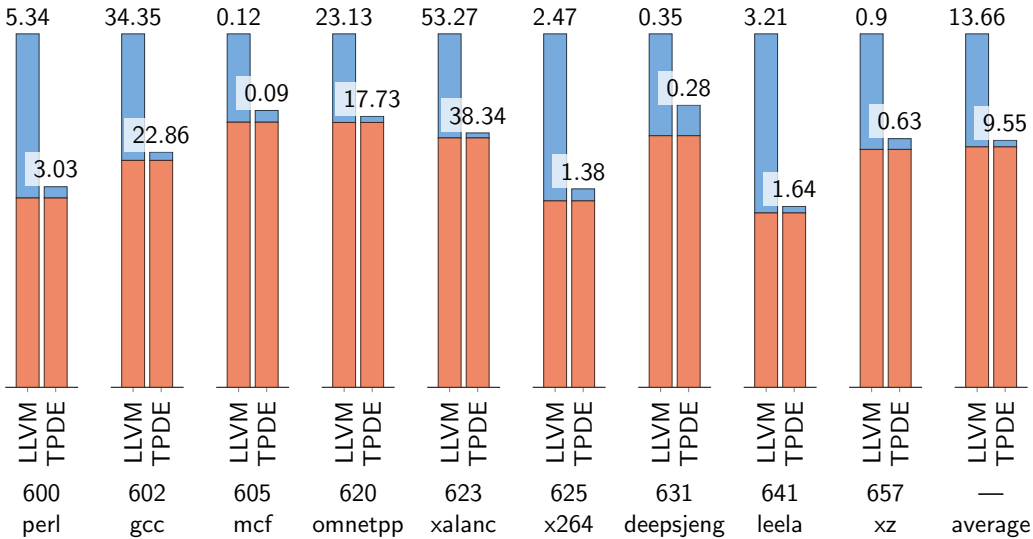
Object Generation

- Can generate ELF object
  $\Rightarrow$ Pass to linker or ORC pipeline
- Can also directly map into memory



