

FALCON: AN OPTIMIZING JAVA JIT

Philip Reames

Azul Systems

AGENDA

- Intro to Falcon
- Why you should use LLVM to build a JIT
- Common Objections (and why they're mostly wrong)

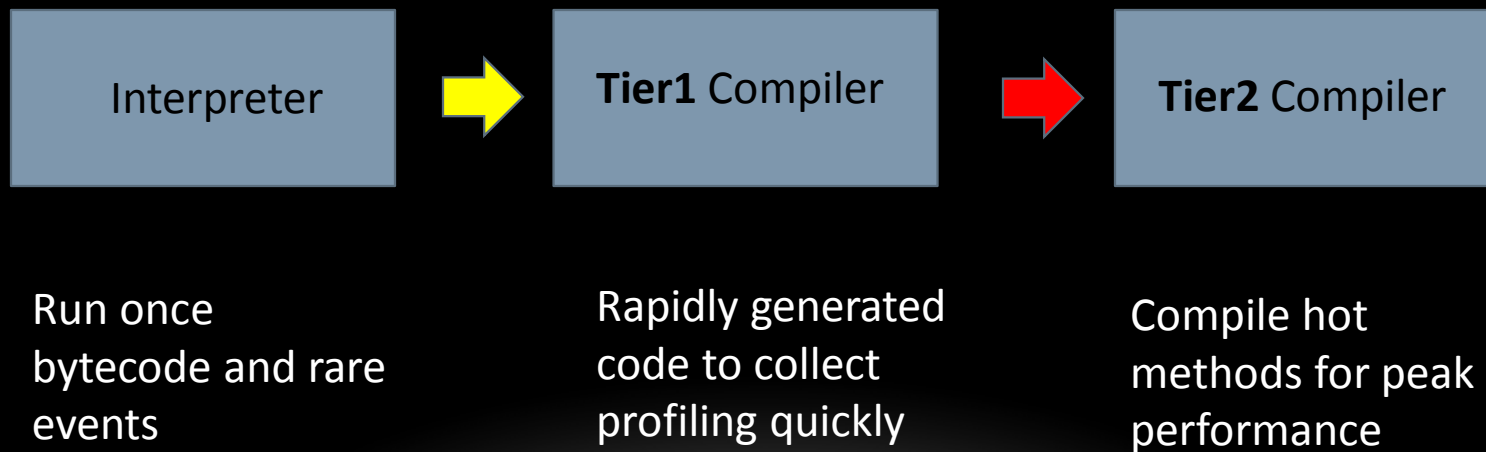
WHAT IS FALCON?

- Falcon is an LLVM based just-in-time compiler for Java bytecode.
- Shipping on-by-default in the Azul Zing JVM.
- Available for trial download at: www.azul.com/zingtrial/

THIS TALK IS ABOUT LESSONS LEARNED, BOTH
TECHNICAL AND PROCESS

ZING VM BACKGROUND

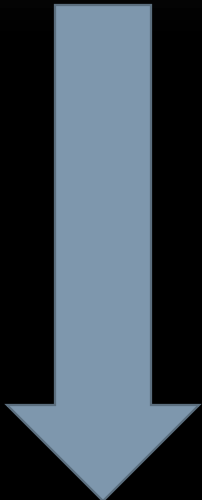
- A hotspot derived JVM with an awesome GC (off topic)
- Multiple tiers of execution



BUSINESS NEED

- The existing C2 compiler is aging poorly
 - vectorization (a key feature of modern x86_64) is an afterthought
 - very complicated codebase; "unpleasant" bug tails the norm
 - difficult to test in isolation
- Looking to establish a competitive advantage
 - Long term goal is to outperform competition
 - Velocity of performance improvement is key

DEVELOPMENT TIMELINE



April 2014	Proof of concept completed (in six months)
Feb 2015	Mostly functionally complete
April 2016	Alpha builds shared with selected customers
Dec 2016	Product GA (off by default)
April 2017	On by default

Team size: 4-6 developers, ~20 person years invested

TEAM EFFORT

Falcon Development

Bean Anderson
Chen Li
Igor Laevsky
Daniel Neilson
Serguei Katkov
Daniil Suchkov
Leela Venati
Nina Rinskaya

