Custom benefit-driven inliner in Falcon JIT

Artur Pilipenko
apilipenko@azul.com
What is Falcon?

- JIT compiler for Java based on LLVM
  - Java bytecode => native
  - Inside of a running JVM
- Final tier compiler in Azul’s Prime JVM (formerly known as Zing JVM)
  - Compiles only the hottest methods
  - Focus on performance
If you want to learn more

LLVM Dev Meeting 15 - LLVM for a managed language: what we’ve learned
https://llvm.org/devmtg/2015-10/#talk14

LLVM Dev Meeting 17 - Falcon: An optimizing Java JIT
https://llvm.org/devmtg/2017-10/#talk12

EuroLLVM 17 - Expressing high level optimizations within LLVM
http://llvm.org/devmtg/2017-03//2017/02/20/accepted-sessions.html#10

EuroLLVM 18 - New PM: taming a custom pipeline of Falcon JIT
https://llvm.org/devmtg/2018-04/talks.html#Talk_13

LLVM Virtual Dev Meeting 20 - Control-flow sensitive escape analysis in Falcon JIT
**Inlining**

Replacement of a function call site with the body of the called function

```c
int foo() {
    return bar(5);
}

int bar(int x) {
    return x + 1;
}
```

```c
int foo() {
    return 5 + 1;
}

int bar(int x) {
    return x + 1;
}
```
Inlining effects

Eliminates call overhead
Enables intraprocedural optimizations
  ○ Facilitates information flow by removing the call boundary
  ○ Specializes the called function for the call site

Increases IR size
  ○ Increases compile-time
  ○ May hurt optimizations
Increases code size
  ○ May hurt performance

Balancing these effects is critically important for performance
Inlining in LLVM
Inliners in LLVM

- Bottom-up inliner
  - Default inliner in standard pipelines, e.g., default inliner in Clang
  - See lib/Transforms/IPO/Inliner.cpp
- Module inliner
  - Experimental inliner
  - See lib/Transforms/IPO/ModuleInliner.cpp
- Partial inliner
  - Inlining hot region only
  - See lib/Transforms/IPO/PartialInlining.cpp
Bottom-up inliner

Traverses the strongly connected components (SCCs) of the call graph in bottom-up order

Call graph

DAG of call graph SCCs
Bottom-up inliner

● Starts with leaf SCCs (typically leaf functions)
● Inlines the callees, runs simplifications
  ○ Iterates if some calls were devirtualized
● Moves up to their callers
Bottom-up inliner

- Starts with leaf SCCs (typically leaf functions)
- Inlines the callees, runs simplifications
  - Iterates if some calls were devirtualized
- Moves up to their callers

No callees to inline, just runs simplifications
Bottom-up inliner

- Starts with leaf SCCs (typically leaf functions)
- Inlines the callees, runs simplifications
  - Iterates if some calls were devirtualized
- Moves up to their callers

Inlines callees, then runs simplifications
Bottom-up inliner

- Starts with leaf SCCs (typically leaf functions)
- Inlines the callees, runs simplifications
  - Iterates if some calls were devirtualized
- Moves up to their callers

Inlines callees, then runs simplifications
Bottom-up inliner

- Starts with leaf SCCs (typically leaf functions)
- Inlines the callees, runs simplifications
  - Iterates if some calls were devirtualized
- Moves up to their callers

In most cases it is looking at fully simplified callees
Module inliner

- Experimental inliner
- Considers all call sites in the module within the same worklist
  - This can possibly enable heuristics that are not limited by the bottom-up traversal order
- We are not taking the advantage of it yet
InlineCost analysis

- Decides whether a given call site should be inlined
- Computes two parameters
  - Cost - estimate of the code size increase in case we inline
  - Threshold - how much cost we are willing to spend to inline this call site
    - Beneficial call sites have higher threshold
- If cost < threshold → inline
Caller context in InlineCost analysis

InlineCost takes the caller context into account
Tries to predict future simplifications enabled by inlining

```c
int bar(bool f) {
    if (f) return 0;
    // lots of code
}

void foo() {
    bar(true)
}
```

InlineCost can recognize that bar(true) is free to inline
**InlineCost bonuses**

- When a beneficial inlining pattern is recognized, a bonus is applied to encourage inlining
  - Either by increasing the threshold, or by decreasing the cost
- For example, we give a bonus if inlining enables a devirtualization opportunity
Inlining in Falcon
Compilation model in Falcon

- Compilation unit - one Java method
- VM requests a compilation of one method and expects to get native code for this method
- The result of the compilation can inline other methods
On demand generation of inline candidates

- We start with one function in the module
  - The method requested by the VM
- If we want to inline we need to request the inlining candidate from the VM
  - VM will parse Java bytecode and generate the LLVM IR
  - We import the new function into our module and can now inline
On demand generation of inline candidates