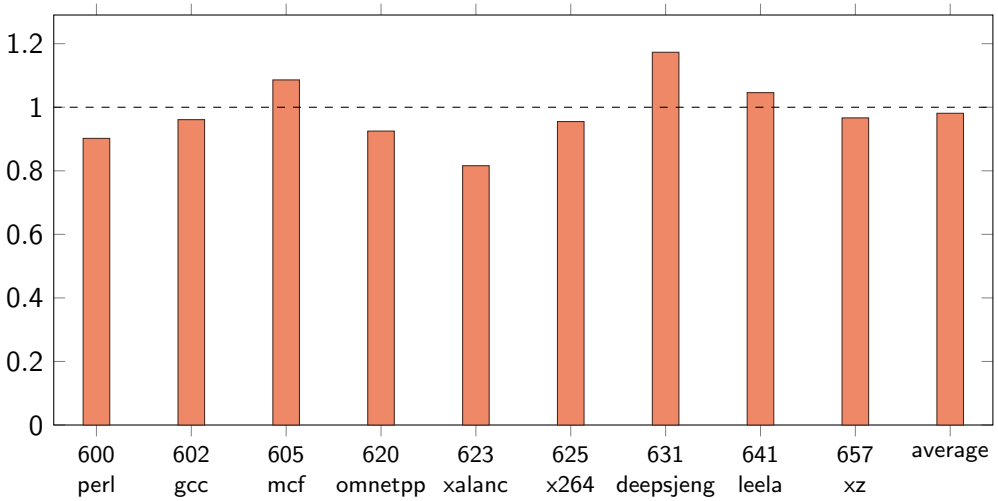
Performance Results – Compile Time

SPEC CPU 2017 – Compared To LLVM O0 [s] – Ryzen 7950X 64GB RAM



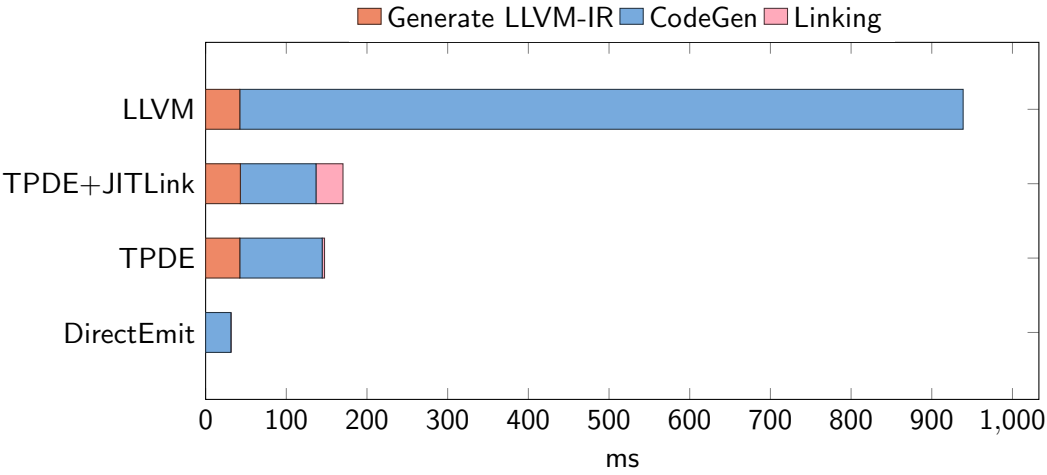
Performance Results – Run Time

SPEC CPU 2017 – Relative To Clang – Ryzen 7950X 64GB RAM



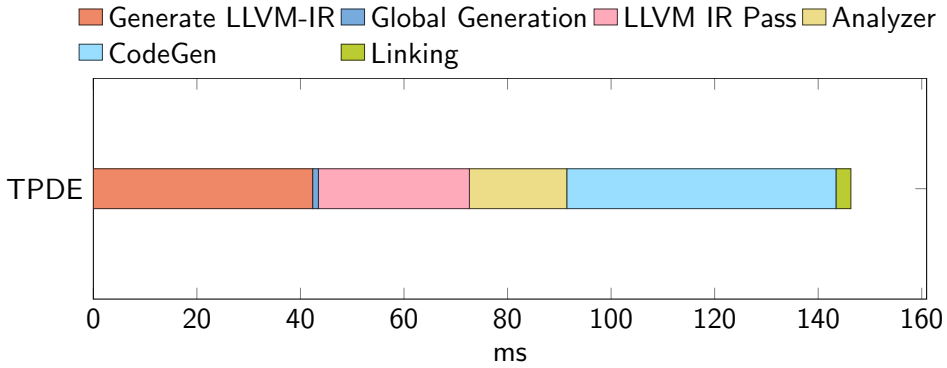
Performance Results

TPCDS SF10 in Umbra – Only Backend – Ryzen 7950X 64GB RAM



Performance Results

TPCDS SF10 – Looking at TPDE – Ryzen 7950X 64GB RAM



Difficulties in LLVM-IR

- IR not really suitable for fast iteration $\rightsquigarrow$ significant speedup achieved using the linearization
- Generating globals is not trivial, especially with constant expressions
- Constant expressions in general are not easy to deal with (e.g. in PHI nodes)
- Arbitrary integer widths are really not funny (they also show up in weird places, e.g. switches)
- In general, seems like IR allows a lot of semantics and each back-end supports some opaque subset
- Some information for compilation encoded as metadata strings (fcmps)

Conclusion

- TPDE compiles typical subset of LLVM-IR in just three passes
- 10-20x faster compilation than LLVM-O0
- Execution speed comparable to LLVM-O0
- Backend itself is fairly small ($\sim$ 15 kLOC)
- Easily plugged into existing users of the LLVM backend