Complexity-Guided Container Replacement Synthesis

Chengpeng Wang	Prism Group, HKUST
Peisen Yao	Prism Group, HKUST
Wensheng Tang	Prism Group, HKUST
Qingkai Shi	Ant Group -> Purdue University
Charles Zhang	Prism Group, HKUST

Container

- General-purpose abstract data type
  - Inserting, retrieving, removing and iterating over elements
  - E.g., ArrayList, HashMap, HashSet, etc

- A variety of implementations

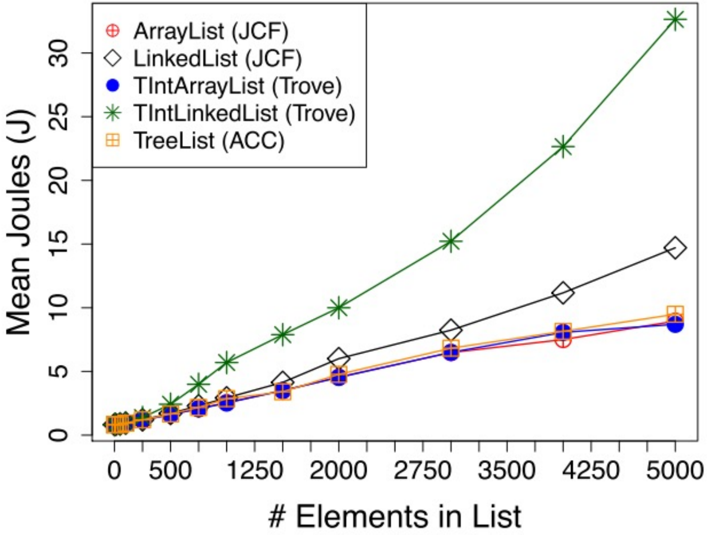
LEARN GUAVA
Java collections framework

GNU Trove
High performance collections for Java

Apache
Commons Collections

Performance Profile

- Resource consumption differs [Hasan, ICSE 16]



Random access:
LinkedList > ArrayList

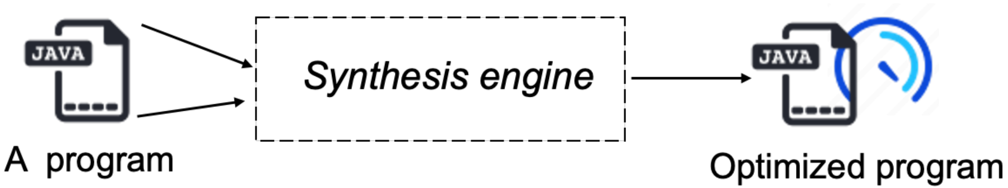
Container Selection

- Programmers are often
  - Unaware of how container objects are manipulated
    - Focus on specific modules of applications
  - Unaware of performance difference of container method calls
    - Unfamiliar with new implementations provided by libraries