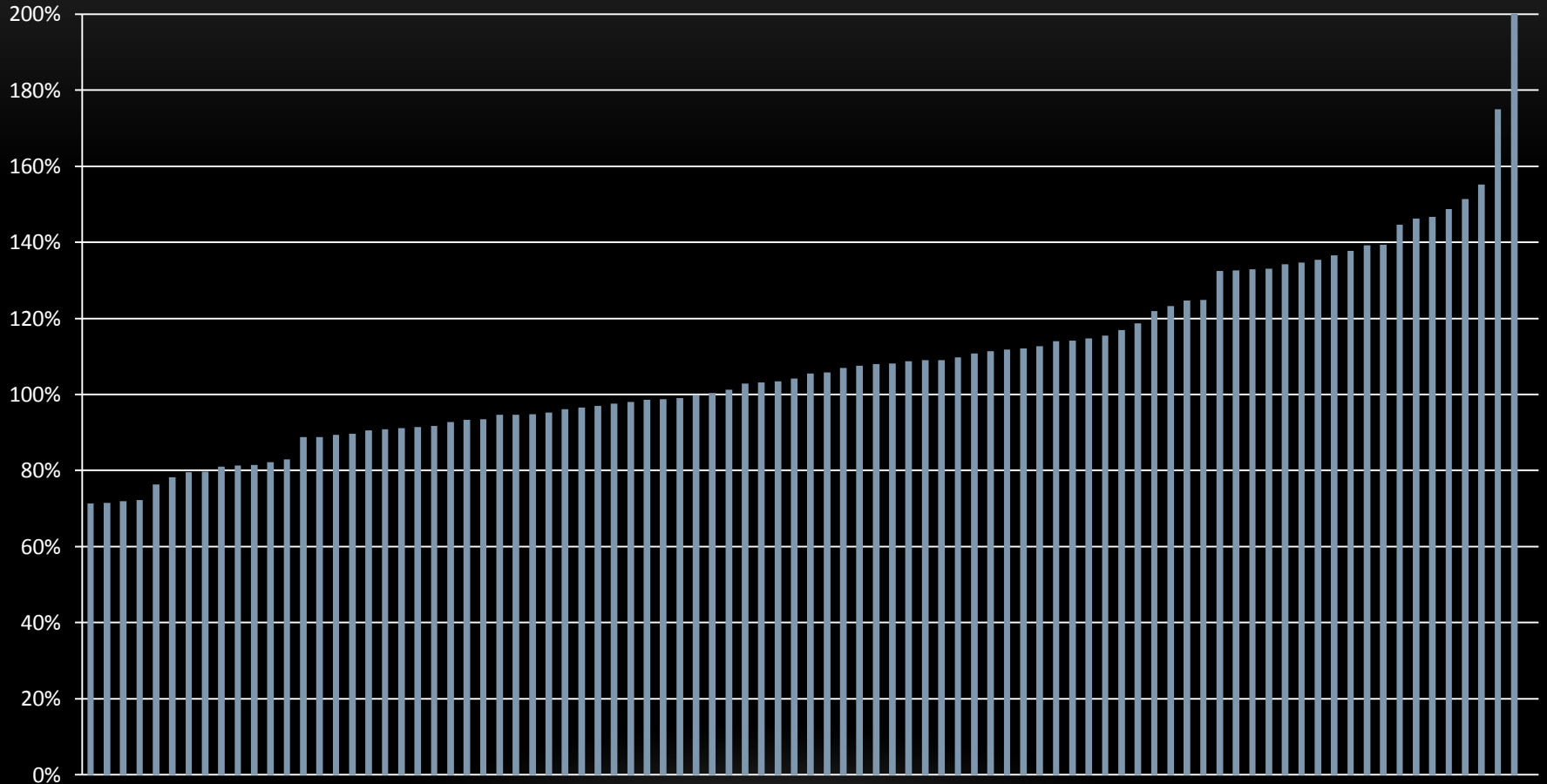
Philip Reames
Sanjoy Das
Artur Pilipenko
Anna Thomas
Maxim Kazantsev
Michael Wolf
Kris Mok

+ VM development team

+ Zing QA

+ All Azul (E-Staff, Sales, Support, etc..)

