Optimizing Flag Selection and Phase-Order in the GHC Compiler

Connor Rhys Peper

Dr. Mike Hewner, Dr. Georg Schied

Rose-Hulman Institute of Technology

pepercr@rose-hulman.edu

May 3, 2024

Overview

Why Haskell?

- Little research done on the GHC
- Haskell is a purely functional programming language
- GHC is written in Haskell!

Goal of My Research

What are some goals of any piece of software?

Goal of My Research

What are some goals of any piece of software?

- Work

Goal of My Research

What are some goals of any piece of software?

- Work
- Readable

Goal of My Research

What are some goals of any piece of software?

- Work
- Readable
- Efficient

Goal of My Research

What are some goals of any piece of software?

- Work
- Readable
- Efficient

Goal of My Research

What are some goals of any piece of software?

- Work
- Readable
- Efficient

Dumb.c

```
int add (int a, int b) {
    int answer = a + b;

    return answer;
}

int main () {
    int a = 3;
    int b = 7;
    int c;

    c = add(a, b);

    return c;
}
```

Un-Optimized Assembly

```
gcc -g -O0 dumb.c
    // int main () {
        endbr64
        push   rbp
        mov    rbp,rsp
        sub    rsp,0x10

        // int a = 3;
        mov    DWORD PTR [rbp-0xc],0x3
        // int b = 7;
        mov    DWORD PTR [rbp-0x8],0x7
        // int c;

      //  c = add(a, b);
        mov    edx,DWORD PTR [rbp-0x8]
        mov    eax,DWORD PTR [rbp-0xc]
```

Unoptmized Assembly Cont.

```
gcc -g -O0 dumb.c
        mov     esi,edx
        mov     edi,eax
        call    1129 <add>
        mov     DWORD PTR [rbp-0x4],eax

        // return c;
        mov     eax,DWORD PTR [rbp-0x4]
        leave
        ret
// }
```

Optimized Assembly Cont.

```
gcc -g -O2 dumb.c
```

```
// int main () {
    endbr64
    mov     eax,0xa
    ret
```

Compiler Improvements

Inlining

```
int c = add(a, b); => int c = a + b;
```

Constant folding

```
int a = 6; int b = 4; int c = a + b; => int c = 10
```

Register allocation

```
mov DWORD PTR [rbp-0xc],0x3
mov eax,DWORD PTR [rbp-0xc]
⇒
mov eax,0x3
```

List of GHC Flags

- -fmax-simplifier-iterations=n
- -fcore-constant-folding
- -fregs-graph

List of GHC Flags

- -fmax-simplifier-iterations=n
- -fcore-constant-folding
- -fregs-graph
- -fcall-arity

List of GHC Flags

- -fmax-simplifier-iterations=n
- -fcore-constant-folding
- -fregs-graph
- -fcall-arity
- -fcase-merge

List of GHC Flags

- -fmax-simplifier-iterations=n
- -fcore-constant-folding
- -fregs-graph
- -fcall-arity
- -fcase-merge
- -fcmm-elim-common-blocks
- -fcmm-sink
- -fcpr-anal
- -fcse
- -fdicts-cheap
- -fdicts-strict
- -fdmd-tx-dict-sel
- -fdo-eta-reduction

List of GHC Flags

- -fmax-simplifier-iterations=n
- -fcore-constant-folding
- -fregs-graph
- -fcall-arity
- -fcase-merge
- -fcmm-elim-common-blocks
- -fcmm-sink
- -fcpr-anal
- -fcse
- -fdicts-cheap
- -fdicts-strict
- -fdmd-tx-dict-sel
- -fdo-eta-reduction
- -fdo-lambda-eta-expansion
- -feager-blackholing
- -fenable-rewrite-rules
- -fexcess-precision
- -fexpose-all-unfoldings
- -ffloat-in
- -ffull-laziness
- -ffun-to-thunk
- -fignore-asserts
- -fignore-interface-pragmas
- -flate-dmd-anal
- -fliberate-case

List of GHC Flags

- -fmax-simplifier-iterations=n
- -fcore-constant-folding
- -fregs-graph
- -fcall-arity
- -fcase-merge
- -fcmm-elim-common-blocks
- -fcmm-sink
- -fcpr-anal
- -fcse
- -fdicts-cheap
- -fdicts-strict
- -fdmd-tx-dict-sel
- -fdo-eta-reduction
- -fdo-lambda-eta-expansion
- -feager-blackholing
- -fenable-rewrite-rules
- -fexcess-precision
- -fexpose-all-unfoldings
- -ffloat-in
- -ffull-laziness
- -ffun-to-thunk
- -fignore-asserts
- -fignore-interface-pragmas
- -flate-dmd-anal
- -fliberate-case
- And 5 more!

Optimization Presets

- -O0 - {}
- -O1 - {-fast-opt, -cool-opt}
- -O2 - {-fast-opt, -cool-opt, -bad-opt}

Optimization Presets

- -O0 - {}
- -O1 - {-fast-opt, -cool-opt}
- -O2 - {-fast-opt, -cool-opt, -bad-opt}
- -Ideal - {-lion-opt, -tiger-opt, -bear-opt}

Steps of BOCA

1. Get Test Data

Steps of BOCA

1. Get Test Data
2. Determine Impactful and Unimpactful Optimization Flags

Steps of BOCA

1. Get Test Data
2. Determine Impactful and Unimpactful Optimization Flags
3. Create New Candidates

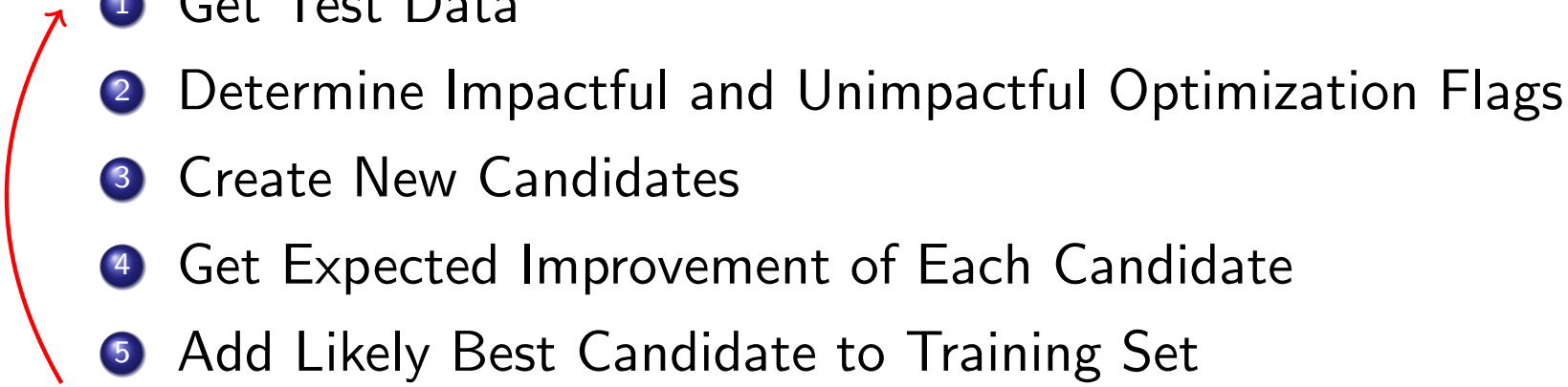
Steps of BOCA

1. Get Test Data
2. Determine Impactful and Unimpactful Optimization Flags
3. Create New Candidates
4. Get Expected Improvement of Each Candidate