● The full call graph is not available
● We dynamically explore the call graph of functions reachable from the top-level
● Bottom-up inlining is problematic
● Top-down traversal is very straightforward
Falcon’s downstream top-down inliner

- Initialize the worklist with all call sites in the top-level function
- For each call site in the worklist
  - Generate candidate
  - Simplify candidate
  - Consider for inlining using InlineCost
    - If inlined add new inlined calls into the worklist

```
foo
  ↓
bar
  ↓
baz
```
Falcon’s downstream top-down inliner

- Initialize the worklist with all call sites in the top-level function
- For each call site in the worklist
  - **Generate candidate**
  - **Simplify candidate**
  - Consider for inlining using InlineCost
    - If inlined add new inlined calls into the worklist
Falcon’s downstream top-down inliner

- Initialize the worklist with all call sites in the top-level function
- For each call site in the worklist
  - Generate candidate
  - Simplify candidate
  - **Consider for inlining using InlineCost**
    - If inlined add new inlined calls into the worklist
Falcon’s downstream top-down inliner

- Initialize the worklist with all call sites in the top-level function
- For each call site in the worklist
  - *Generate candidate*
  - *Simplify candidate*
  - Consider for inlining using InlineCost
    - If inlined add new inlined calls into the worklist
Falcon’s downstream top-down inliner

- Initialize the worklist with all call sites in the top-level function
- For each call site in the worklist
  - Generate candidate
  - Simplify candidate
  - **Consider for inlining using InlineCost**
    - If inlined add new inlined calls into the worklist
Falcon’s downstream top-down inliner

• Iterate inlining and simplifications on the top-level function until the fixed point
• Simplifications in the top-level function can open new opportunities for inlining
  ○ devirtualization
  ○ improved information about arguments
Summary

- We’ve implemented a downstream top-down inliner
- On-demand generation of candidates
- Iterative application of inlining and simplifications
Total budget and prioritization
Top-level function size blow up

- Top-down inlining makes it possible to blow up the size of the top-level function
  - InlineCost decisions are local
  - Iterative application of reasonable local decisions can result in overinlining
- Less of an issue for bottom-up inlining
  - As a bottom-up inliner progresses up the call graph, it increases the size of functions
  - This, in turn, prevents their inlining further up
Total budget for inlining

- We introduced a total budget for inlining into the top-level function
- With total budget the order of inlining becomes important
  - Can waste our budget on non-beneficial calls and not be able to inline beneficial calls later
- We need prioritization of call sites and a universal measure of inlining “goodness”
Cost?

- We already have cost computed by InlineCost
- This is not enough for good prioritization
  - A large callee can be more beneficial to inline
  - E.g., out of these two we should prefer the former
    - a large callee in a hot loop that enables a devirtualization opportunity once inlined
    - small callee on a cold code path
Cost/benefit analysis

- We introduced an explicit measure of benefits and prioritize by benefit/cost ratio
- Benefits are computed by a downstream part of InlineCost
- It is based on InlineCost bonuses
  - Benefit is the sum of all the bonuses we’ve given
  - We’ve added multiple bonuses for Java-specific optimizations
- It is NOT based on cost/benefit analysis in upstream InlineCost
  - There is no good reason for us to have a separate implementation
  - Ideally, we want to merge our cost/benefit analysis with the one upstream
Performance impact - disabled total budget

- Renaissance Suite v0.11
  - 25 workloads
- Azul Prime JVM JDK 8, dev ToT build
- Intel Skylake 4 core machine
- Total compile time
  +76%
- Geomean of scores
  -3.1%
Summary

- Doing top-down inlining requires a total budget for the top-level function
- Total budget requires prioritization
- We introduced an explicit notion of benefits
  - Based on InlineCost bonuses
- Prioritize by benefit/cost ratio
  - Helps us avoid bad inlining when we exhaust the budget
Top-down vs bottom-up
**Top-down vs bottom-up inlining**

- Doing pure top-down we lose some benefits of bottom-up inlining
- Bottom-up inliner is looking at fully simplified candidates
  - Sees more context and more simplification opportunities
- Trivial functions like getters/setters obscure these opportunities

```c
int foo() {
    return bar(5);
}

int bar(int x) {
    return x + baz();
}

int baz() {
    return 5;
}
```

Looking top-down, we can’t see the constant folding opportunity
Top-down vs bottom-up inlining

- Doing pure top-down we lose some benefits of bottom-up inlining
- Bottom-up inliner is looking at fully simplified candidates
  - Sees more context and more simplification opportunities
- Trivial functions like getters/setters obscure these opportunities

```
int foo() {
    return bar(5);
}

int bar(int x) {
    return x + 5;
}

int baz() {
    return 5;
}
```

