LLVM for a Managed Language

What we've learned

Sanjoy Das, Philip Reames

`{sanjoy,preames}@azulsystems.com`

LLVM Developers Meeting
Oct 30, 2015

This presentation describes advanced development work at Azul Systems and is for informational purposes only. Any information presented here does not represent a commitment by Azul Systems to deliver any such material, code, or functionality in current or future Azul products.

Who are we?

Azul Systems

- We make scalable virtual machines
- Known for low latency, consistent execution, and large data set excellence

The Project Team

Bean Anderson

Philip Reames

Sanjoy Das

Chen Li

Igor Laevsky

Artur Pilipenko

What are we doing?

We're building a production quality JIT compiler for Java[1] based on LLVM.

[1]: Actually, for any language that compiles to Java bytecode

Design Constraints and Liberties

- Server workload, targeting peak throughput
- Compile time is less important
    - We already have a “Tier 1” JIT and an interpreter
- Small team, maintainability and debuggability are key concerns

An “in memory compiler”

- LLVM is not the JIT, it’s the optimizer, code generator, and dynamic loader
- The JIT magic’y stuff lives in the runtime
  - High quality profiling information already available
  - Has support for re-profiling and re-compiling methods
  - Has support for “deoptimization” (discussed later)
  - Same with compilation policy, code management, etc..

An existing runtime with a *flexible internal* ABI

(within reason and with cause)

Architectural Overview

- A “high level IR” embedded within LLVM IR
- Callbacks from mid level optimizer passes to the runtime
- Record and replay compiles outside of the VM

Embedding a high level IR

- Starting off, we have “high level” operations represented using calls to known abstraction functions

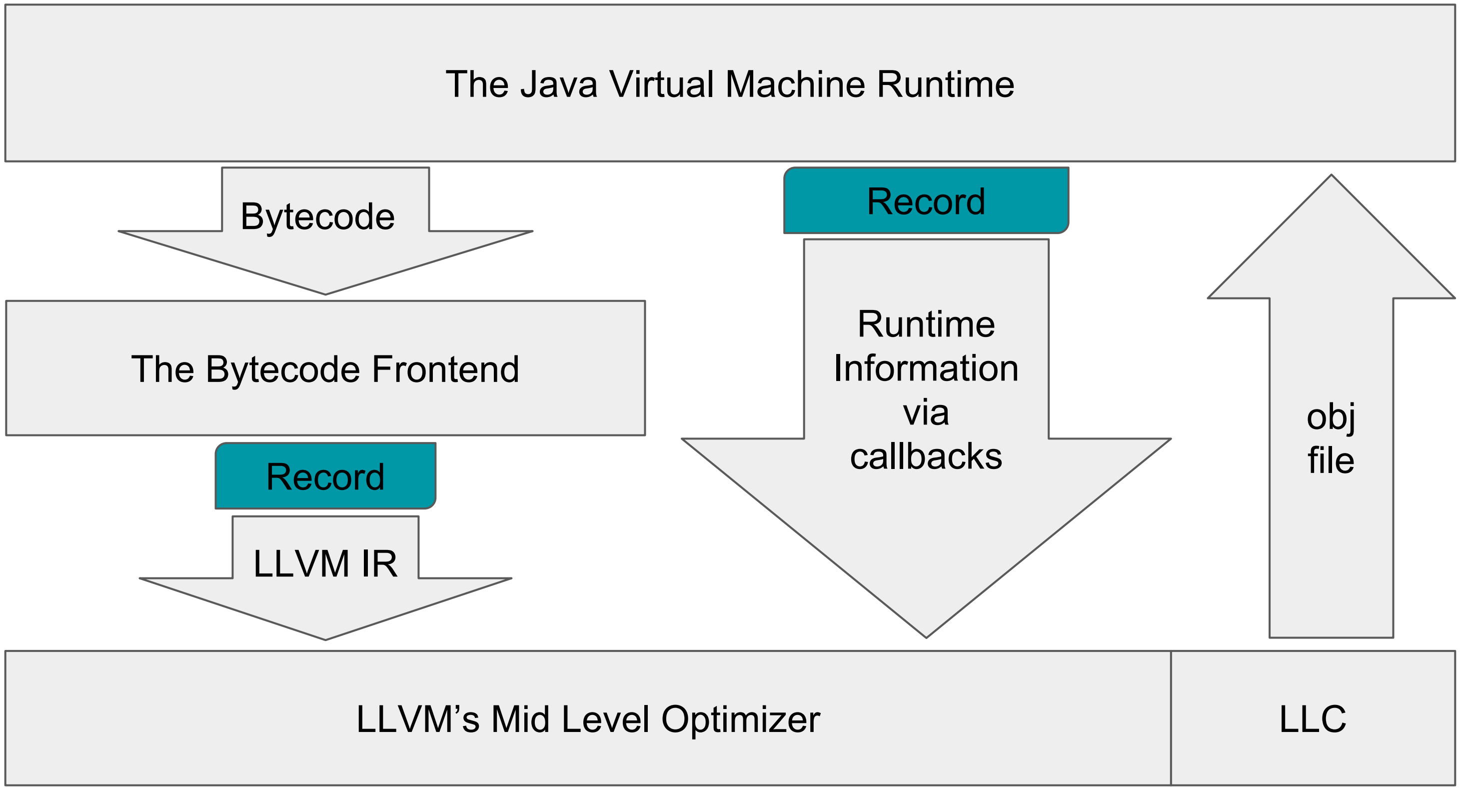
```
call void @azul.lock(i8 addrspace(1)* %obj)
```