Steps of BOCA

1. Get Test Data
2. Determine Impactful and Unimpactful Optimization Flags
3. Create New Candidates
4. Get Expected Improvement of Each Candidate
5. Add Likely Best Candidate to Training Set

Steps of BOCA



1. Get Test Data
2. Determine Impactful and Unimpactful Optimization Flags
3. Create New Candidates
4. Get Expected Improvement of Each Candidate
5. Add Likely Best Candidate to Training Set

Determining Impactful Optimizations

To find the impact of a specific flag [1], f:

$$Impact_f = \frac{\sum_{n=1}^{N} G(t_{nf})}{N} \tag{1}$$

$Impact_f$ = Impact of a flag f
N = number of decision trees which use f for splitting
$G(t_{nf})$ = Gini impurity of a n^{th} tree

Choosing New Candidates

Use a decay function to determine the total number of new candidates [1]:

$$C(i) = C_1 \cdot e^{-\frac{max(0, i-offset)^2}{2 \cdot \sigma^2}}, \sigma = -\frac{scale^2}{2 \cdot log(decay)} \quad (2)$$

$C(i)$ = Number of candidates for the i^{th} iteration
C_1 = Size of the initial training set
$offset$ = Initial reduction for iteration
$decay$ = rate at which C(i) decreases
$scale$ = constant multiplier for decay rate

Choosing New Candidates

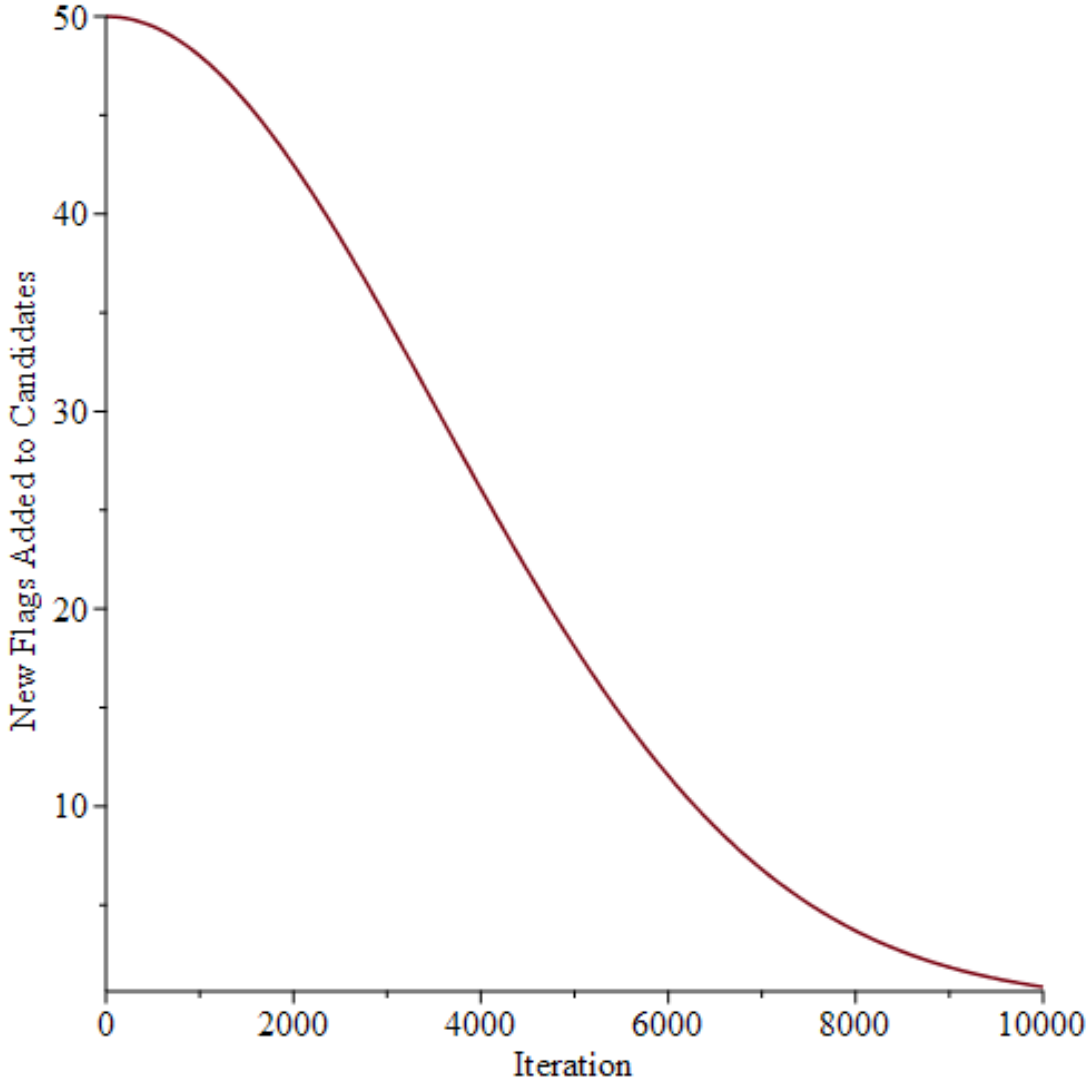


Figure: Decay function used in BOCA tests

Choosing New Candidates

1. Calculate expected improvement (EI) of each candidate.
2. Choose most promising candidate.
3. Add it to the training set.

