Improved Inlining for OMR

Presented by: Andrew Craik – IBM

Work completed as a research collaboration between IBM and
Erick Ochoa, Karim Ali and Nelson Amaral – University of Alberta

21-Feb-2019

Current State-of-the-Compiler

- OMR has one inliner called the trivial inliner
  - Very basic inliner that works by inlining a limited amount of small methods
  - No concept of ‘best’ things to inline
  - A minimum viable concept of an inliner
- OpenJ9 has a massive & complex inliner called the MultiTargetInliner
  - Developed over the last two decades
  - VERY Java centric – full of heuristics
  - Does a form of guided eager inlining
  - Can miss opportunities because of a depth-first approach to searching
  - Conflates small method with low benefit with large method with large benefit
  - Code is convoluted and hard to reason about / control

How can we do better?

- Inlining provides a number of benefits:
  - Reduced function call overheads
  - Improved opportunities for optimization
- Inlining can have negative effects:
  - Methods can get too large to easily analyze and compile
  - Inlining the ‘wrong’ things may have adverse impact on hardware behavior
- Current state-of-the-art inliners are guided using a single metric
  - You have a budget, you choose candidates to inline until you fill the budget
  - You guide the inliner by inflating/deflating the metric
  - Conflates size and opportunity for optimization – optimality can’t be achieved

A New Inliner - Goals

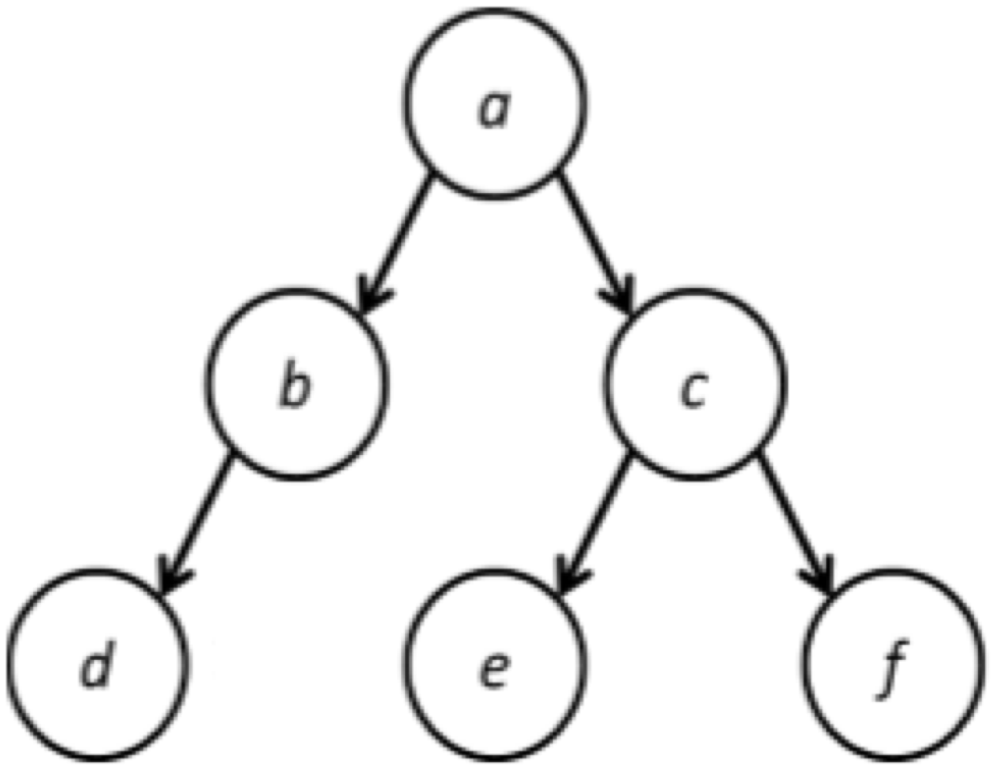
- Separate the notions of cost and benefit
- Benefit should represent the opportunity for improved optimization
- Benefit should also include execution frequency
  - The greatest benefit is from optimizations on the hottest execution paths
- Make inliner guidance more scientific and less ‘magical’