- Most of the frontend lowers directly to normal IR
- Abstraction inlining events form the boundaries of each optimization phase

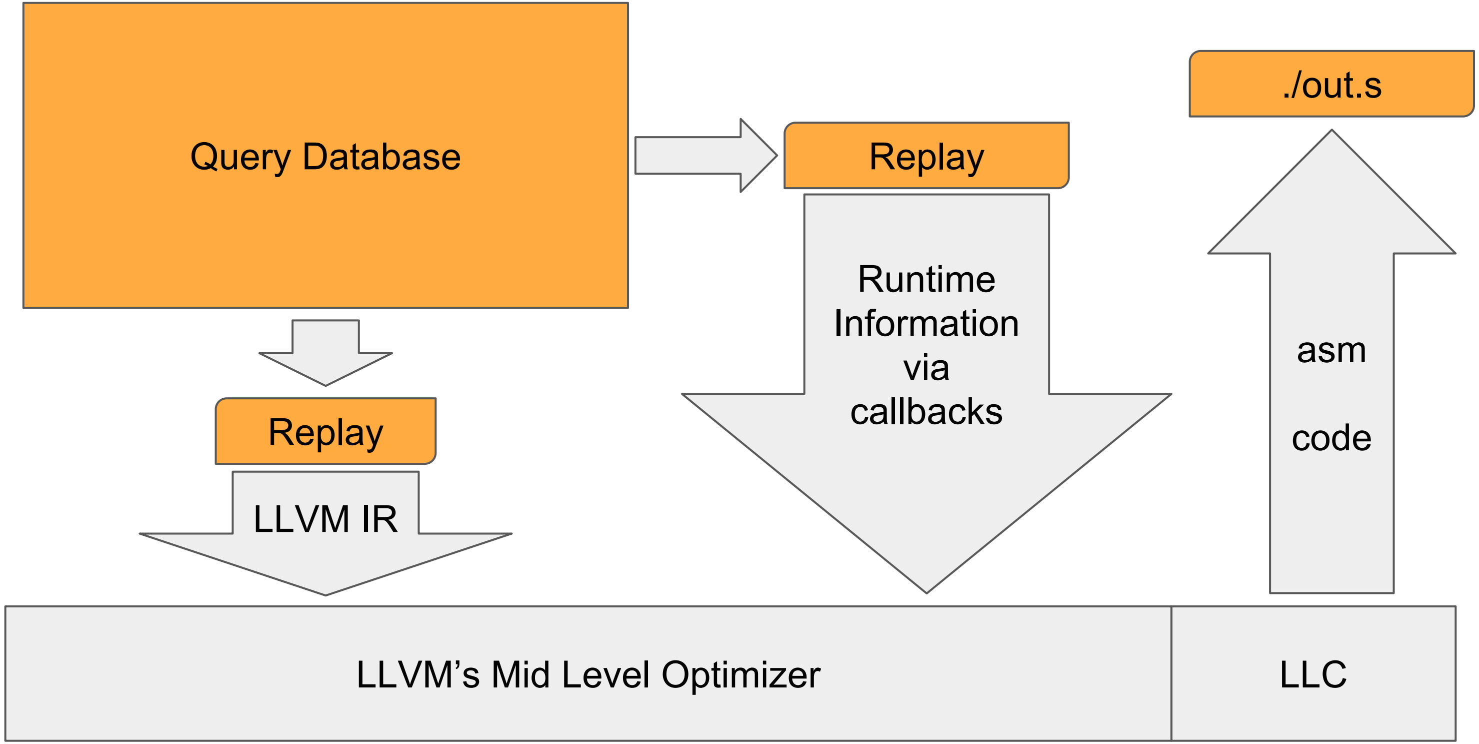
Why an embedded HIR?