Zing 17.08 vs Oracle 8u121



Various application benchmarks + SPECjvm + Dacapo
Collected on a mix of haswell and skylake machines

WHY YOU SHOULD USE LLVM TO BUILD A JIT

- Proven stability, widespread deployments
- Active developer community, support for new micro-architectures
- Proven performance (for C/C++)
- Welcoming to commercial projects

COMMON OBJECTIONS

- "LLVM doesn't support X"
- "LLVM is a huge dependency"
- "We added an LLVM backend; it produced poor code"
- "LLVM generates too much code"
- "LLVM is a slow JIT"
- "My language has feature X (which requires a custom compiler)"

Objection 1 of 6

LLVM DOESN'T SUPPORT X

FIRST, A BIT OF SKEPTICISM...

- Is this something you can express in C? If so, LLVM supports it.
e.g. deoptimization via spill to captured on-stack buffer

```
buf = alloca(...);  
buf[0] = local_0;  
...  
a->foo(buf, actual_args...)
```

The real question, is *how well* is it supported?

FUNCTIONAL CORNER-CASES

- If you can modify your ABI (calling conventions, patching sequences, etc.), your life will be *much* easier.
- You will find a couple of important hard cases. Ours were:
 - “anchoring” for mixed stack walks
 - red-zone arguments to assembler routines
 - support for deoptimization both checked and async
 - GC interop

BE WARY OF OVER DESIGN

- Common knowledge that quality of safepoint lowering matters.
- gc.statepoint design
 - Major goal: allow in register updates
 - Topic of 2014 LLVM Dev talk
 - 2+ person years of effort
- It turns out that hot safepoints are inliner bug.

GET TO FUNCTIONAL CORRECTNESS FIRST

- Priority 1: Implement all interesting cases, get real code running
- Priority 2: Add tests to show it continues working
- Design against an adversarial optimizer.
- Be wary of over-design, while maintaining code quality standards.

See backup slides for more specifics on this topic.

Objection 2 of 6

LLVM IS A HUGE DEPENDENCY

- Potentially a real issue, but depends on your definition of large.
- From our shipping product:
 - libJVM = ~200mb
 - libLLVM = ~40mb
- So, 20% code size increase.

Objection 3 of 6

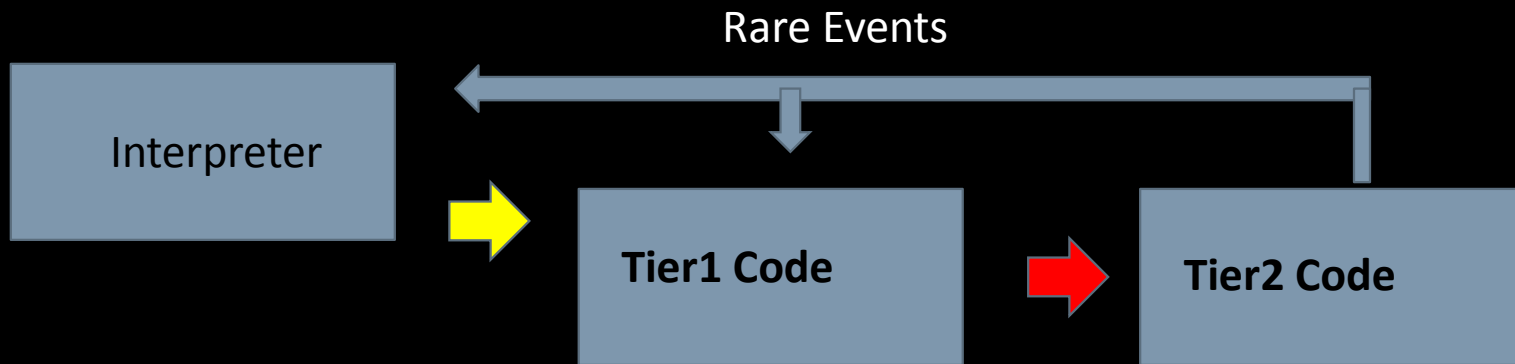
WE ADDED AN LLVM BACKEND;
IT PRODUCED POOR CODE

IMPORTANCE OF PROFILING

- Tier 1 collects detailed profiles; Tier 2 exploits them
- 25% *or more* of peak application performance