Knapsack Packing with Dependencies

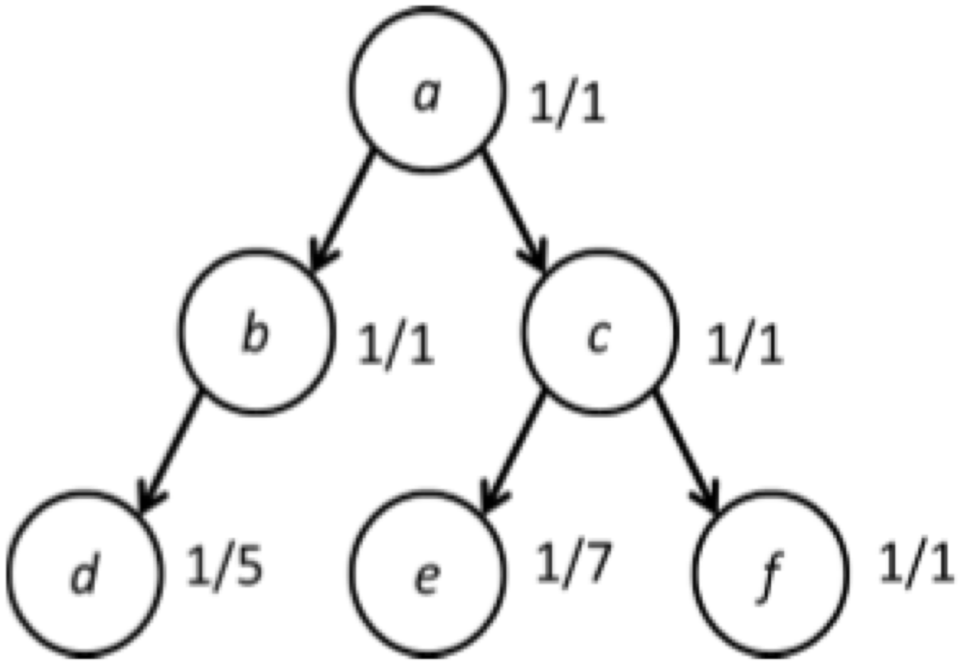
- IBM developed an algorithm to solve the knapsack packing with dependencies problem – work done by Andrew Craik, Rachel Craik and Patrick Doyle
- This algorithm has not been formally proven optimal (yet) but in practice it produces optimal solutions
- It is a dynamic programming algorithm that uses two layers of backtracking to allow for ‘deoptimization’ during the search for the ‘best’ inlining solution

Build an Inlining Dependency Tree

```
// cost: 1 benefit: 1
function a() { … b(); … c(); … }
// cost: 1 benefit: 1
function b() { … d(); … }
// cost: 1 benefit: 1
function c() { … e(); … f(); … }
// cost: 1 benefit: 5
function d() { … }
// cost: 1 benefit: 7
function e() { … }
// cost: 1 benefit: 1
function f() { … }
```



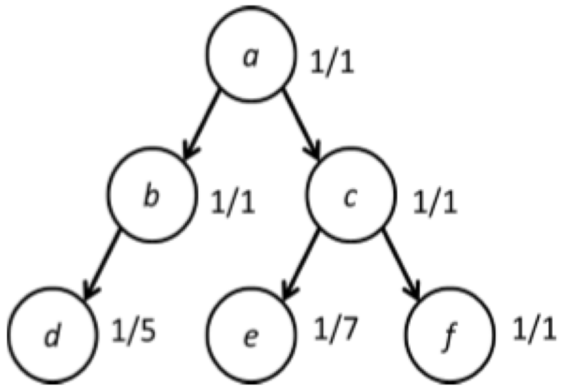
Annotate IDT with Cost & Benefit



Postorder Traversal Considering Inlining

Considering node a:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}



Postorder Traversal Considering Inlining

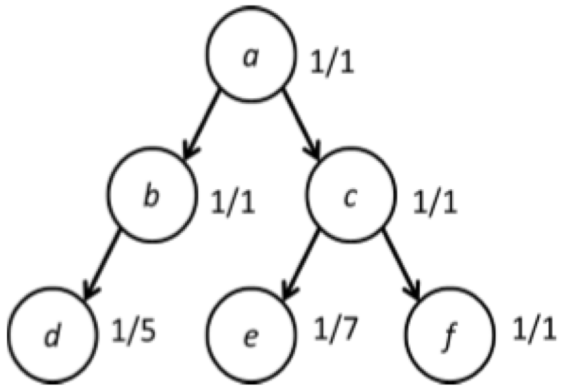
Considering node a:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}

Considering node b:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}

Note: at a cost budget of 1 we can only choose node a – node b is an invalid choice w/o a.