- We didn't really want to write another optimizer
- A split optimizer seemed likely to suffer from pass ordering problems.
  - So does an embedded one, but at least it's easier to change your mind

Over time, we've migrated to eagerly lowering more and more pieces.



Architecture (artistic rendition)



Architecture (artistic rendition)

Code Management

- Generate and relocate object file in memory
- Most data sections are not relocated into permanent storage
  - Notable exception: `.rodata*`
  - Data sections like `.eh_frame`, `.gcc_except_table`, `.llvm_stackmaps` are parsed and discarded immediately after
- Runtime expects to patch code (patchable calls, inline call caches)

Optimizing Java

Java is not C

- All memory accesses are checked
  - Null checks, range checks, array store checks
  - Pointers are well behaved
- No undefined behavior to “exploit”
- Data passed by reference, not value
- `s.m.Unsafe` implies we’re compiling both C and Java at the same time

```
int sum_it(MyVector v, int len) {
  int sum = 0;
  for (int i = 0; i < len; i++)
    sum += v.a[i];
  return sum;
}
```

```
if (v == null) {
  throw new NullPointerException();
}
a = v.a;
if (a == null) {
  throw new NullPointerException();
}
if (i < 0 || i > a.length) {
  throw new IndexOutOfBoundsException();
}
sum += a[i]
```

Very few custom passes needed

Focus on improving existing passes

- lots of small changes
- mostly around canonicalization

Speculative Optimization

- Overly aggressive, “wrong” optimizations:
    - Speculatively prune edges in the CFG
    - Speculatively assume invariants that may not hold forever
    - Often better to “ask for forgiveness” than to “ask for permission”
- Need a mechanism to fix up our mistakes ...

AZUL SYSTEMS®

```
int f() {
  // No subclass of A overrides foo
  return this.a.foo()
}
```

```
int f() {
  return A::foo(this.a);
}
```

```
void f() {
  this.a.foo();
  this.a.foo();
}
```

A new class B is loaded here, which subclasses A and implements foo

Might now be an instance of B

Any call can invalidate speculative assumptions in the caller frame

`invoke @A::foo()`

(Abstract VM State)

Interpreter @ invokevirtual a.foo()

Normal Return Path

Exception Flow

The runtime ensures we "return to" the right continuation.

Speculative Optimization: Deoptimizing

- Deoptimize(verb): replace my (physical) frame with N interpreter frames, where N is the number of abstract frames inlined at this point
- We can construct interpreter frames from abstract machine state
- Abstract Machine State:
    - The local state of the executing thread (locals, stack slots, lock stack)
        - May contain runtime values (e.g. my 3rd local is in %rbx)
    - Writes to the heap, and other side effects

AZUL SYSTEMS®

Deoptimization: What the Runtime Needs

- The runtime needs to map the N interpreted frames to the compiled frame
- The frontend needs to emit this “map”, and LLVM needs to preserve it
- This map is only needed at call sites
- Call sites also need to be something like “sequence points”

Deoptimization State: Codegen / Lowering

Four step process

1. `(deopt args)` = encode abstract state at call
2. Wrap call in a `statepoint`, `stackmap` or `patchpoint`
    a. Warning: subtle differences between live through vs. live in
3. Run “normal” code generation
4. Read out the locations holding the abstract state from `.llvm_stackmaps`

Deoptimization State: Early Representation

- We need a representation for the mid-level optimizer
- `statepoint`, `patchpoint` or `stackmap` are not ideal for mid level optimizations (especially inlining)
- Solution: operand bundles

Deoptimization State: Operand Bundles

- “deopt” operand bundles (in progress, still very experimental)
  - `call void @f(i32 %arg) [ “deopt”(i32 0, i8* %a, i32* null) ]`
  - Lowered via `gc.statepoint` currently; other lowerings possible
- Operand bundles are more general than “deopt”
  - `call void @g(i32 %arg) [ “tag-a”(i32 0, i32 %t), “tag-b”(i32 %m) ]`
  - Useful for things other than deoptimization: value injection, frame introspection

Specific Improvements

Implicit Null Checks

- Despite best efforts (e.g. loop unswitching, GVN), some null checks remain
    - `obj.field.subField++`
- Standard Solution: issue an unchecked load, and handle the `SIGSEGV`
- Works because in practice `NullPointerExceptions` are very rare