Assist programmers in finding proper container types

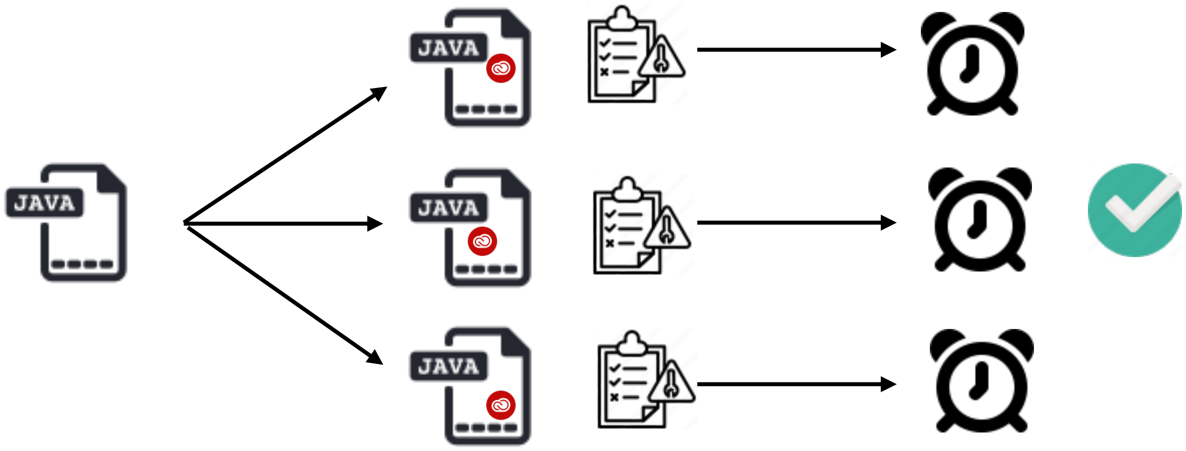
Our Aim

- Synthesize container replacements automatically to reduce the resource consumption
  - Container replacement
    - Container types in allocation statements
    - Container method calls
  - Resource consumption
    - Time, memory, CPU usage, energy
    - Focus on time cost but can be generalized



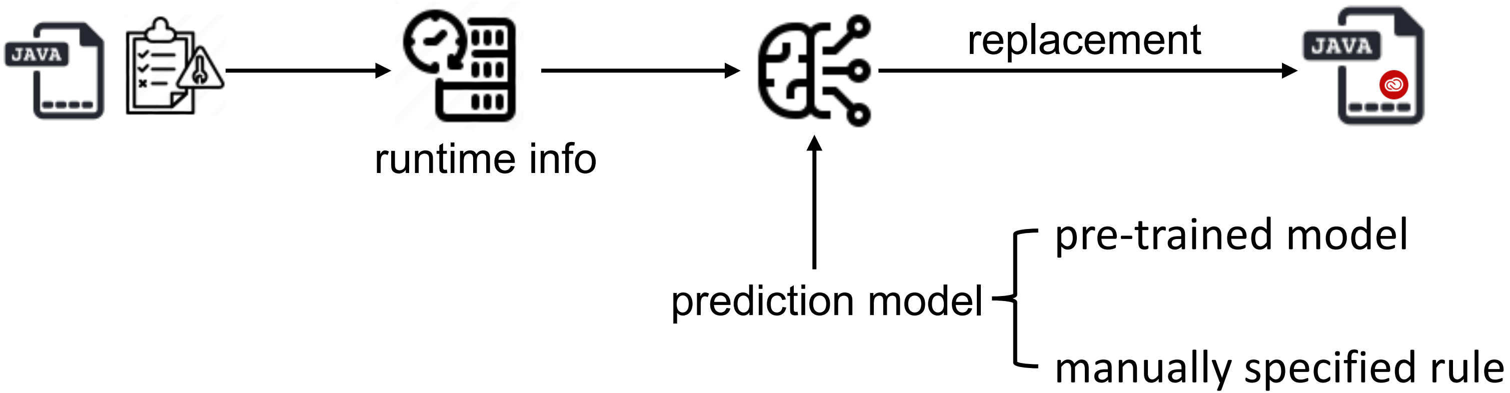
Existing Studies

- Solving an optimization problem [Basios, FSE 18] [Manotas, ICSE 14]
  - Enumerate container types
  - Monitor resource consumption when executing test cases
  - Find the optimal replacement to minimize the resource consumption



Existing Studies

- Solving a prediction problem [Jung, PLDI 11] [Vechev, PLDI 09]
  - Profile the program to obtain runtime info
  - Apply pre-trained model or pre-defined rules
  - Infer the container replacement



Limitations of Existing Approaches

- Huge overhead
  - Execute programs with test suites to profile dynamically
- Overfitting
  - Optimal replacements for specific inputs rather than general inputs
- Unsoundness
  - Unable to preserve behavioral equivalence, e.g., replace TreeSet with HashSet

Problem Formulation

- Replace container types and container methods in the program P and obtain a new program P', such that
  - (Behavioral equivalence) P and P' are behavioral equivalent
  - (Complexity superiority) P' consumes less time than P for a sufficiently large input

Behavioral Equivalence:
For any given input, P and P' always return the same value

Two Critical Goals

- Which container types are exchangeable to ensure behavioral equivalence?
- How to measure the performance of a container-manipulating program to check complexity superiority?