Postorder Traversal Considering Inlining

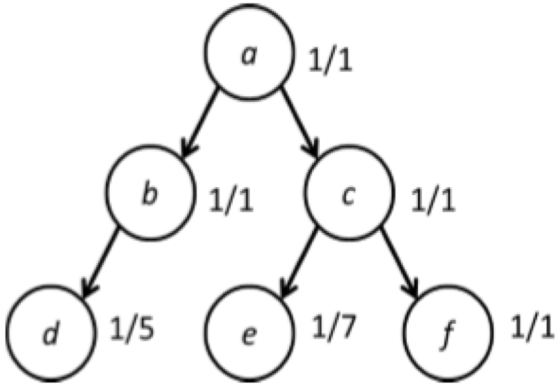
Considering node a:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}

Considering node b:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}

Note: at a cost budget of 1 we can only choose node a – node b is an invalid choice w/o a.



Postorder Traversal Considering Inlining

Considering node a:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}

Considering node b:

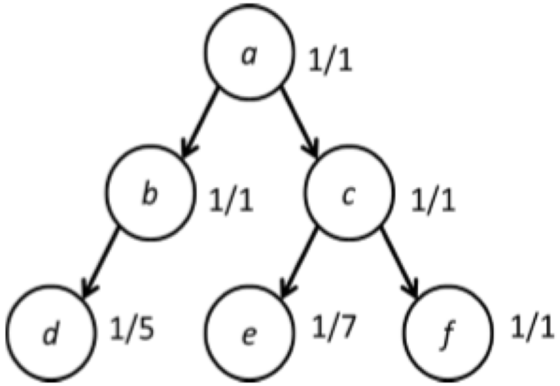
	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}

Note: at a cost budget of 1 we can only choose node a – node b is an invalid choice w/o a.

Considering node d:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}
d	{a}	{ab}	{abd}	{abd}	{abd}

Note: at a cost budget of 1 or 2 we cannot choose node d – it is only a valid choice with a & b



Postorder Traversal Considering Inlining

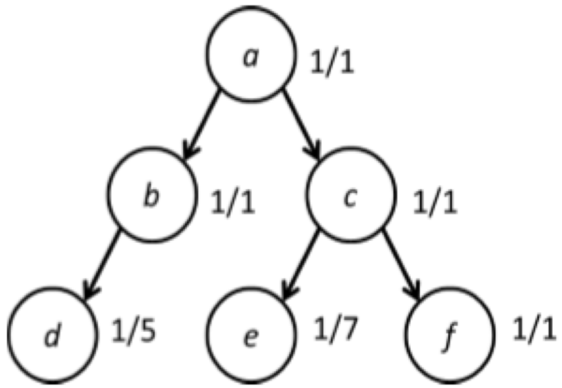
Considering node c:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}
d	{a}	{ab}	{abd}	{abd}	{abd}
c	{a}	{ab}	{abd}	{abdc}	{abdc}

Consider node e:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}
d	{a}	{ab}	{abd}	{abd}	{abd}
c	{a}	{ab}	{abd}	{abdc}	{abdc}
e	{a}	{ab}	{ace}	{abce}	{abcde}

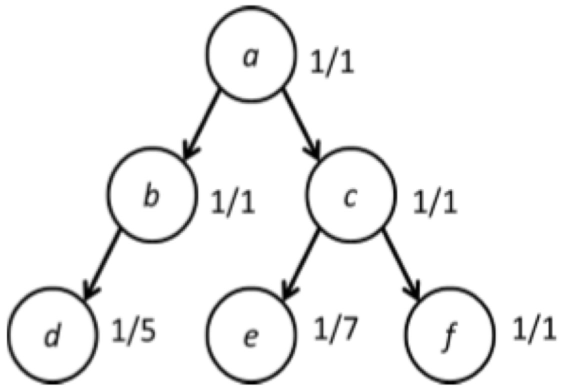
Note that nodes ace have a higher benefit than nodes abd; this is an example of how the algorithm effectively allows backtracking by undoing the decision to include b and switch to including c when a better inlining option becomes available.