Inlining bottom-up, the constant folding opportunity becomes apparent
Simplify-with-inlining

- We do limited form of bottom-up inlining
- We recursively apply the top-down inliner during simplifications of the candidates
- Effectively, we pregenerate 2 levels of the call graph and run bottom-up inlining
Simplify-with-inlining

- We do limited form of bottom-up inlining
- We recursively apply the top-down inliner during simplifications of the candidates
- Effectively, we pregenerate 2 levels of the call graph and run bottom-up inlining

Simplifying bar, we inline baz into it

foo

bar

baz

Copyright ©2022 Azul Systems Inc.
Simplify-with-inlining

- We do limited form of bottom-up inlining
- We recursively apply the top-down inliner during simplifications of the candidates
- Effectively, we pregenerate 2 levels of the call graph and run bottom-up inlining

Simplifying bar, we inline baz into it
Simplify-with-inlining

- This gives us more context in the candidates
  - Enables more simplifications
  - Makes InlineCost able to recognize more simplifications after inlining
- Mitigates the situation when we need to look through several levels of inlining to see benefits
Inlining policy for inlining into candidates

- Trade-offs of inlining into candidates:
  - Exposes more context in the candidate
  - Inflates the candidate and prevents its inlining
  - Wastes compile-time if candidate is not inlined

- Our policy
  - We use InlineCost analysis with small threshold = 45
  - For context, default inline threshold is 225

- View this inlining as canonicalization
  - Getting rid of trivial methods, like getters/setters/wrappers
Performance impact - disabled bottom-up inlining

- Renaissance Suite v0.11
  - 25 workloads
- Azul Prime JVM JDK 8, dev ToT build
- Intel Skylake 4 core machine
- **Total compile time**
  - -12.8%
- **Geomean of scores**
  - -9.6%
Summary

● Pure top-down inlining loses some benefits of bottom-up inlining
● We do a limited form of bottom-up inlining
   o Pregenerate 2 levels of the call graph and do bottom-up on these functions
● This has significant performance impact
Clustering
Clustering

Sometimes benefits can only be revealed if multiple call sites are inlined together

```java
A a = new A()
foo(a)
bar(a)

void foo(A a) {
    // doesn't escape a
}

void bar(A a) {
    // doesn't escape a
}
```

Inlining both foo(a) and bar(a) will make a fully unescaping
Clusters

- This kind of benefits are not recognized by a single call site inliner
- We want to be able to recognize them
- We need to consider multiple call sites for inlining together
- **Cluster - multiple call sites considered for inlining together**
Examples of clusters

Cluster of siblings:
- foo()
- baz()

Cluster of nested calls:
- foo()
  - bar()

Mixed cluster:
- foo()
- bar()
- baz()
Clustering

● Because of total budget for the top-level method we can’t do cluster inlining outside of regular inlining
  ○ We may run out of budget doing regular inlining
● Clusters must be prioritized against regular inlining decisions
  ○ Clusters must be first-class citizens in the regular inliner
● We made our regular inliner work on clusters, not single call sites
  ○ A single call site is just a trivial cluster
Cluster inliner

- In order to prioritize clusters, we need to know their costs and benefits
  - It is computed as an aggregate of costs and benefits of individual call sites
  - + extra cluster benefit, e.g., the benefit for making an allocation unescaped
- Clustering heuristics look for candidate clusters, e.g., escape-analysis driven clustering
  - Different heuristics populate the priority queue
  - Prioritization takes care of the rest
Escape-analysis driven clustering heuristic

Looking for clusters that make an allocation fully unescaping

```java
A a = new A();
foo(a) // doesn't escape a
bar(a) // doesn't escape a

void foo(A a) { baz(a); }
void bar(A a) { // doesn't escape a }
void baz(A a) { // doesn't escape a }
```

Cluster:
- foo()
- bar()
- baz()
Performance impact - disabled clustering

- Renaissance Suite v0.11
  - 25 workloads
- Azul Prime JVM JDK 8, dev ToT build
- Intel Skylake 4 core machine
- **Total compile time** -3.3%
- **Geomean of scores** -1.5%
Summary

- Some benefits can only be revealed if multiple call sites are inlined together
- We made our inliner work on clusters of call sites
- We implemented EA-driven clustering heuristics
Future work and conclusions
Future work

- Better estimation of benefits
  - In general and for clusters specifically
  - Merge our cost/benefit analysis with the one upstream
- New clustering heuristics
- Cluster merging
Conclusions

● We build a custom downstream inliner in Falcon
  ○ Top-down inliner
  ○ On-demand generation of candidates
    ○ Fixed point iteration of inlining and simplifications in the top-level function
● Total budget and prioritization by benefit/cost ratio
● Limited bottom-up inlining for 2 level of the call graph
● EA-driven cluster inlining
Thank You.