Definition: Expected Improvement

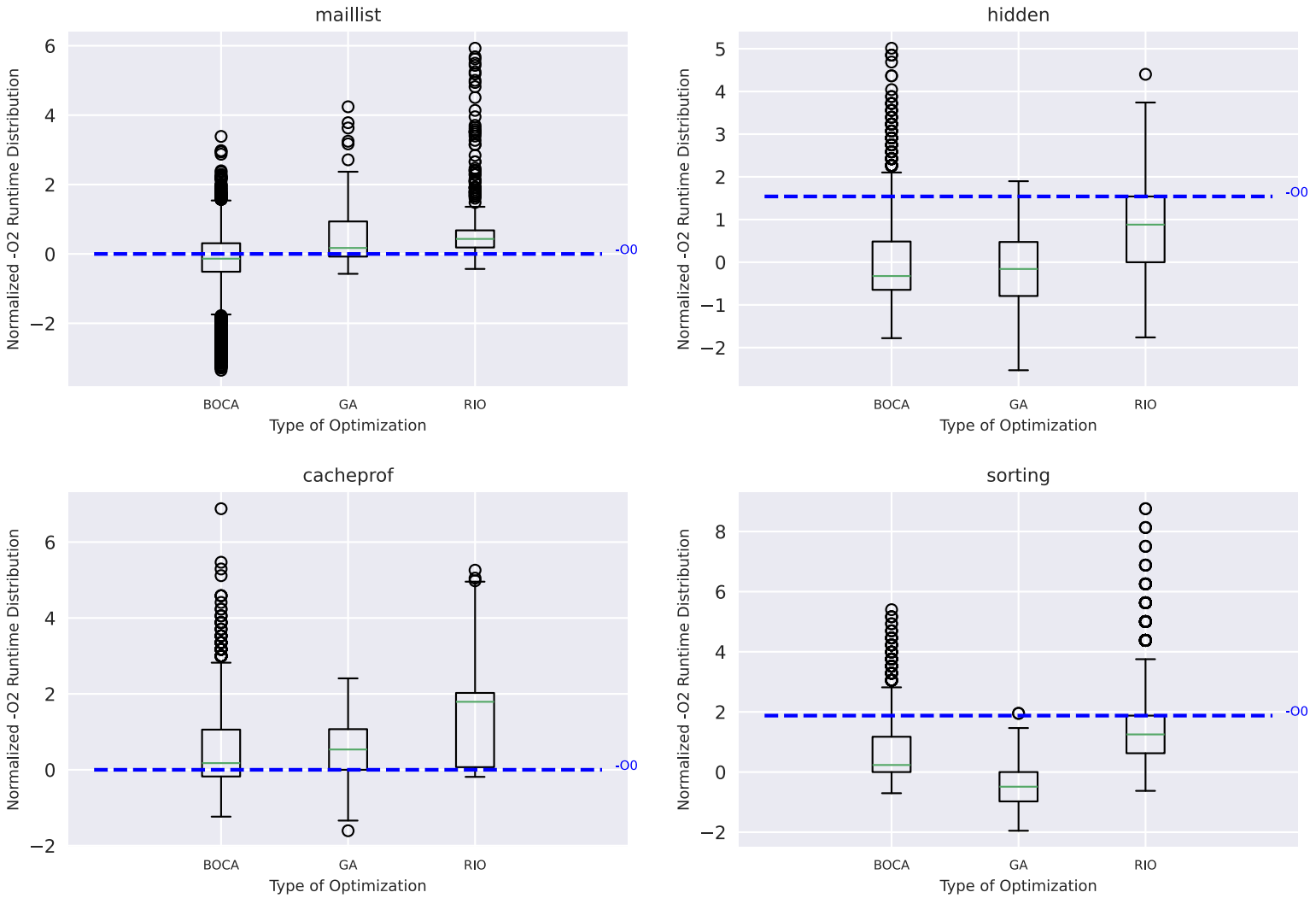
Expected Improvement (EI) is a widely-used Bayesian optimization algorithms. It is a greedy improvement-based heuristic that samples the point offering greatest expected improvement over the current best sampled point [1].

Best of Each Model

Program	Configuration	-O2	Optimal	RT Change	Avg. Change
cacheprof	BOCA	0.343	0.336	-1.449%	1.364%
	GA	0.437	0.431	-1.373%	0.407%
	RIO	0.439	0.431	-1.822%	12.943%
sorting	BOCA	0.207	0.204	-1.449%	1.364%
	GA	0.211	0.207	-1.896%	-0.474%
	RIO	0.211	0.207	-0.481%	1.106%
hidden	BOCA	0.482	0.471	-2.282%	0.009%
	GA	0.517	0.501	-3.095%	-0.226%
	RIO	0.521	0.506	-1.556%	0.719%
maillist	BOCA	0.646	0.646	-15.17%	-1.031%
	GA	0.535	0.520	-2.804%	2.371%
	RIO	0.527	0.520	-1.328%	2.059%

Table: Improvement over -O2 across each test program

O2 Score Distribution



Other Ways of Changing the Code

- Code changed depending on order optimizations are applied

Other Ways of Changing the Code

- Code changed depending on order optimizations are applied
- Use BOCA to find the optimal ordering!

Other Ways of Changing the Code

- Code changed depending on order optimizations are applied
- Use BOCA to find the optimal ordering!
- Profit

An Example

Function Inlining

```
int c = add(a, b); => int c = a + b;
```

Common Sub-Expression Elimination (cse)

```
int a = square(a) + square(a);
=>
int tmp = square(a); int a = tmp + tmp
```

Dumb.hs

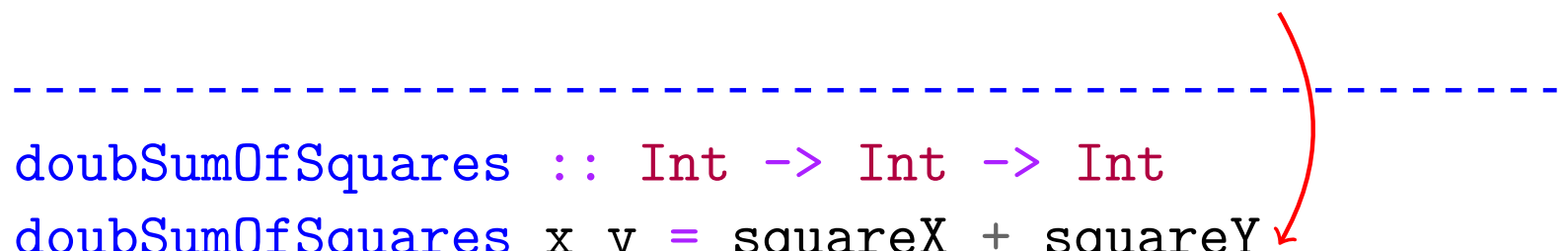
```
sumOfSquares :: Int -> Int -> Int
sumOfSquares x y = x*x + y*y

doubSumOfSquares :: Int -> Int -> Int
doubSumOfSquares x y = sumOfSquares x y
                     + sumOfSquares y x
```

Inlining before CSE

```
sumOfSquares :: Int -> Int -> Int
sumOfSquares x y = x*x + y*y

doubSumOfSquares :: Int -> Int -> Int
doubSumOfSquares x y = sumOfSquares x y
                     + sumOfSquares y x
```



```
doubSumOfSquares :: Int -> Int -> Int
doubSumOfSquares x y = squareX + squareY
                     + squareX + squareY
    where
        squareX = x*x
        squareY = y*y
```

CSE before Inlining

```
sumOfSquares :: Int -> Int -> Int
sumOfSquares x y = x*x + y*y

doubSumOfSquares :: Int -> Int -> Int
doubSumOfSquares x y = sumOfSquares x y
                     + sumOfSquares y x
```

```
doubSumOfSquares :: Int -> Int -> Int
doubSumOfSquares x y = sum + sum
    where
        sum = x*x + y*y
```

The Same?

```haskell
-- Inlining before CSE
doubSumOfSquares :: Int -> Int -> Int
doubSumOfSquares x y = squareX + squareY
                     + squareX + squareY
    where
        squareX = x*x
        squareY = y*y
-- CSE before Inlining
doubSumOfSquares :: Int -> Int -> Int
doubSumOfSquares x y = sum + sum
    where
        sum = x*x + y*y
```

Difference? Obviously!

