New PM: taming a custom pipeline of Falcon JIT

Fedor Sergeev

Azul Systems
Compiler team

AGENDA

- Intro to Falcon JIT
- Legacy Pass Manager
  - Falcon-specific problems
- New Pass Manager
  - Design & current status
- Falcon port to New Pass Manager
  - Individual passes
  - Current pipeline
  - Numbers
- TODOs

Falcon JIT

- Optimizing LLVM-*based* Java JIT compiler
- Default Top-Tier compiler in the Azul Zing VM
- Custom “opt” pipeline, -O3 codegen pipeline
    - always runs with profile from Tier-1
- Upstream (TOT) based!
- More details on how and why:
    - see US LLVM Dev 2017 keynote talk by Philip Reames: *“Falcon: an optimizing Java JIT”*
    - see EuroLLVM Dev 2017 talk by Artur Pilipenko, “Expressing high level optimizations within LLVM”

Falcon pipeline

- Codegen pipeline ~ stock -O3
- Optimization pipeline is fully custom and … HUGE
  - on a small 200-lines IR
  - ~700 lines in -debug-pass=Structure output (52 PassManagers)

    (vs <300 in stock opt -O3; 18 PassManagers)
  - 2100 individual runs in -debug-pass=Execution trace

    (vs 500 in stock opt -O3)
- Why?
  - Multiple stages of Java semantics lowerings
  - Separate custom devirtualization iteration
  - Obsessive attention to loop performance

Falcon pipeline, contd...

- Upstream passes contributed by Azul, not in stock pipelines:
    - Inductive Range Check Elimination
    - Loop Predication
    - Rewrite Statepoints For GC
- ~20 downstream passes
    - Either utility/experimental or Java/VM-specific

LLVM Pass Manager

- Pass : IR unit → IR unit
- Pass Manager:
  - structure of the pipeline
  - dependencies
  - execution - walk through the pipeline graph
- pipeline structure is "nested" similar to IR units (nested Pass Managers)

  Module ← CGSCC← Function ←Loop←BasicBlock

- Graph structure determines Pass execution order

Legacy Pass Manager

- hierarchy of classes: llvm::**Pass**

```
class Pass {
  virtual bool doInitialization(Module &) = 0;
  virtual bool doFinalization(Module &) = 0;
  virtual Pass *createPrinterPass() = 0;
};
class ModulePass :   public Pass {  virtual bool runOnModule(Module &M) = 0; };
class FunctionPass : public Pass {  virtual bool runOnFunction(Function &F) = 0; };
```

- hierarchy of classes: llvm::legacy::**PassManagerBase**

```
class PassManager : public PassManagerBase {
  void add(Pass *P) override;
  bool run(Module &M);
};
```

- Analyses are Passes, managed by Pass Manager

```
class DominatorTreeWrapperPass : public FunctionPass {
  bool runOnFunction(Function &F) override;
};
```

Legacy PM: **features**/issues

- Passes are registered prior to being added
- Passes have their dependencies encoded at Pass registration time
- Dependencies read from Passes as they are added to the Pass Manager
- Static pipeline schedule is created
- Static pipeline structure is kept **immutable**

  There is no way to dynamically modify the schedule :(
- it works! :)

Legacy Pass Manager: features/**issues**

- nested nature of pipeline is not *explicit* in source code
- BarrierNoOpPass is a hack created to control nesting:

```
MPM.add(Inliner);
// FIXME: The BarrierNoopPass is a HACK! The inliner pass above implicitly
// creates a CGSCC pass manager, but we don't want to add extensions into
// that pass manager.
MPM.add(createBarrierNoopPass());
MPM.add(SomePass()); // goes WHERE?
```

!! Implicit nesting makes order of execution unobvious !!