Postorder Traversal Considering Inlining

Considering node c:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}
d	{a}	{ab}	{abd}	{abd}	{abd}
c	{a}	{ab}	{abd}	{abdc}	{abdc}



Consider node e:

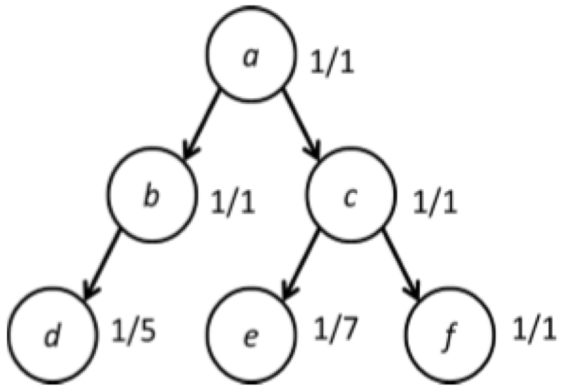
	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}
d	{a}	{ab}	{abd}	{abd}	{abd}
c	{a}	{ab}	{abd}	{abdc}	{abdc}
e	{a}	{ab}	{ace}	{abce}	{abcde}

Note that nodes ace have a higher benefit than nodes abd; this is an example of how the algorithm effectively allows backtracking by undoing the decision to include b and switch to including c when a better inlining option becomes available.

Postorder Traversal Considering Inlining

Considering node c:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}
d	{a}	{ab}	{abd}	{abd}	{abd}
c	{a}	{ab}	{abd}	{abdc}	{abdc}



Consider node e:

	1	2	3	4	5
a	{a}	{a}	{a}	{a}	{a}
b	{a}	{ab}	{ab}	{ab}	{ab}
d	{a}	{ab}	{abd}	{abd}	{abd}
c	{a}	{ab}	{abd}	{abdc}	{abdc}
e	{a}	[illegible]	{ace}	{abce}	{abcde}

Note that nodes ace have a higher benefit than nodes abd; this is an example of how the algorithm effectively allows backtracking by undoing the decision to include b and switch to including c when a better inlining option becomes available.

How to compute benefit?

- Components to consider
  - Frequency – if two things have the same cost inline the hotter one
  - Optimization Opportunity – if two things have the same hotness pick the one that is going to allow the optimizer to do more once inlining is done
- Frequency of a call within a method is represented as the ratio of the method entry frequency to the frequency of the callsite
- Frequency factor is multiplied on the path from the IDT root to the site being considered to give a multiplier to the benefit

Opportunity for Optimization

- Want to model which optimizations may be ‘unlocked’ by the inlining of a method
- Idea
  - Run an abstract interpreter over program representation computing symbolic values / constraints
  - At a call abstract interpret the callee and pattern match optimization opportunities
  - At an opportunity record the constraints, in terms of parameter values, that would prove the optimization could happen
  - Store this summary of potential transformations in a table
  - Intersect the callsite constraints with each potential transformation, sum the benefits of all which could apply, scale for frequency

Abstractions – VP Constraints

Values	Abstract Values
Integers	Integer ranges
Strings	Constant string constraints
Null	Null constraints
Objects	Constraint on class Constraint on nullness
Arrays	Constraint on the array size as a range Constraint on the class

Optimizations Modeled

Optimization	Potential Optimizations
Branch Folding	Integer constraints determine the control flow
Null check elimination	Null check constraints determine whether we can eliminate this instruction
Check cast elimination	Object constraints can determine whether we can eliminate this instruction
Length of constant string	Constraint on class of object
Partial evaluation	How much can we partially evaluate

Sample Method Summary

Opt	Benefit	Location	Arg #1	Arg #2	Arg #3	Arg #4
IFEQ	1	Callee = 15 Bytecode = 33				[0]
IFLT	1	Callee = 15 Bytecode = 5			[0, INT_MAX]	
IFLT	1	Callee = 15 Bytecode = 5			[INT_MIN, -1]	

Evaluation

- Abstract interpreter implemented for Java bytecode in OpenJ9
- Optimizations modeled:
  - Branch folding
  - Null check elimination
  - Check cast elimination
  - Folding of constant string lengths
  - Patrial evaluation
- DaCapo Benchmark suite:
  avora, pmd, lusearch, luindex, fop, eclipse, sunflow