Goal I: Behavioral Equivalence

- Which container types are exchangeable to ensure behavioral equivalence?

```
public boolean foo1(String area) {
    ArrayList<String> l = new ArrayList<>();
    l.add("PL");   l.add("SE");
    boolean b1 = l.contains(area);
    return b1;
}
```

{LinkedList, HashSet, TreeSet, …}

whether a value is in the list

- Exchangeable container types achieve the original container usage intention.

Two Classes of Container Usage Intention

- Container-property queries
- Container-property modifiers

Class I: Container Property Queries

- Value ownership
    - ArrayList.contains(O), HashSet.contains(O)
- Index ownership
    - HashMap.containsKey(O)
- Index-value correlation
    - ArrayList.get(I), HashMap.get(O)
- Size
    - ArrayList.size(), HashSet.size()
- Insertion order
    - LinkedHashMap.iterator()
- Key order
    - TreeMap.firstKey(), TreeMap.lastKey()

Class II: Container Property Modifiers

- A container method can update container properties
  - Support querying container properties in other program locations

```
public boolean foo1(String area) {
    ArrayList<String> l = new ArrayList<>();
    l.add("PL");    l.add("SE");
    boolean b1 = l.contains(area);
    return b1;
}
```

ArrayList.add(O): increase the size by 1 / add a new value / add a new value at the end $\xrightarrow{\text{Modify}}$ size / value ownership / index-value correlation

ArrayList.add(O):			
	increase the size by 1	Modify →	size
	add a new value		value ownership
	add a new value at the end		index-value correlation

Method Semantic Specification

- Decompose method semantics into container-property queries and container-property modifiers

	queries	modifiers
ArrayList.contains(O)	{ isVal }	{ }
HashSet.contains(O)	{ isVal }	{ }
ArrayList.get(I)	{isCor}	{ }

isVal: query **value ownership**
isCor: query **index-value correlation**

Method Semantic Specification

- Decompose method semantics into container-property queries and container-property modifiers

	queries	modifiers
ArrayList.add(O)	$\{\ \}$	$\{\text{t-size}_{+}^{=1}\ \ \text{t-val}_{+}\ \ \text{t-cor}_{+}^{e}\}$
HashSet.add(O)	$\{\ \}$	$\{\text{t-size}_{+}^{\leq 1}\ \ \text{t-val}_{+}\}$

$\text{t-size}_{+}^{=1}$: increase the size by 1

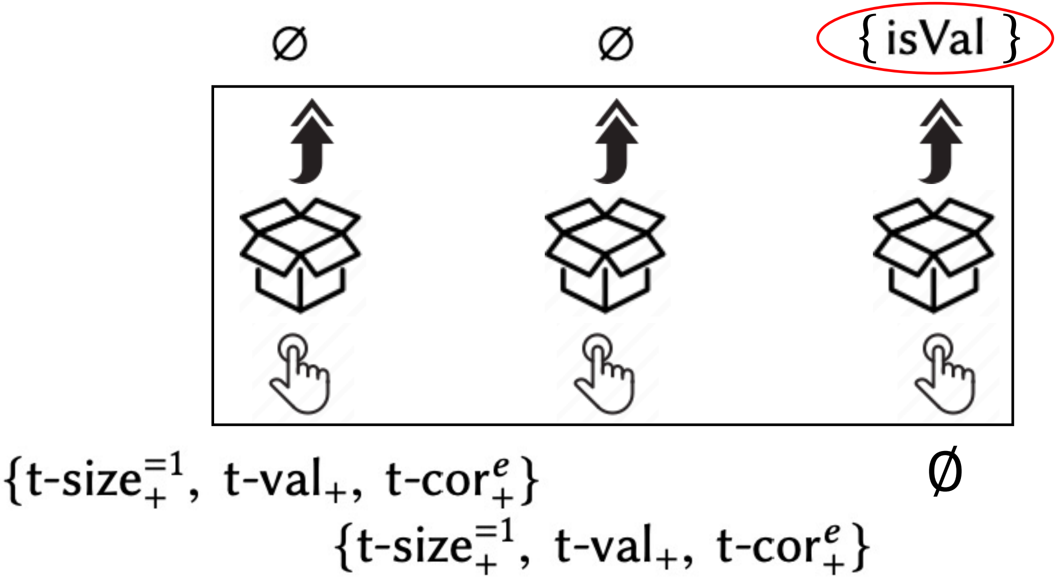
$\text{t-size}_{+}^{\leq 1}$: increase the size by at most 1

t-val_{+} : add a new value

t-cor_{+}^{e} : add a new value at the end

Key Idea: Achieve Original Usage Intention

- Support the original container-property queries
- Modify the queried container properties as the original ones