- Arbitrary limitations on how passes can depend on an analysis
  - Module passes have a hack to depend on Function pass analyses
  - But not SCC passes...
- No conditional invalidation of analyses
  - It is all decided by the static structure

Falcon Issues with Legacy Pass Managers

- Giant pipeline, lots of Passes/Analyses
- Eats CPU time massively, small methods take 10+ms to compile
- Always with Profile Info:
  - but **Inliner** can't use **BranchProbabilityInformation** :-O
- *Would* use even more analyses in Inliner: **DomTree/LoopInfo/MemorySSA**
- Falcon pipeline de-facto contains groups of passes:
  - Worker pass + Cleanup passes
  - … no need for cleanup if worker does nothing
  - … no way to efficiently implement that in Legacy PM

New Pass Manager

- Effort started ... 2012/2013, by Chandler Carruth
  - Jul 11, 2012; "RFC: Pass Manager Redux"
  - Sep 15, 2013; "Heads up: Pass Manager changes will be starting shortly"
- After all these years it is still **New**!
  - May 05, 2016; "Status of new pass manager work"
  - Oct 18, 2017; "RFC: Switching to the new pass manager by default"
- dependencies tracked here: (?)
  - https://bugs.llvm.org/showdependencytree.cgi?id=28315
  - still quite a few (~5 non-umbrella PRs)

New Pass Manager: easy!

- no single Pass hierarchy:
  - inherit PassInfoMixin<> boilerplate helper
  - simply define method:
    `PreservedAnalyses run (IRUnitT &IR, AnalysisManagerT &AM ...);`
  - `llvm::PreservedAnalyses`
    - a set of analyses preserved after a transformation
    - replaces bool result of legacy runXXX methods
- register your Pass for PassBuilder in PassRegistry.def
- Templatized `llvm::PassManager`, `llvm::AnalysisManager`
- PassManager iterates through passes over a single IR unit
  - analyses are requested through AnalysisManagers
- Pipeline construction is very **explicit**

New PM: Adaptors, pipeline beauty

- *Function*Pass → ModulePassManager
- Explicit use of adaptors:
  - ModuleToFunctionPassAdaptor
    - runs function pass(es) over every Function in a Module
  - ModuleToPostOrderCGSCCPassAdaptor
    - runs CallGraph SCC pass(es) over every SCC in a CallGraph of a Module
  - CGSCCToFunctionPassAdaptor
    - runs function pass(es) over every Function in SCC

- Canonicalization passes - dedicated pipelines:

```
FunctionToLoopPassAdaptor::FunctionToLoopPassAdaptor(LoopPassT Pass) {
  LoopCanonicalizationFPM.addPass(LoopSimplifyPass());
  LoopCanonicalizationFPM.addPass(LCSSAPass());
}
```

New PM: Analyses & Passes

- Analysis : IR → result

```
DominatorTree DominatorTreeAnalysis::run(Function &F, FunctionAnalysisManager&) {
  DominatorTree DT;
  DT.recalculate(F);
  return DT;
}
```

  - result may actually be lazy

- Pass has a direct access to the **AnalysisManager** corresponding to its IRUnit

```
PreservedAnalyses InstCombinePass::run(Function &F, FunctionAnalysisManager &AM) {
```

- Gets analysis result through queries to **AnalysisManager**

```
auto &DT = AM.getResult<DominatorTreeAnalysis>(F);
auto *LI = AM.getCachedResult<LoopAnalysis>(*F);
```

- Analysis managers do **caching** and **invalidation** of results

New PM: Proxies

- Proxy - analysis that caches result of *outer* or *inner* analysis
- Module Pass needs Function Analysis?

```
PreservedAnalyses RewriteStatepointsForGC::run(Module &M, ModuleAnalysisManager &AM) {
  // getting "inner" FunctionAnalysisManager from a ModuleAnalysisManager
  FunctionAnalysisManager &FAM =
    AM.getResult<FunctionAnalysisManagerModuleProxy>(M).getManager();
  auto &DT = FAM.getResult<DominatorTreeAnalysis>(F);
}
```

- Function Pass needs Module Analysis?

```
PreservedAnalyses LoopUnrollPass::run(Function &F,FunctionAnalysisManager &AM) {
  const ModuleAnalysisManager &MAM =
      AM.getResult<ModuleAnalysisManagerFunctionProxy>(F).getManager();
  ProfileSummaryInfo *PSI = MAM.getCachedResult<ProfileSummaryAnalysis>(*F.getParent());
}
```

- Reasonable restriction - can't do getResult() from a readonly proxy
- Can't force a run of *outer* analysis from within an *inner* unit transform