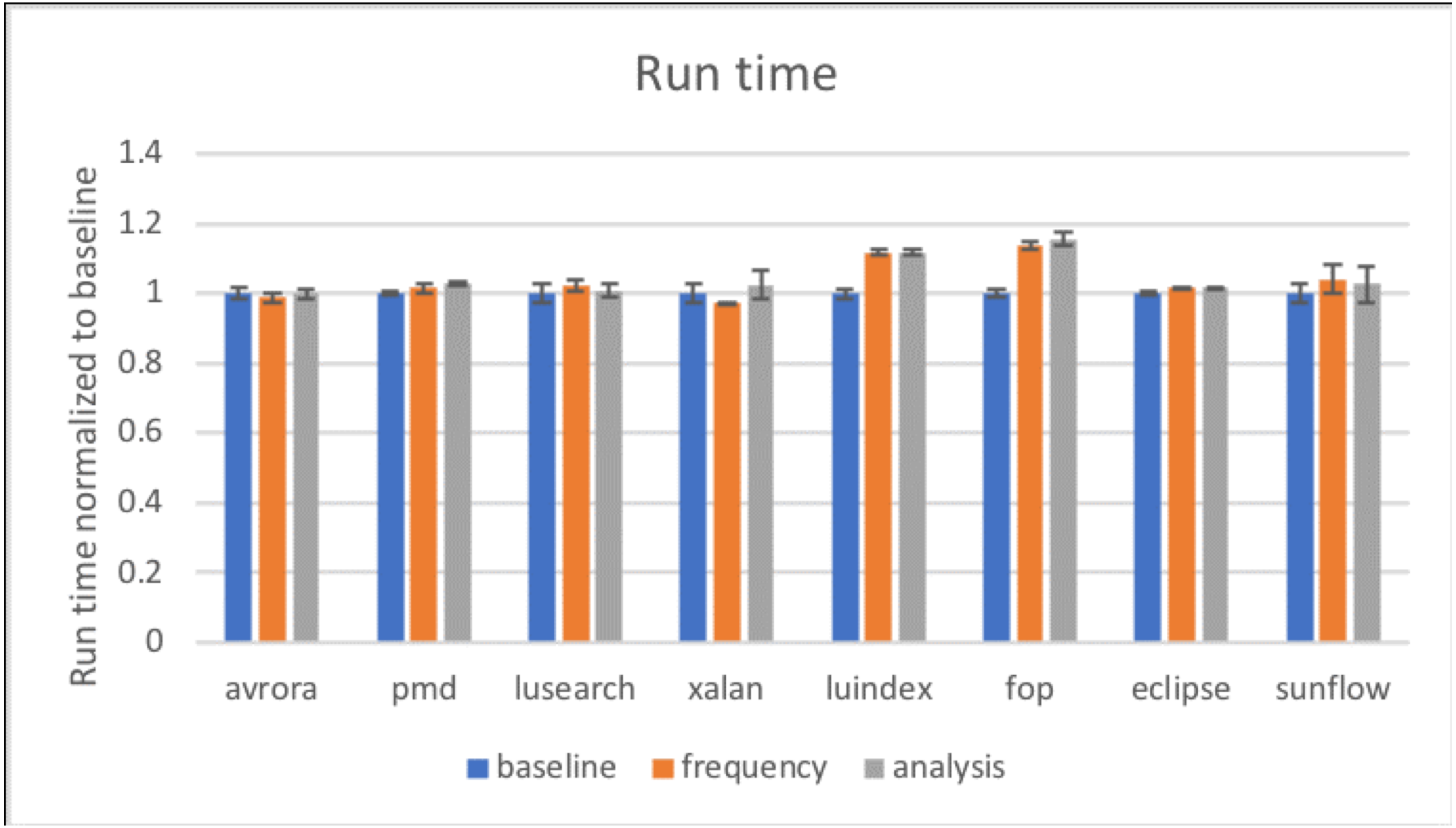
Evaluation - Continued

- Each benchmark runs in a separate JVM
- Benchmark iteratively executed to 'warm' the JVM up – warm up iteration count benchmark specific and determined by average iterations for compilation to cease
- Measurements
  - Compile time – total cpu time consumed by compilation threads (from vlog)
  - Compile memory – total memory consumed during compilation (from vlog)
  - Generated Code Size – number of bytes of instructions generated (from vlog)
  - Runtime – time to run the final iteration after the warm-up period (eg steady-state throughput)