The Phase Order Problem

Compiler Fun Fact:

The Phase Order Problem

Compiler Fun Fact:

- The compiler determines the order of optimization flags.

The Phase Order Problem

What if we COULD determine the order?

Haskell Code

```
core_todo = getPhaseOrder fString
 [
-- We want to do the static argument transform before full laziness as it
-- may expose extra opportunities to float things outwards. However, to fix
-- up the output of the transformation we need at do at least one simplify
-- after this before anything else
    ("static_args", runWhen static_args (CoreDoPasses [ simpl_gently, CoreDoStaticArgs ])),

    -- initial simplify: mk specialiser happy: minimum effort please
    ("presimplify", runWhen do_presimplify simpl_gently),

    -- Specialisation is best done before full laziness
    -- so that overloaded functions have all their dictionary lambdas manifest
    ("specialise", runWhen do_specialise CoreDoSpecialising),

    ("full_laziness_1", if full_laziness then
       CoreDoFloatOutwards FloatOutSwitches {
                                floatOutLambdas   = Just 0,
                                floatOutConstants = True,
                                floatOutOverSatApps = False,
                                floatToTopLevelOnly = False }
            -- Was: gentleFloatOutSwitches
            --
            -- I have no idea why, but not floating constants to
            -- top level is very bad in some cases.
            --
            -- Notably: p_ident in spectral/rewrite
            --          Changing from "gentle" to "constantsOnly"
            --          improved rewrite's allocation by 19%, and
```

Figure: Pipeline List

Haskell Code

```haskell
getPhaseOrder :: [Char] -> [(String, CoreToDo)] -> [CoreToDo]
getPhaseOrder fString coreToDoList
  = flattenTuples (orderTuples (listToNewList coreToDoList my_dictionary))
  where
    decodedString = map read (splitBy '|' fString) :: [Int]
    parsedTuples = parseTupleArray coreToDoList
    my_dictionary = fString2Dictionary decodedString parsedTuples createDictionary
```

Figure: getPhaseOrder function in Pipeline.hs

fString example: "0||1||2||3||4||5||6||7||8||9||10|| . . ."

Ideal vs. Real World

Kind of...

Ideal vs. Real World

Kind of...

- Some optimizations are not in a pipeline.

Ideal vs. Real World

Kind of...

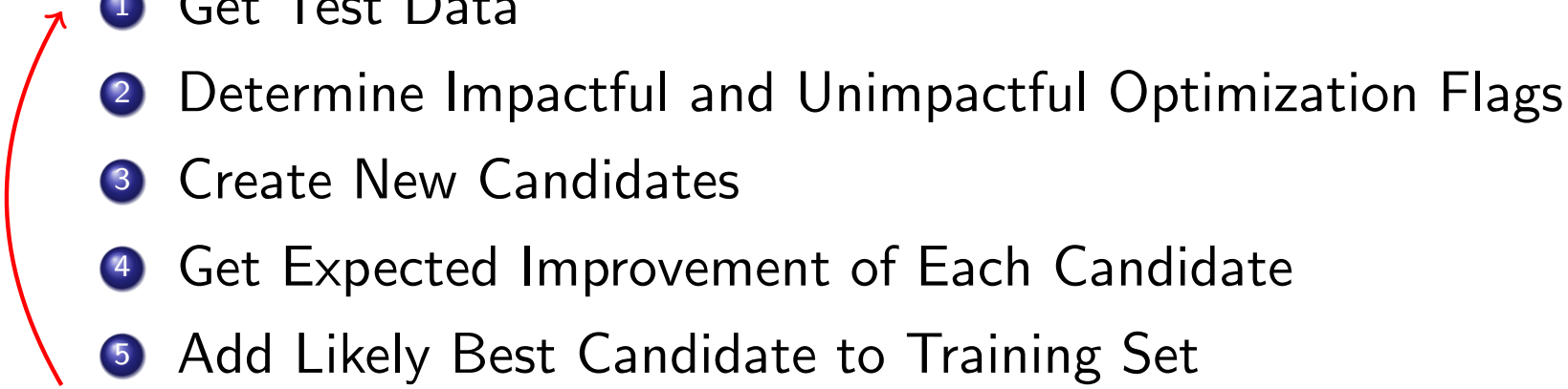
- Some optimizations are not in a pipeline.
- Some optimizations in the pipeline cannot be moved

Ideal vs. Real World

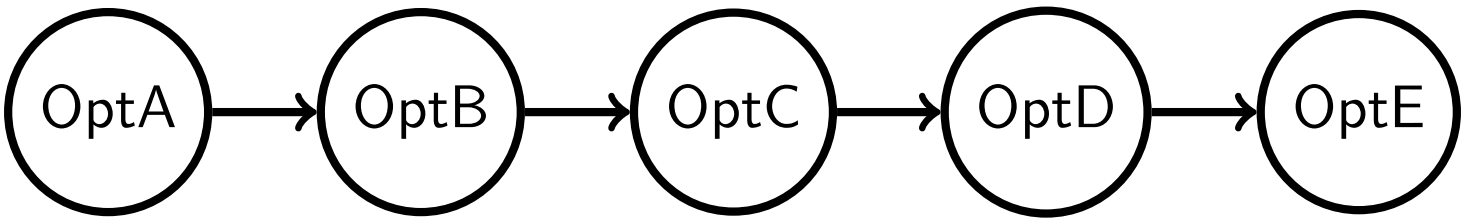
Kind of...

- Some optimizations are not in a pipeline.
- Some optimizations in the pipeline cannot be moved
- There might still be some buggy ones among the optimizations we think can be moved.

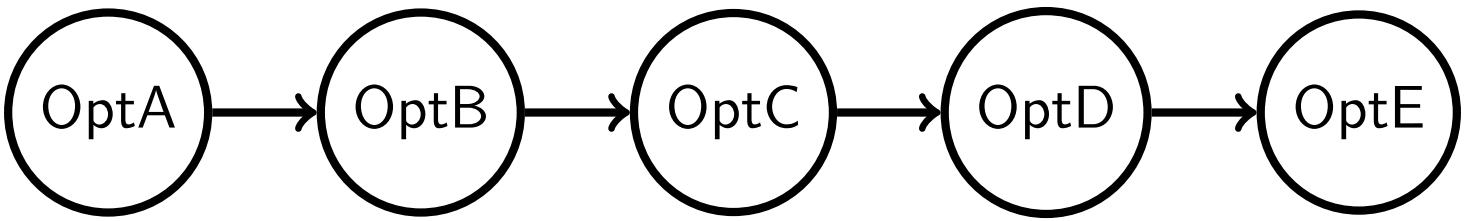
Let's Optimize the Phase-Order!