Falcon port to New Pass Manager

- All the *required* passes were ported:
  - 20 downstream passes
  - InductiveRangeCheckElimination
  - RewriteStatepointsForGC
- NoUnwind inference added to PostOrderFunctionAttrs
  - Replacement for PruneEH
- Patches to fix a few minor issues (AA usage in InstCombine etc)
- Single command-line flag to switch between NewPM and OldPM
- <3 man-months

New PM: Converting Pass

- Process of single Pass conversion is rather mechanical
- Refactoring for passes with nontrivial doInitialization()
- Separating get-analysis part from the actual transformation

```
bool RewriteStatepointsForGC::runOnModule(Module &M) {
  for (Function &F : M)
    runOnFunction(F);
}

bool RewriteStatepointsForGC::runOnFunction(Function &F) {
  DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>(F).getDomTree();
  // Do Rewrite using DT
}
```

New PM: Converting Pass

- Separating get-analysis part from the actual transformation

```
bool RewriteStatepointsForGCLegacyPass::runOnModule(Module &M) {
  RewriteStatepointsforGC Impl;
  for (Function &F : M) {
    auto &DT = getAnalysis<DominatorTreeWrapperPass>(F).getDomTree();
    Impl.runOnFunction(F, DT);
  }
bool RewriteStatepointsForGC::runOnFunction(Function &F, DominatorTree &DT) {
  // Do Rewrite using DT
}
PreservedAnalyses RewriteStatepointsForGC::run(Module &M, ModuleAnalysisManager &AM) {
  auto &FAM = AM.getResult<FunctionAnalysisManagerModuleProxy>(M).getManager();
  for (Function &F : M) {
    auto &DT = FAM.getResult<DominatorTreeAnalysis>(F);
    runOnFunction(F, DT);
  }
}
```

Transformation

Get analysis

New PM Falcon pipeline

- With Adaptors it looks quite *“nesty”*, compare :

```
MPM.addPass(AlwaysInlinerPass())
{
 FunctionPassManager FPM;
 FPM.addPass(GVN());
 {
  LoopPassManager LPM;
  LPM.addPass(LICMPass());
  LPM.addPass(LPM, SimpleLoopUnswitchPass(false));
  FPM.addPass(createLoopAdaptor(std::move(LPM));
 }
 FPM.addPass(InstCombinePass());
}
MPM.addPass(createFunctionAdaptor(std::move(FPM)));
```