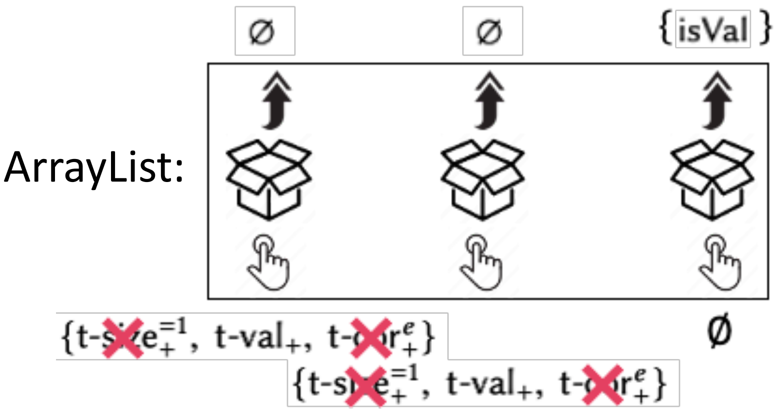


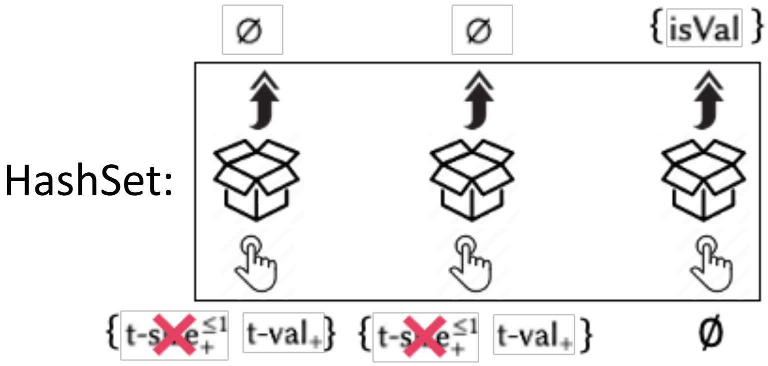

```
public boolean foo1(String area) {
    ArrayList<String> l = new ArrayList<>();
    l.add("PL");    l.add("SE");
    boolean b1 = l.contains(area);
    return b1;
}
```

original program

Key Idea: Achieve Original Usage Intention

- Support the original container-property queries
- Modify the queried container properties as the original ones





Key Idea: Achieve Original Usage Intention

- Support the original container-property queries
- Modify the queried container properties as the original ones

```
public boolean foo1(String area) {
    ArrayList<String> l = new ArrayList<>();
    l.add("PL");    l.add("SE");
    boolean b1 = l.contains(area);
    return b1;
}
```

original program

```
public boolean foo2(String area) {
    HashSet<String> s = new HashSet<>();
    s.add("PL");    s.add("SE");
    boolean b2 = s.contains(area);
    return b2;
}
```

new program

Guarantee behavioral equivalence

Ensuring Behavioral Equivalence

- Achieve the original usage intention with exchangeable container types
  - Support the original container-property queries
  - Modify the queried container properties as the original ones

Goal II: Complexity Superiority

- How to measure the performance of a container-manipulating program to check complexity superiority?

```
public boolean foo1(String area) {
    ArrayList<String> l = new ArrayList<>();
    l.add("PL");    l.add("SE");
    boolean b1 = l.contains(area);
    return b1;
}
```

```
public boolean foo2(String area) {
    HashSet<String> s = new HashSet<>();
    s.add("PL");    s.add("SE");
    boolean b2 = s.contains(area);
    return b2;
}
```

- Only measure the time costs of container method calls.