PRUNE UNTAKEN PATHS


- Handle rare events by returning to lower tier, reprofiling, and then recompiling.



```
%ret = call i32 @llvm.experimental.deoptimize() ["deopt"(..)]  
ret i32 %ret
```

PREDICATED DEVIRTUALIZATION

```
switch (type(o))
case A: A::foo();
case B: B::foo();
default: o->foo();
```



```
switch (type(o))
case A: A::foo();
case B: B::foo();
default: @deoptimize() ["deopt"(...)]
```

Critical for Java, where everything is virtual by default

IMPLICIT NULL CHECKS

```
%is.null = icmp eq i8* %p, null  
br i1 %is.null, label %handler, label %fallthrough, !make.implicit !{}
```

```
test rax, rax  
jz <handler>  
rsi = ld [rax+8]
```



```
rsi = ld [rax+8] ← fault_pc
```

```
__llvm_faultmaps[fault_pc] -> handler
```

```
handler:  
  call @__llvm_deoptimize
```

LOCAL CODE LAYOUT

- branch_weights for code layout
- Sources of slow paths:
 - GC barriers, safepoints
 - handlers for builtin exceptions
 - result of code versioning
- 50%+ of total code size

Hot		hot
cold		hot
hot	→	hot
cold		cold
hot		cold
cold		cold

GLOBAL CODE LAYOUT

- Best to put cold code into it's own section

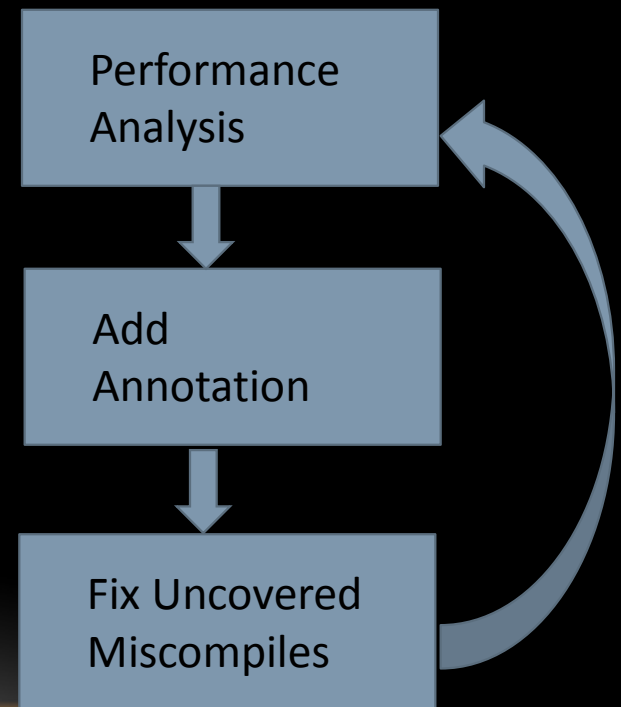
func1-hot		func1-hot
func1-cold		func2-hot
func2-hot	→	func3-hot
func2-cold		func1-cold
func3-hot		func2-cold
func3-cold		func3-cold

See:

LLVM back end for HHVM/PHP,
LLVM Dev Conf 2015

EXPLOITING SEMANTICS

- LLVM supports a huge space of optional annotations
- Both metadata and attributes
- ~6-12 month effort



DEFINING A CUSTOM PASS ORDER

```
legacy::PassManager PM;  
PM.add(createEarlyCSEPass());  
...  
PM.run(Module)
```

- Requires careful thought and experimentation.
 - MCJIT's OptLevel is not what you want.
 - PassManagerBuilder is tuned for C/C++!
- May expose some pass ordering specific bugs

EXPECT TO BECOME AN LLVM DEVELOPER

- You will uncover bugs, both performance and correctness
- *You* will need to fix them.
- Will need a downstream process for incorporating and shipping fixes.

See backup slides for more specifics on this topic.

STATUS CHECK

- We've got a reasonably good compiler for a c-like subset of our source language.
- We're packaging a modified LLVM library.
- This is further than most projects get.