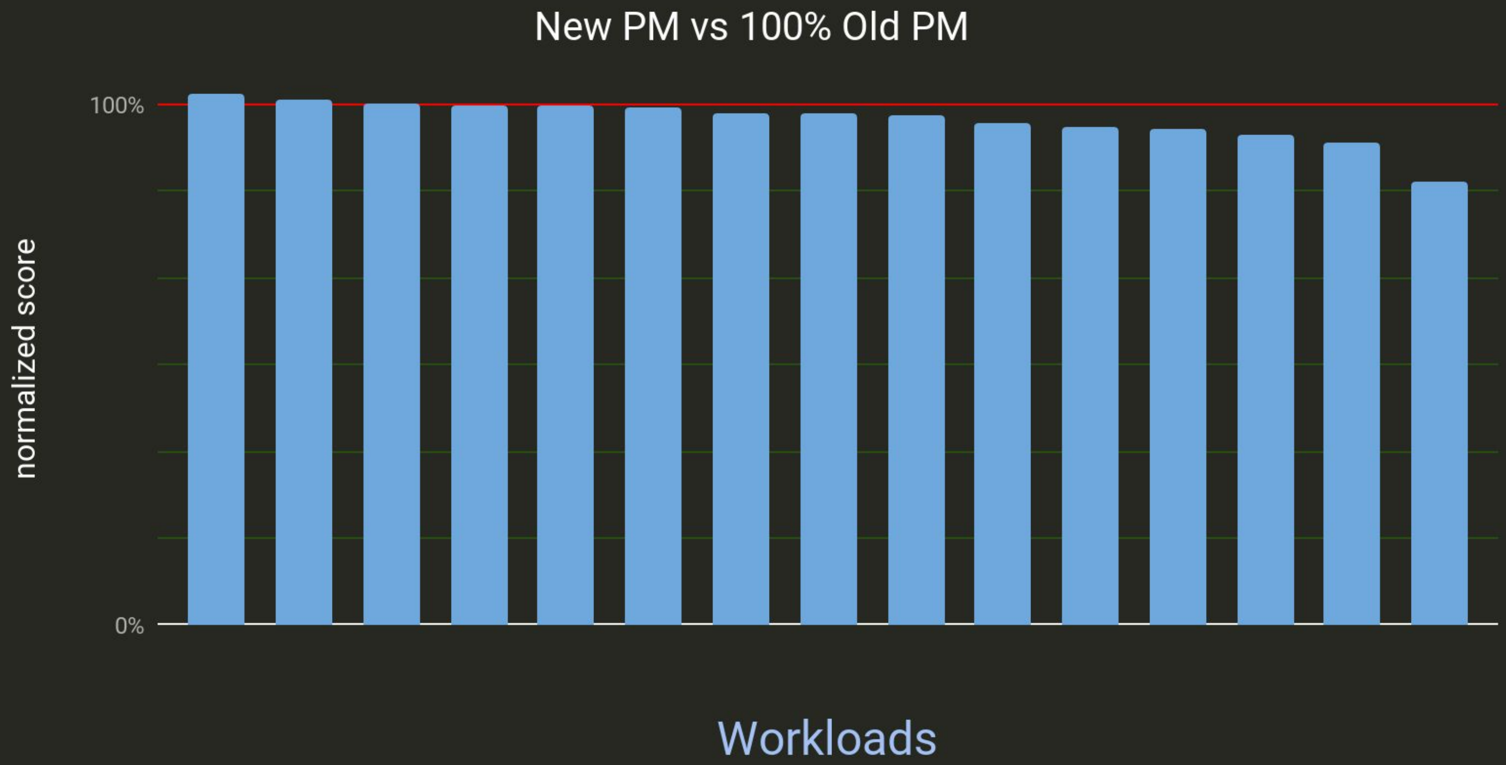
```
PM.addPass(createAlwaysInlinerLegacyPass());
PM.addPass(createBarrierNoopPass());
{
  PM.addPass(createGVNPass());
  {

    PM.addPass(createLICMPass());
    PM.addPass(createLoopUnswitchPass(true));
   }
}
PM.addPass(createInstructionCombiningPass());
```

- Functionally it is *almost* identical...

New PM migration: observations

- LoopUnswitch is a completely new code
  - some functionality is missing
  - thanks to parallel development? :-(
- Inliner is partially a new code, though uses a common InlineCost
  - Heuristics need to be tuned
  - Yes, it already uses BPI !! :-D
- Loop passes can not use BPI yet
  - even IRCE and LoopPredication, which already rely on it
  - There is a solution - LoopStandardAnalyses
- -print-before/after-all **not** implemented at all
- -time-passes **not** implemented at all