Method Complexity Specification

- Cost model CS
  - Complexity classes
    - Constant
    - Amortized constant
    - Logarithmic
    - Amortized logarithmic
    - Linear
    - Amortized linear
    - Super linear

time complexity of
container methods

  - Complexity functions

constant	amortized constant	logarithmic	amortized logarithmic	linear	amortized linear	super linear
$tc_1(n) = 1$	$tc_2(n)$	$tc_3(n) = log\ n$	$tc_4(n)$	$tc_5(n) = n$	$tc_6(n)$	$tc_7(n)$

Method Complexity Specification

- Cost model CS
  - Constant factor

Container	Method	Complexity Function	θ
ArrayList	add(O)	$tc_2(n)$	1
ArrayList	add(I, O)	$tc_2(n)$	2
ArrayList	contains(O)	$tc_5(n)$	1
ArrayList	get(I)	$tc_1(n)$	1
ArrayList	iterator()	$tc_1(n)$	1
ArrayList	remove(I)	$tc_1(n)$	2
ArrayList	size()	$tc_1(n)$	1

Container	Method	Complexity Function	θ
HashSet	add(O)	$tc_2(n)$	2
HashSet	contains(O)	$tc_1(n)$	1
HashSet	iterator()	$tc_1(n)$	1
HashSet	remove(O)	$tc_2(n)$	2
HashSet	size()	$tc_1(n)$	1

$tc_1(n)$: constant

$tc_2(n)$: amortized constant

$tc_5(n)$: linear

$$\mathrm{CS}\big(\mathrm{ArrayList.add(O)}\big) = 1 \cdot \mathrm{tc}_2(\mathrm{n})$$

$$\mathrm{CS}\big(\mathrm{HashSet.add(O)}\big) = 2 \cdot \mathrm{tc}_2(\mathrm{n})$$

Checking Complexity Superiority

- How to measure the performance of a container-manipulating program to check complexity superiority?

- Introduce *container complexity superiority*
    - For each container object o, S and S' are the sets of methods manipulating o in P and P', we need to ensure that

$$\sum_{f' \in S'} CS(f') \leq \sum_{f \in S} CS(f)$$

Key Idea: Container Complexity Superiority

- For each container object o, S and S’ are the sets of methods manipulating o in P and P’, we need to ensure that

$$\sum_{f' \in S'} CS(f') \leq \sum_{f \in S} CS(f)$$

```
public boolean foo1(String area) {
    ArrayList<String> l = new ArrayList<>();
    l.add(“PL”);   l.add(“SE”);
    boolean b1 = l.contains(area);
    return b1;
}
```

$2 \cdot tn_2(n) + tn_5(n)$

```
public boolean foo2(String area) {
    HashSet<String> s = new HashSet<>();
    s.add(“PL”);   s.add(“SE”);
    boolean b2 = s.contains(area);
    return b2;
}
```

$4 \cdot tn_2(n) + tn_1(n)$

$tc_1(n)$: constant $tc_2(n)$: amortized constant $tc_5(n)$: linear

Cres: Synthesizing Container Replacement

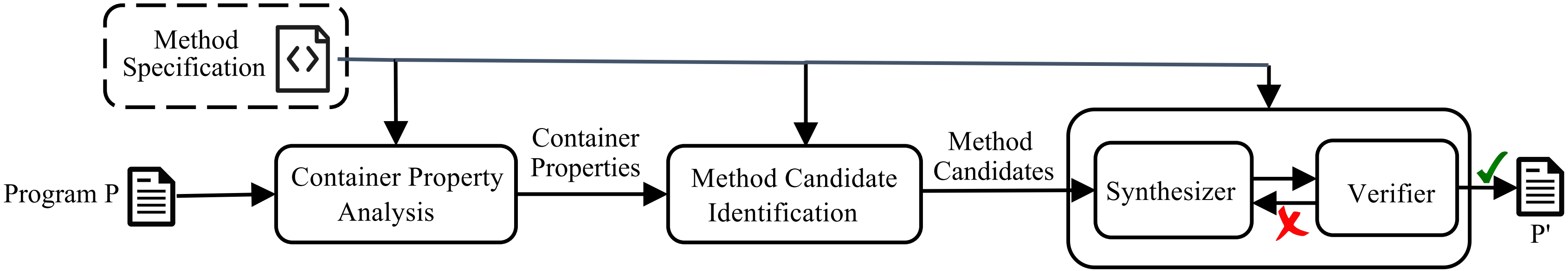
- Achieve the original intention of container usage to ensure behavioral equivalence
  - Support the original container-property queries
  - Modify the queried container properties as the original ones