BOCA Problem: Default Ordering

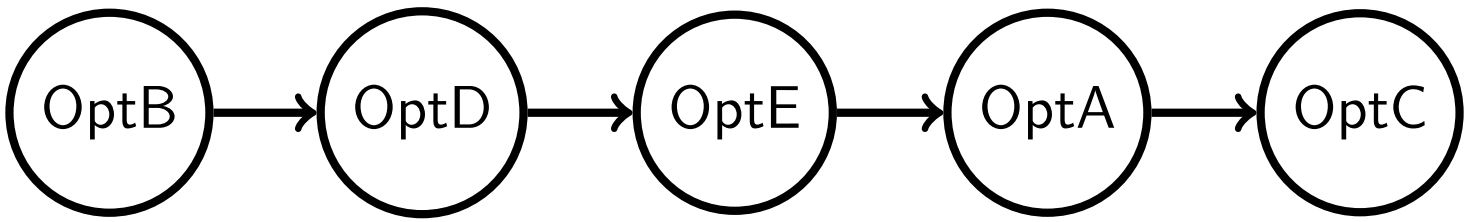


BOCA Problem: Default Ordering



Permutation = Graph

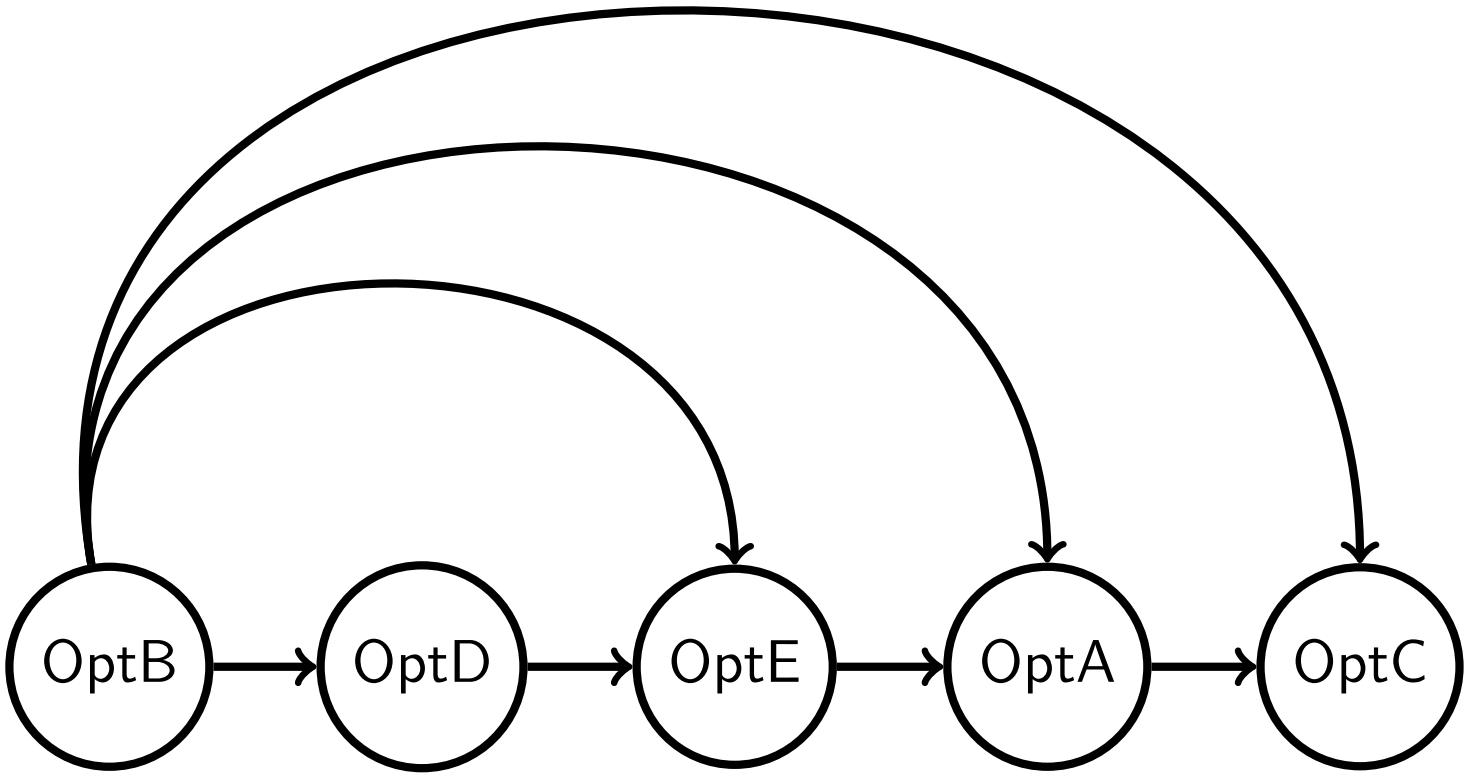
BOCA Problem: New Permutation



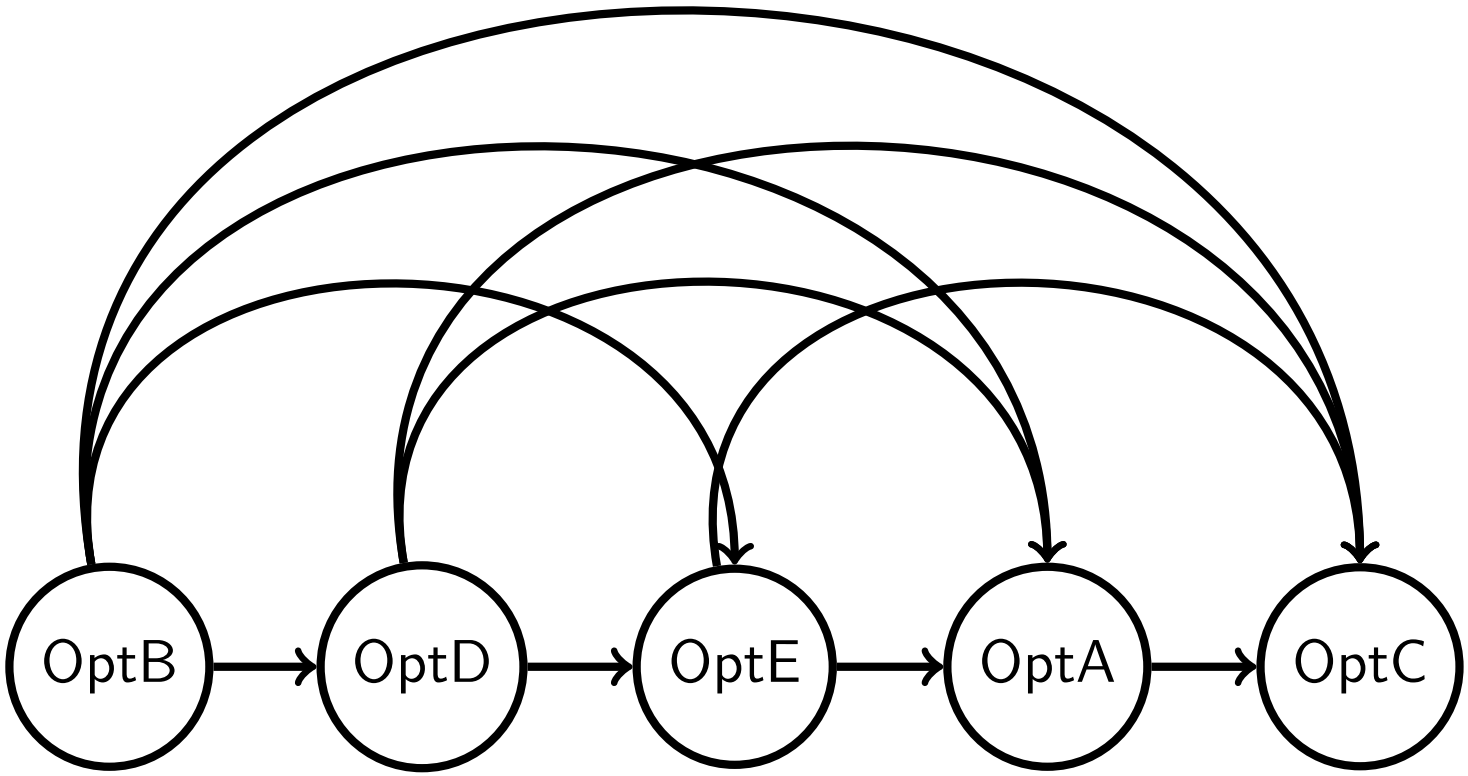
Assumption: A before B (A, B) = Change

Assumption: OptB before OptC *might* be more important than OptA before OptC

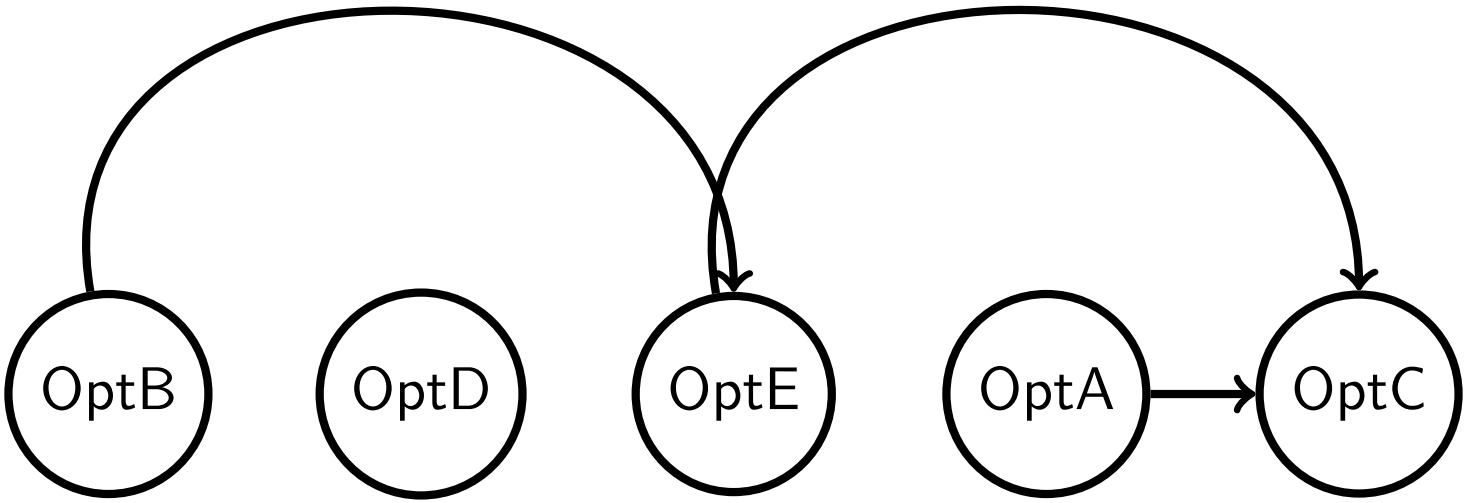
BOCA Problem: New Candidate Rules