Implicit Null Checks

```
 testq   %rdi, %rdi
 je      is_null
 movl    32(%rdi), %eax
 retq
is_null:
 movl    $42, %eax
 retq
```

Legality: the load faults if and only if `%rdi` is zero

```
load_inst:
 movl    32(%rdi), %eax
 retq
is_null:
 movl    $42, %eax
 retq
```

SIGSEGV

Implicit Null Checks

- `.llvm_faultmaps` maps faulting PC's to handler PCs
- Inherently a profile guided optimization
- Possible to extend this to checking for division by zero
- In LLVM today for x86, see `llc -enable-implicit-null-checks`

Optimizing Range Checks

- We’ve made (and are still making) `ScalarEvolution` smarter
- `-indvars` has been sufficient so far, no separate range check elision pass
- Java has well defined integer overflow, so SCEV needs to be even smarter

SCEV'isms: Exploiting Monotonicity

$$
\begin{array}{l}
\texttt{for (i = M; i <}_s\texttt{ N; i++)} \\
\texttt{\{} \\
\quad \texttt{if (i <}_s\texttt{ 0) return;} \\
\quad \texttt{a[i] = 0;} \\
\texttt{\}}
\end{array}
$$

$$
\begin{array}{l}
\texttt{for (i = M; i <}_s\texttt{ N; i++}_{\texttt{nsw}}\texttt{)} \\
\texttt{\{} \\
\quad \texttt{if (M <}_s\texttt{ 0) return;} \\
\quad \texttt{a[i] = 0;} \\
\texttt{\}}
\end{array}
$$

The range check can fail *only* on the first iteration. $\texttt{i} <_s 0 \Leftrightarrow \texttt{M} <_s 0$

SCEV'isms: Correlated IVs

```
j = 0
for (i = L-1; i >=s 0; i--)
{
  if (!(j <u L)) throw();
  a[j++] = 0;
}
```

```
j = 0
for (i = L-1; i >=s 0; i--)
{
  if (!(true)) throw();
  a[j++] = 0;
}  // backedge taken L-1 times
```

SCEV'isms: Multiple Preconditions

```
if (!(k <u L)) return;
for (int i = 0; i <u k; i++)
{
  if (!(i <u L)) throw();
  a[i] = 0;
}
```

Today this range check does not optimize away.

Partially Eliding Range Checks: IRCE

```
for (i = 0; i <_s n; i++) {
  if (i <_u a.length)
    a[i] = 42;
  else throw();
}
```

```
t = smin(n, a.length)
for (i = 0; i <_s t; i++)
  a[i] = 42;  // unchecked
for (i = t; i <_s n; i++) {
  if (i <_u a.length)
    a[i] = 42;
  else throw();
}
```

Dereferenceability

```
  if (arr == null) return;
loop:
  if (*condition) {
    t = arr->length;
    x += t
  }
```

```
  if (arr == null) return;
  t = arr->length;
loop:
  if (*condition)
    x += t
```

Subject to aliasing, of course.

Dereferenceability

- Dereferenceability in Java has well-behaved control dependence
  - Non-null references are dereferenceable in their first `N` bytes (`N` is a function of the type)
  - We introduced `dereferenceable_or_null(N)` specify this
- Open Question: Arrays?
  - `dereferenceable_or_null(<runtime value>)` ?

Aliasing

- We haven’t needed a language specific AA implementation yet; we use TBAA and struct TBAA to convey basic facts
- Fairly coarse so far; not heavily leveraging the Java type system
- We generalized `argmemonly` to non-intrinsics
    - Really helpful for high level abstractions

Constant Memory

- We use `invariant.load` for:
  - VM level final fields (e.g. length of an array)
  - Java level final fields (`static final`) of heap reference type
    - Primitive `static final`s can be directly constant folded
    - Instance finals are a bit tricky (forthcoming)

Constant Memory: Open problems

- Memory which “becomes constant”
  - Inlining allocation functions and `invariant.load`
  - `final` instance fields in Java
- Subtly different (?) representations for the same thing
  - The backend’s notion of `invariant.load` is different than the IR’s
  - TBAA’s notion of `isConstant` vs. `invariant.load`

Takeaways

- Embedded high level IR enables rapid development
- New support for operand bundles (i.e. deoptimization, frame introspection, frame interjection)
- Canonicalization required for effective optimization; per language work needed
- LLVM powerful building block for debuggable managed language compiler

Questions?