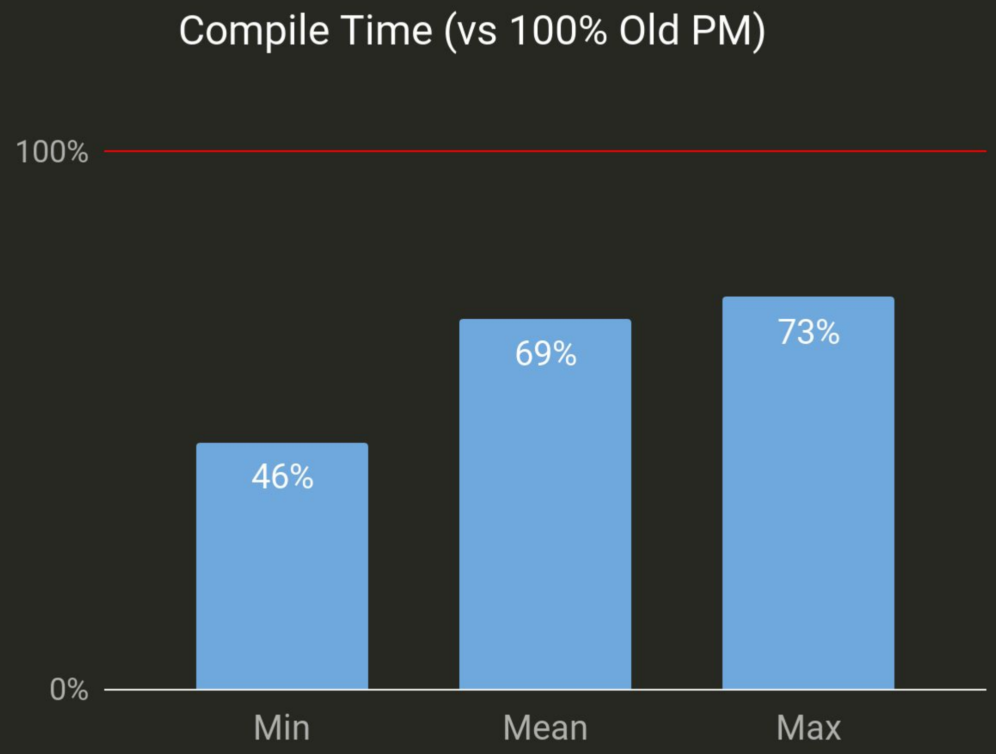
Falcon: **Produced code Performance**



Why not 100% everywhere?

- LoopUnswitch is a completely new code
  - functionality is missing in Non-trivial unswitch
  - Bug in non-trivial unswitch - *PR36379* (assert when modifying loop structure)
  - Non-trivial unswitch ***off*** →regressions in Java-specific benchmarks
- Inliner has not been tuned yet
- IRCE/LoopPredication are less effective w/o BPI

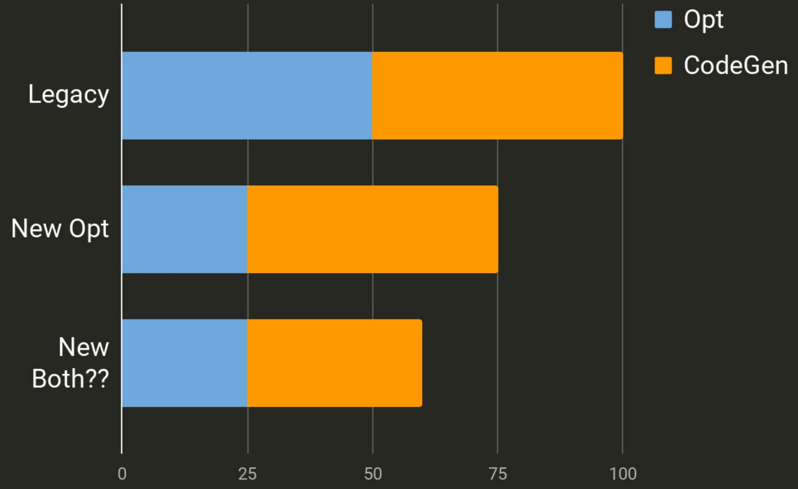
New PM: **Compile Time**



New PM: Compile Time

- Compile-time reduction comes from an improved pass/analyses management
- Now only in **Opt**
- CodeGen is still **not** under New PM
- Imagine overall savings with

  CodeGen not redoing all the analyses



- CodeGen Hero to save the World! :)

New Pass Manager

- Effort started ... 2012/2013, by Chandler Carruth
    - Jul 11, 2012; "RFC: Pass Manager Redux"
    - Sep 15, 2013; "Heads up: Pass Manager changes will be starting shortly"
- After all these years it is still **New**!
    - May 05, 2016; "Status of new pass manager work"
    - Oct 18, 2017; "RFC: Switching to the new pass manager by default"
- Still not default ??

New PM by default - TODOs

- Implement missing developer features:
  - -print-before/after, -time-passes, -opt-bisect
- Non-trivial LoopUnswitch
  - Fix *PR36379* (assert when modifying loop structure)
  - move functionality from legacy version
- Tune inlining heuristics
- Add BranchProbabilityInformation to LoopStandardAnalyses (as optional dep)
  - Needed for IRCE, LoopPredication

fedor.sergeev@azul.com

Questions?