QUICK EXAMPLE

```
public int intsMin() {  
    int min = Integer.MAX_VALUE;  
    for (int i = 0; i < a_I.length; i++) {  
        min = min > a_I[i] ? a_I[i] : min;  
    }  
    return min;  
}
```

4x faster than competition on Intel Skylake

Objection 4 of 6

"LLVM GENERATES TOO MUCH CODE"

CALLING ALL LLVM DEVELOPERS...

- We regularly see 3-5x larger code compared to C2.
- Partly as a result of aggressive code versioning.
- But also, failure to exploit:
 - mix of hot and cold code, need selective Oz
 - lots and lots of no-return paths
 - gc.statepoint lowering vs reg-alloc
 - code versioning via deopt (unswitch, unroll, vectorizer)

Objection 5 of 6

LLVM IS A SLOW JIT

- Absolutely. LLVM is *not* suitable for a *first tier* JIT.
- That's not what we have or need.
- We use the term "in memory compiler" to avoid confusion.

SYSTEM INTEGRATION

- VM infrastructure moderates impact
 - The tier 2 compiler sees a small fraction of the code.
 - Background compilation on multiple compiler threads.
 - Prioritized queueing of compilation work
 - Hotness driven fill-in of callee methods
- This stuff is standard (for JVMs). Nothing new here.

IN PRACTICE, MOSTLY IGNORE-ABLE

- Typical compile times around 100ms.
- Extreme cases in the seconds to low minute range.
- At the extreme, that's (serious) wasted CPU time, but nothing else.

WAIT, YOU DO *WHAT*?

- There are cases where this isn't good enough.
 - a popular big-data analytics framework spawns a JVM per query
 - multi-tenancy environments can spawn 1000s of JVMs at once
- Improving compile time is hard. So what do we do?

CACHED COMPILES

- Reuse compiled code across runs.
 - Profiling continues to apply.
 - No decrease in peak performance.
- Planned to ship in a future version of Zing.

After all, our "in memory compiler" does produce normal object files.

Credit for inspiration goes to the Pyston project and their article "Caching object code". <https://blog.pyston.org/2015/07/14/caching-object-code>

Objection 6 of 6

MY LANGUAGE HAS FEATURE X
(WHICH REQUIRES A CUSTOM COMPILER)

LANGUAGE SPECIFIC DEFICIENCIES

- LLVM has been tuned for certain languages
- The more different your language, the more work needed.
- For Java, our key performance problems were:
 - range checks
 - null checks
 - devirtualization & inlining
 - type based optimizations
 - deoptimization

DO YOU ACTUALLY HAVE A PROBLEM?

- Try the naive version, seriously try it!
- Very useful to try different C++ implementations
 - requires good knowledge of Clang
 - may find a viable implementation strategy
 - may provide insight into what optimizer finds hard
- Check to see if existing metadata/attributes are relevant

CUSTOM ATTRIBUTES/METADATA

- Is there a single key missing fact? Or a small set thereof?
- A lot of advantages:
 - Factoring for long term branching
 - May be upstreamable
 - Easy to functionally test
 - Serialization

Examples from our tree:

```
LICM isGuaranteedToExecute  
"allocation-site"  
"deopt-on-throw"  
"known-value"  
"java-type"="XYZ"
```

METADATA HEALING

- Optimizer tends to strip metadata it doesn't understand.
- **Don't try to change this**
- Need a mechanism to "heal" previously stripped metadata

See backup slides for more on this topic

CALLBACKS

Narrow interface between optimizer and containing runtime

```
Optional<FieldInfo>
```

```
getKnownValueKeyDependent(LLVMContext &C,  
                           KnownValueID BaseObj,  
                           int64_t Offset,  
                           uint64_t Size);
```

CALLBACKS

- Primarily used for metadata healing
 - But also useful for type system specific optimizations
- Resist temptation to add them freely.
 - Each one should be strongly and continuously justified.
 - We have 24 total. And that's too many.
- Optimizer can visit dead code (Critically Important)
 - Fuzz this interface. Seriously, fuzz it!