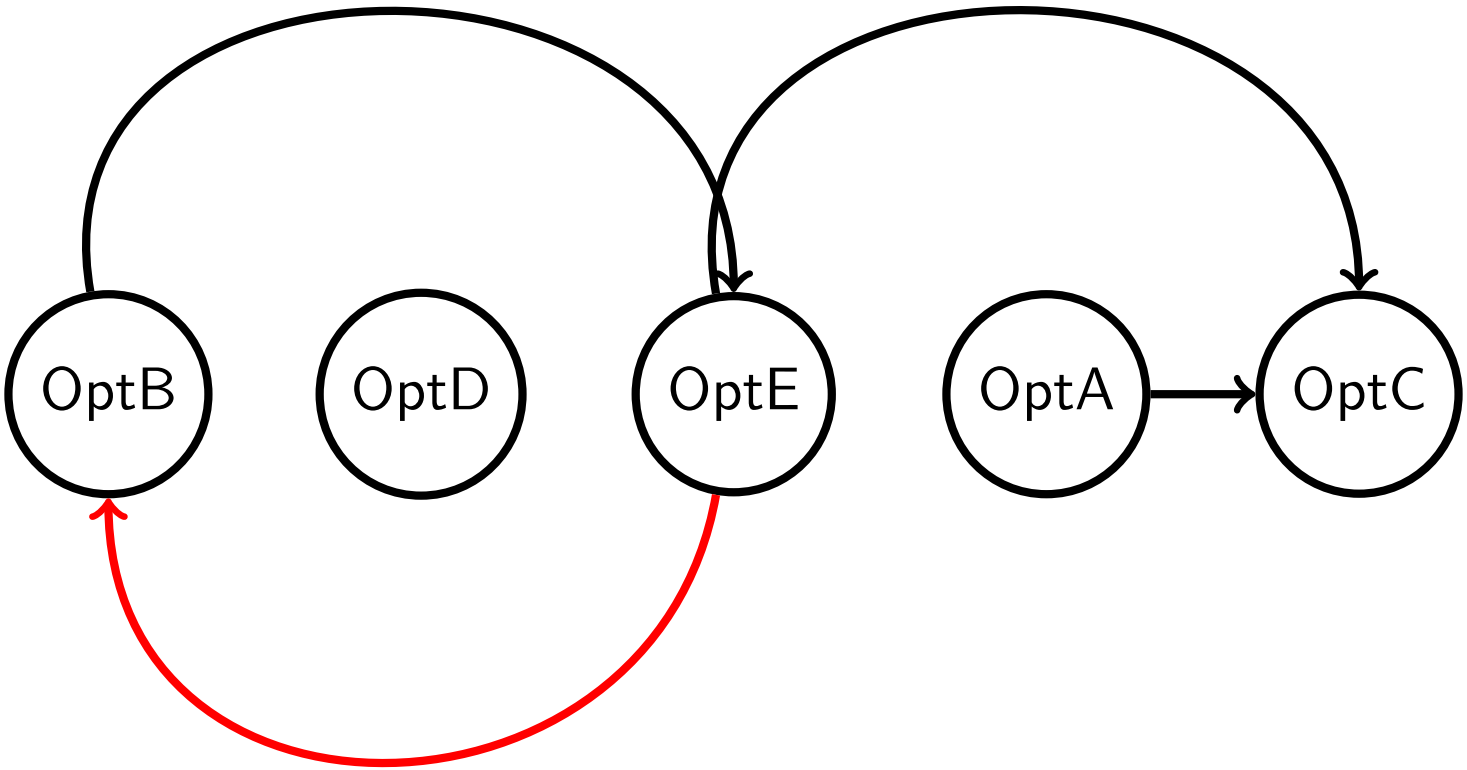
BOCA Problem: Default Ordering Rules



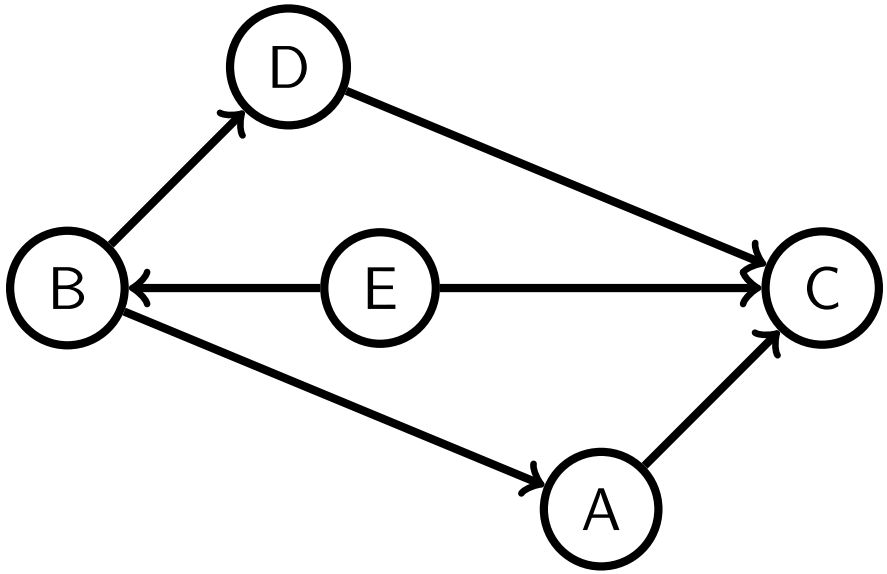
BOCA Problem: Find Impactful Rules



BOCA Problem: Add Unimpactful Rules



The Solution: DAGs Only!



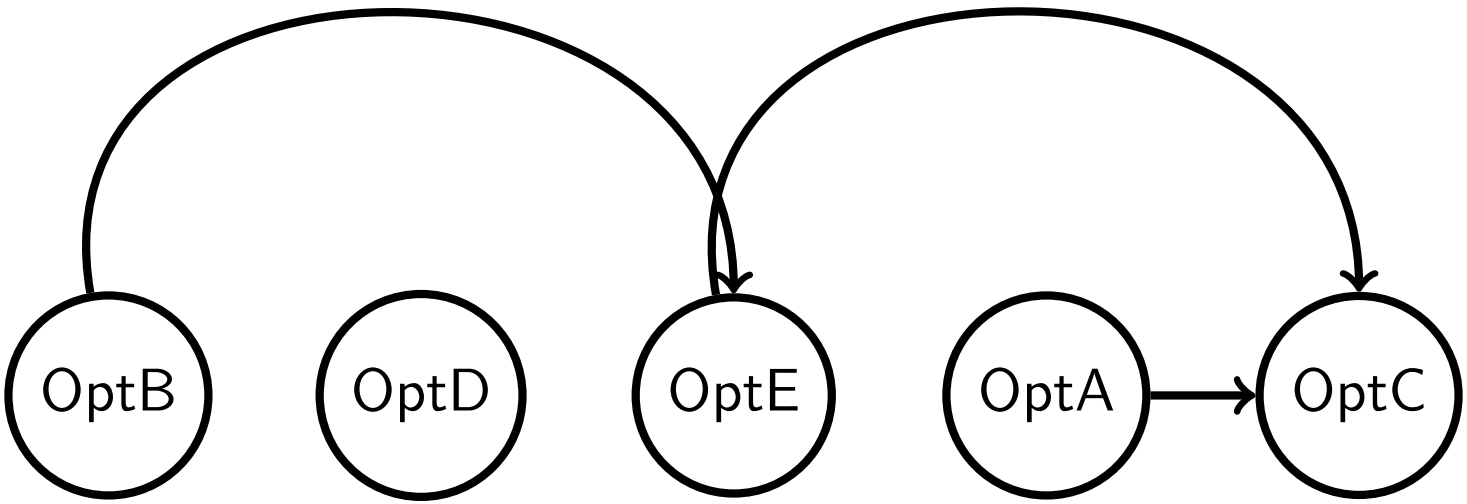
Directed Acyclic Graph (DAG)

A directed graph with no directed cycles. That is, it consists of vertices and edges (also called arcs), with each edge directed from one vertex to another, such that following those directions will never form a closed loop.

Creating New Candidates with BOCPA

- Sub-graph of a DAG is a DAG
- Choose most impactful rules
- Get a sub-graph of our impactful rules
- Topologically sort to get remaining rules (Probabilistic Kahn's Algorithm)

Creation after sort



One Possible Topological Sort: D, A, B, E, C

Can BOCPA discriminate?