Evaluation - Continued

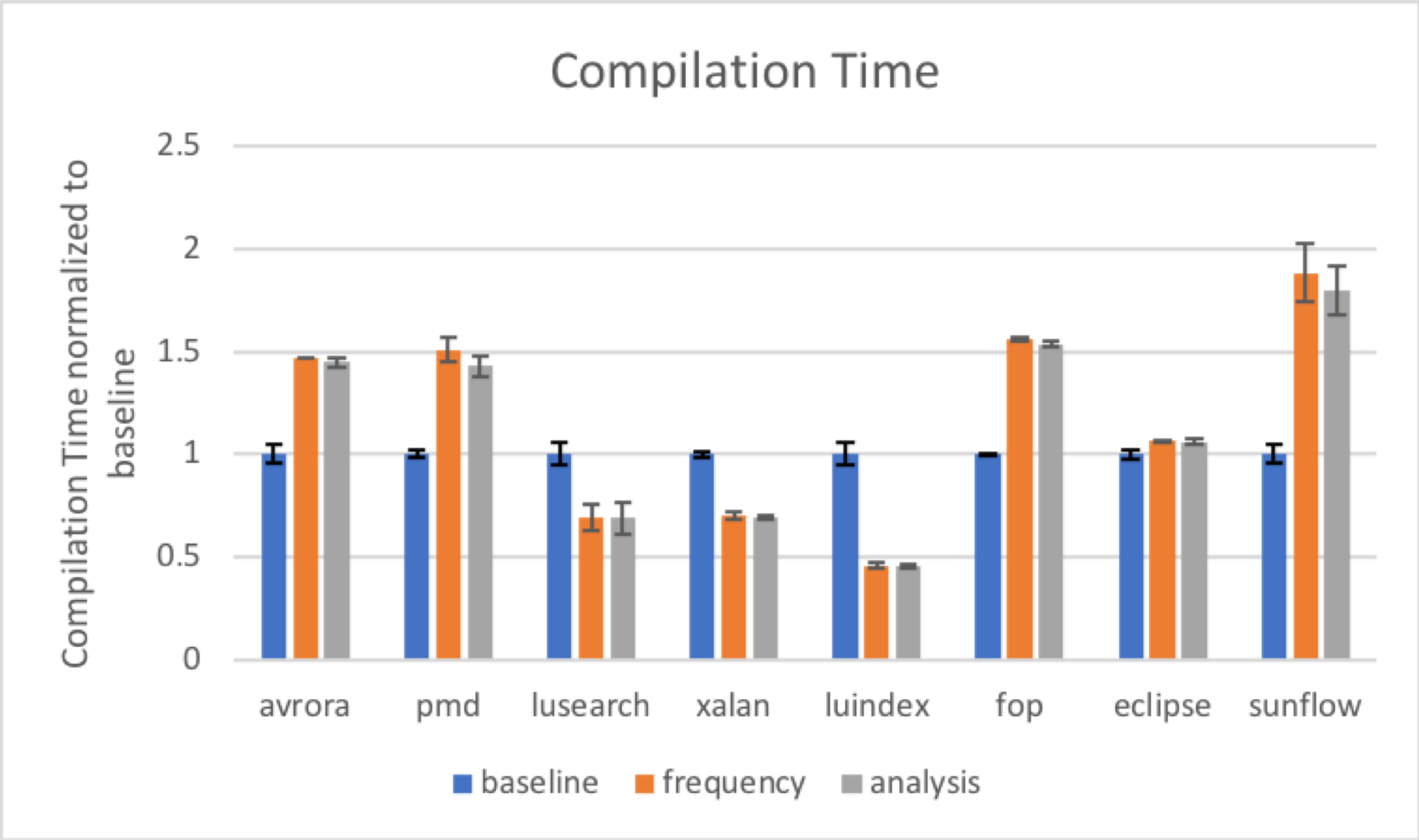
- Compared:
  - Baseline: current OpenJ9 heuristic inlininer
  - Frequency: new inliner with all methods having benefit 1
  - Analysis: new inliner using abstract interpreter benefits & frequency scaling
- Evaluation on x86-64 linux Skylake; heap size set to ensure no global GCs, heap size fixed to prevent growth/shrinkage, machine isolated