IMPLICATIONS FOR REPLAY

Replay is an absolutely essential debugging and diagnostic tool

```
$ opt -O3 -S input.ll
```

To preserve replay w/callbacks, need a query-log mechanism.

```
$ opt -O3 -S input.ll --read-query-log-from-file=input.ll.queries
```

EMBEDDED HIGH LEVEL IR

- "Abstraction" functions
 - Opaque by default to the optimizer
 - Can be first class citizens with builtin knowledge
 - Useful for building certain classes of high level optimizations

```
declare i8 addrspace(1)*  
@azul.resolve_virtual(i8 addrspace(1)* %receiver,  
                      i64 %vtable_index)  
nounwind noinline readonly "late-inline"="2"  
"gc-leaf-function"
```

EMBEDDED HIGH LEVEL IR

- Enable high level transforms such as:
 - devirtualization
 - object elision
 - allocation sinking & partial escape analysis
 - lock coarsening & elision
- Most of the above expressed as custom passes

See backup slides for discussion of the tradeoffs around custom passes

A (SLIGHT) CHANGE IN VIEW ON ABSTRACTIONS

- Useful for rapid evolution and prototyping
- But each is an optimization barrier until lowered
 - As few as you can reasonably manage long term
 - Lower early in the pipeline
 - Minimize number of unique lowering points

CONCLUSION

For the right use case, LLVM can be used to build a production quality JIT which is performance competitive with state of the art.

QUESTIONS?

BACKUP SLIDES

ASIDE: STAFFING

- Hiring a team from within the existing community is hard, slow, and expensive.
- Expect to train most of your team; including yourself.

SO IT'S COMPLETE, IS IT CORRECT?

- Design against a adversarial optimizer.
- Don't fall for "no real compiler would do that".
- You'll be wrong; if not now, later.

A RECOMMENDATION

Implement the cases claimed to be unreasonable. :)

HIGH VALUE TESTING TECHNIQUES

- Compile everything (not standard for high tier compilers) in a very large corpus w/a Release+Asserts build
 - 1500+ JARs, every method, every class
 - Finds many subtle invariant violations
- Randomly generated test cases (Fuzzing) works.
 - By far cheapest way to find miscompile bugs.
 - **Good way to find regressions quickly.**

“Be Wary of Overdesign” continued...

ONE WORD OF CAUTION

- While ugly design is expected, your code should be production quality and well tested.
- Surprisingly large amounts of our initial prototype code made it into the final product.
- "of course we'll rewrite this" only works if you have perfect knowledge
- Still not an excuse for over-design.

MANAGING BRANCHES

- Daily integrations are strongly encouraged.
- Key considerations:
 - LLVM evolves quickly, internal APIs are not stable
 - Isolating regressions
 - Revert-to-green culture *for recent changes*
- Can be very time consuming if not carefully arranged.

ONE WORKING BRANCH DESIGN

pristine -- plain LLVM passing make check (w/our build config)

merge-unstable - merged with local changes (conflict resolutions)

testing - VM merge and basic tests pass (catches all API changes)

merge-stable - last point known to pass all tests

master

FRONTEND INTRINSICS

- Replace body of A::foo with hand written IR
- Here be Dragons:
 - Major implications for correctness testing
 - Too easy to side-step hard issues
- Hard Lessons:
 - Express all preconditions in source language
 - Intrinsicify only the critical bit
 - Be willing to change core libraries early

METADATA HEALING TRADEOFFS

- Pros:
 - Robust to upstream changes
 - Robust to pass order changes
 - Works for custom and upstream metadata
 - Testable
- Cons:
 - Wastes compile time
 - Can't always recover lost information

Custom passes continued...

CUSTOM PASSES

Pros:

- Easy to implement
- Handle language specific IR patterns
- Required for certain high level optimizations

Cons:

- Simplicity is often deceptive
- May cover a hard problem which needs solved
- Not in the default pipeline, major testing implications

KEY LESSON: SHORT VS LONG TERM

- Improving existing passes is the right long term answer.
- Not all problems need long term solutions.
- A specialized pass generally gets 80% of cases, much sooner.

KEY LESSON: TESTING

- Code that runs rarely is often buggy. Subtly so.
- Unit testing only goes so far.
- Our painful lessons:
 - Inductive Range Check Elimination (IRCE)
 - Loop Predication
- (Only) three options:
 - Extensive, exhausting testing via fuzzers
 - Well factored shallow code
 - Minimize language specific code