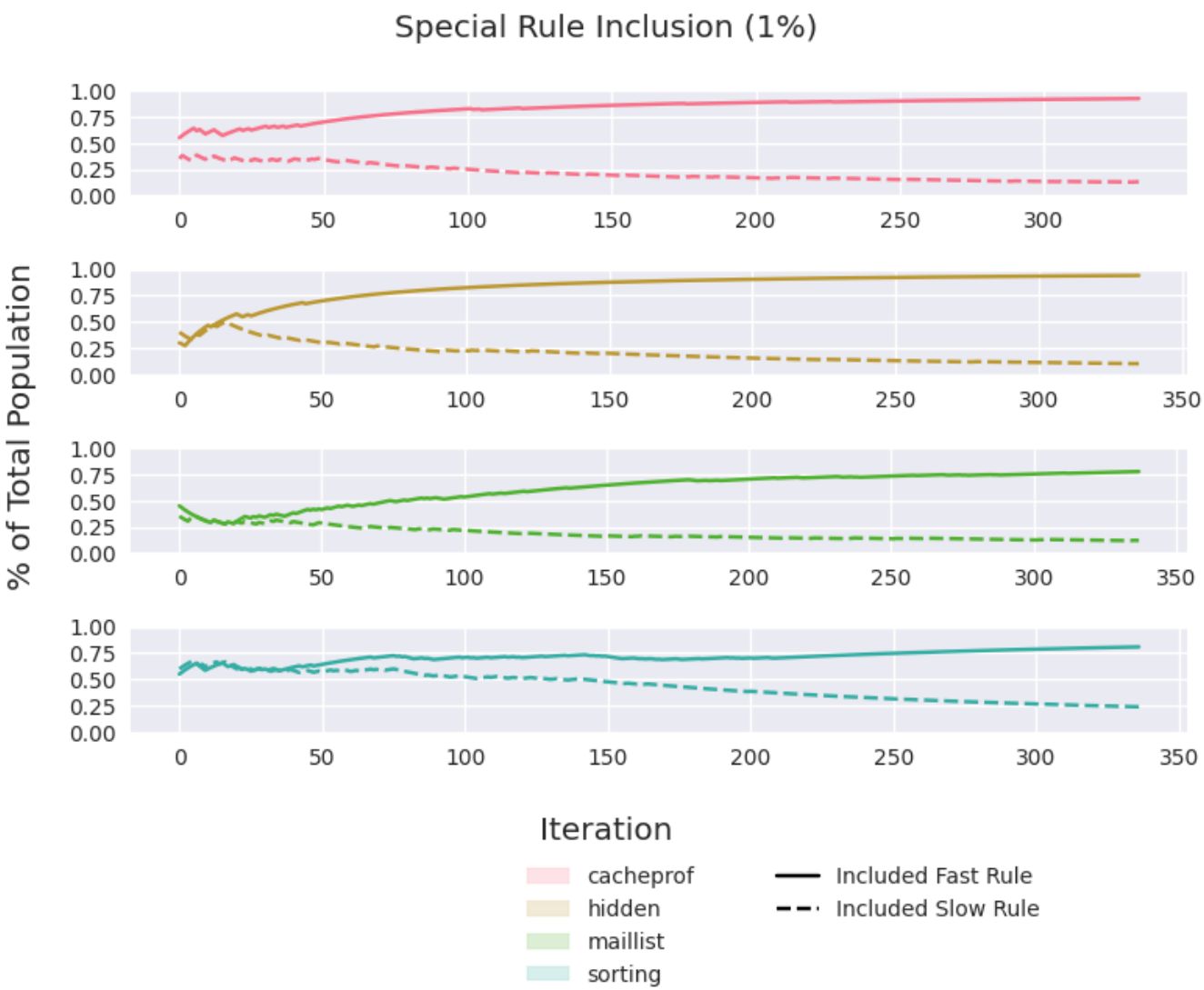


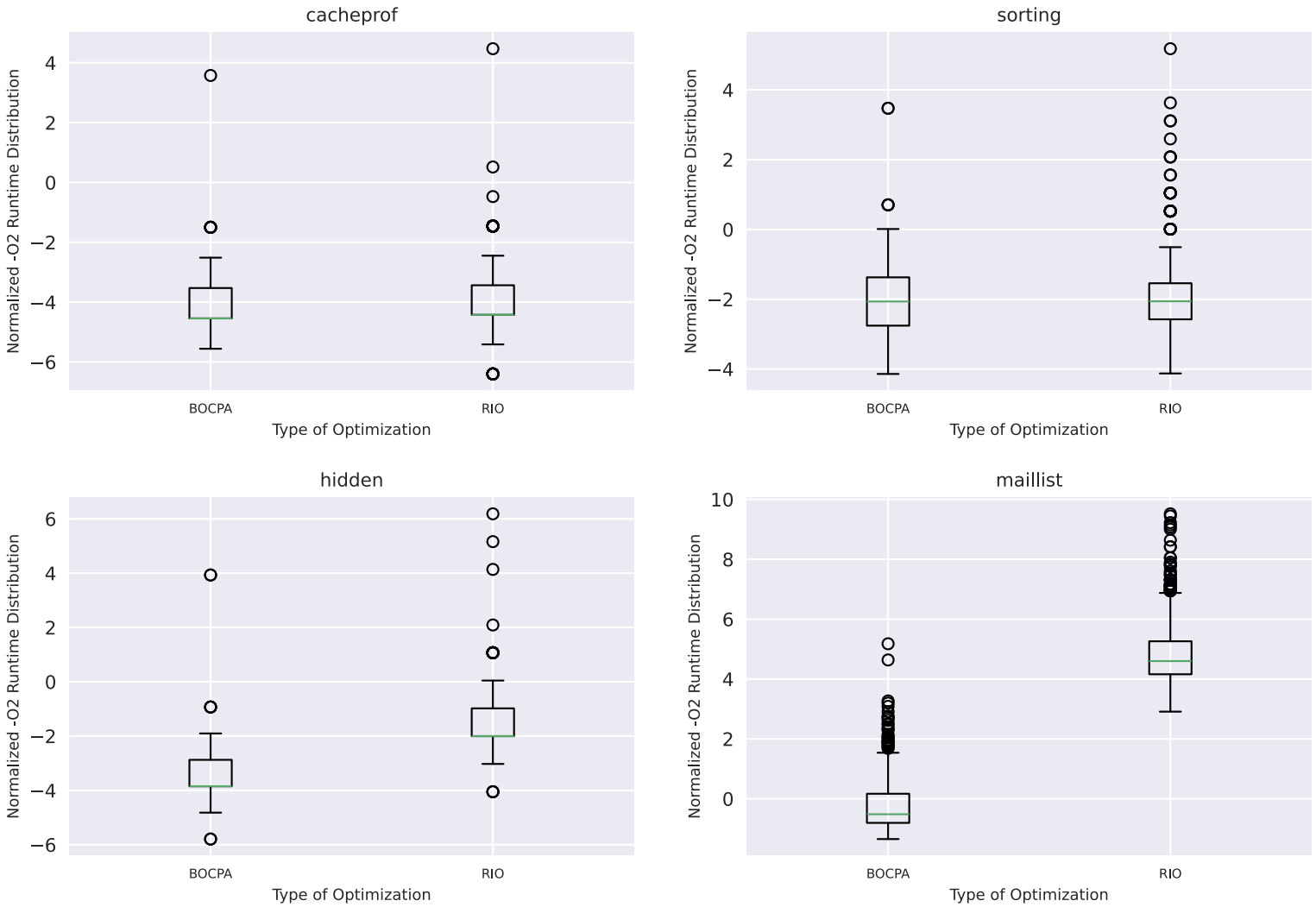
Figure: Discrimination capabilities of BOCPA up to 1%

Results

Program	Configuration	Optimal	Default	Change in RT	Avg. Change
cacheprof	BOCPA	0.113	0.118	-4.237%	-3.061%
	RIO	0.112	0.118	-5.085%	-2.999%
sorting	BOCPA	0.141	0.147	-4.082%	-2.189%
	RIO	0.139	0.147	-5.442%	-2.189%
hidden	BOCPA	0.122	0.128	-4.688%	-2.979%
	RIO	0.124	0.128	-3.125%	-1.284%
maillist	BOCPA	0.580	0.617	-5.997%	4.399%
	RIO	0.657	0.617	6.483%	10.697%

Table: BOCPA vs. RIO vs. Default Phase-Order Test Results

Default Ordering Graph



The End

References

[1] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209. IEEE, 2021.