Runtime

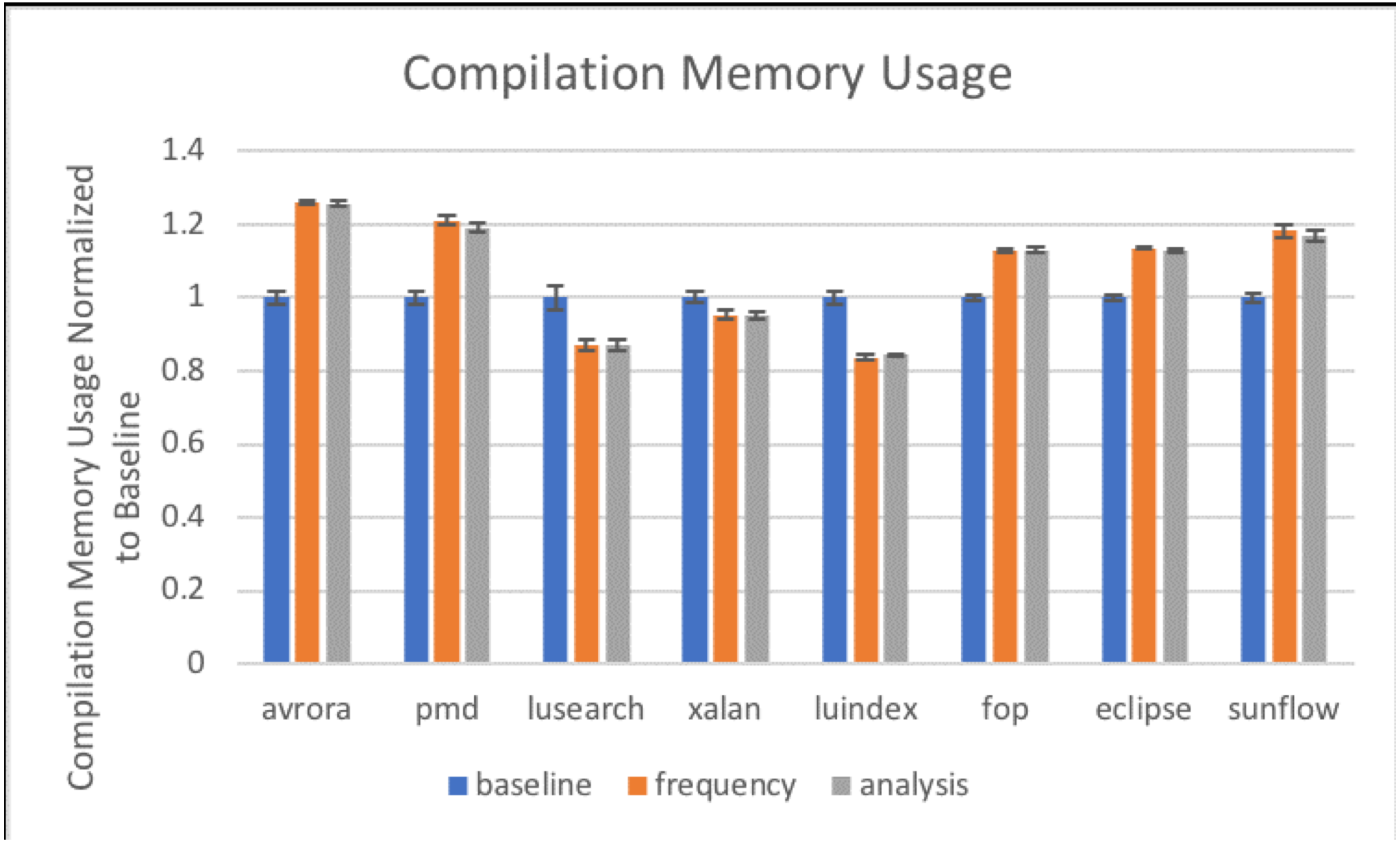


Run time
Run time normalized to baseline
1.4
1.2
1
0.8
0.6
0.4
0.2
0
avrora
pmd
lusearch
xalan
luindex
fop
eclipse
sunflow
baseline
frequency
analysis

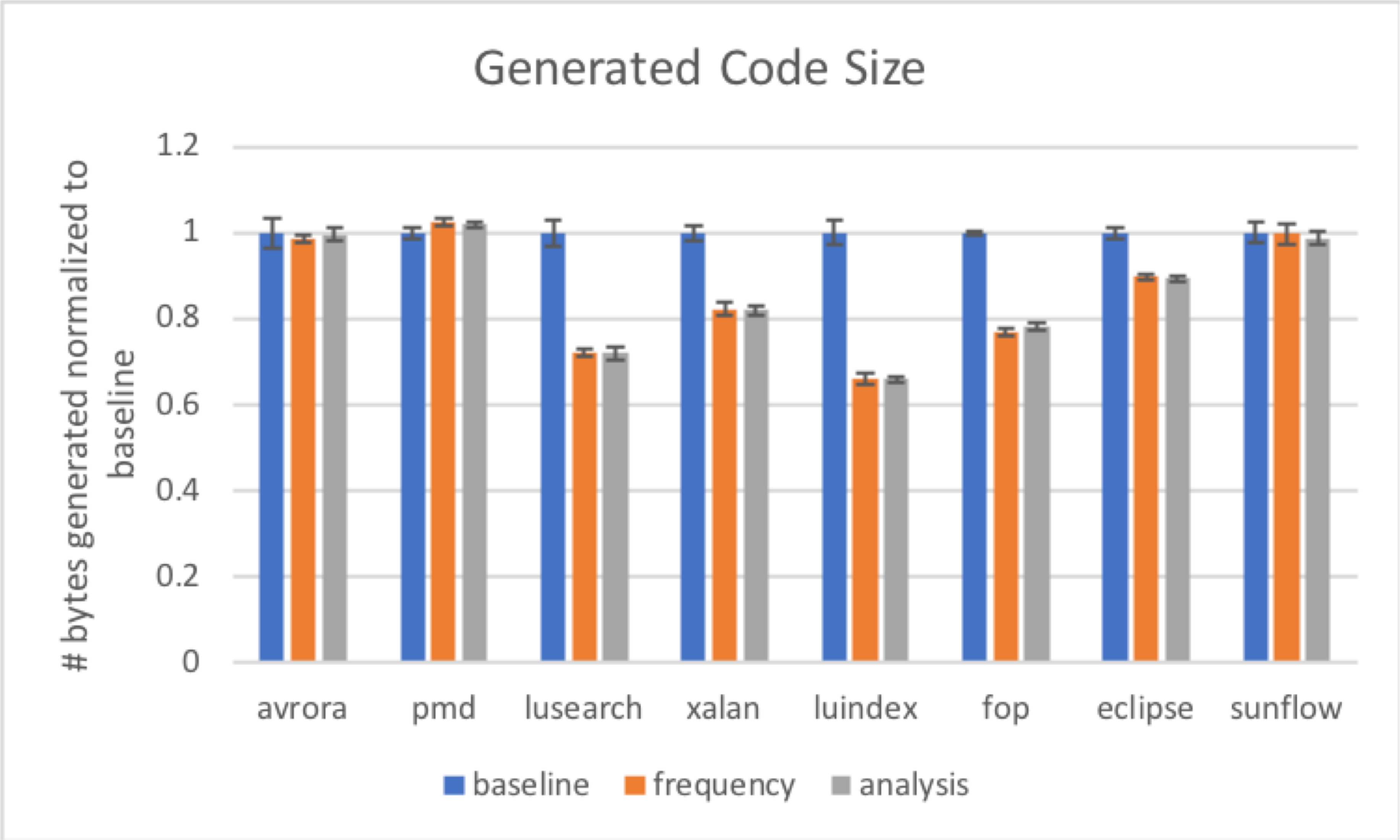
Compile Time



Compilation Memory Usage



Generated Code Size



Analysis

- New inliner is more expensive than OpenJ9's current inliner
  - Current inliner does not do a full exploration of state-space (not guaranteed optimal and may be trapped in local minima)
  - Compile-time generally comparable – worst case was ~2x
  - Memory was within +20% of baseline
- Abstract interpretation is cheap – most of the cost comes from the state space exploration
- New inliner can produce the same performance with less code
- Runtime performance very good considering the limited number of optimizations modeled & lack of heuristics

Future Work

- Current propagation of information is ‘down’ the IDT – add ‘upward’ propagation for improved information in caller for a given callee
- Model more complex optimizations – current thought is Escape Analysis
- Abstract interpreter for trees – OpenJ9 uses bytecodes to save the cost of interpreting trees and other OMR languages may want this

Contribution Proposal

- Place the core of the new inliner in OMR with an abstract API for the abstract interpreter & its results
- Use OptimizationManager to select inliner based on –Xjit option – default to current inliners but allow testing of new inliner
- Contribute Abstract Interpreter implementation to OpenJ9 so they can continue experiments

Q&A / Discussion