- Achieve container complexity superiority to improve program efficiency
  - For each container object o, S and S' are the sets of methods manipulating o in P and P', we need to ensure that

$$\sum_{f' \in S'} CS(f') \leq \sum_{f \in S} CS(f)$$

Workflow of *Cres*

- Method semantic specification (queries, modifiers)
- Method complexity specification (complexity function, constant factor)



Stage I: Container Property Analysis

- Collect queried container properties for each container object via a sound points-to analysis

ArrayList object o2

{isVal}

ArrayList object o11

{size, isVal, isCor}

```
public List getUniqueResource(String curdir) {
  List uniqueResources = new ArrayList<IResource>();
  for (int i = 0; i < RESOURCE_NUM; i++) {
    IResource s = getResourceInCurrentDir(curdir, i);
    if (!uniqueResources.contains(s))
      uniqueResources.add(s);
  }
  return uniqueResources;
}
public List computeAllDirs() {
  List dirs = new ArrayList<Dir>();
  for (int i = 0; i < DIR_NUM; i++)
    dirs.add(getSubDirInWorkingDir(i));
  return dirs;
}
public void main() {
  List resources = getUniqueResource("/home/OOPSLA");
  List dirs = computeAllDirs();
  for (Dir dir : dirs) {
    if (resources.contains(dir))
      System.out.println("Accessible Resource");
  }
}
```

Stage II: Method Candidate Identification

- A method f’ is a candidate for the method call v=c.f(u) iff
  - f and f’ support the same container-property queries
  - f and f’ have the same modifiers on the queried container properties

ArrayList object o2

{isVal}

{LinkedList.contains,
ArrayList.contains,
HashSet.contains,
…}

{LinkedList.add,
ArrayList.add,
HashSet.add,
…}

```
public List getUniqueResource(String curdir) {
    List uniqueResources = new ArrayList<IResource>();
    for (int i = 0; i < RESOURCE_NUM; i++) {
        IResource s = getResourceInCurrentDir(curdir, i);
        if (!uniqueResources.contains(s))
            uniqueResources.add(s);
    }
    return uniqueResources;
}
public List computeAllDirs() {
    List dirs = new ArrayList<Dir>();
    for (int i = 0; i < DIR_NUM; i++)
        dirs.add(getSubDirInWorkingDir(i));
    return dirs;
}
public void main() {
    List resources = getUniqueResource("/home/OOPSLA");
    List dirs = computeAllDirs();
    for (Dir dir : dirs) {
        if (resources.contains(dir))
            System.out.println("Accessible Resource");
    }
}
```

{LinkedList.contains,
ArrayList.contains,
HashSet.contains,
…}

Stage III: Container Replacement Synthesis

- For each container object o, S and S' are the sets of methods manipulating o in P and P', we need to ensure that

$$\sum_{f' \in S'} CS(f') \leq \sum_{f \in S} CS(f)$$

HashSet

ArrayList object o2

{isVal}

```
public List getUniqueResource(String curdir) {
           uniqueResources = new           <IResource>();
  for (int i = 0; i < RESOURCE_NUM; i++) {
    IResource s = getResourceInCurrentDir(curdir, i);
    if (!uniqueResources.          (s))
      uniqueResources.    (s);
  }
  return uniqueResources;
}
public List computeAllDirs() {
  List dirs = new ArrayList<Dir>();
  for (int i = 0; i < DIR_NUM; i++)
    dirs.add(getSubDirInWorkingDir(i));
  return dirs;
}
public void main() {
  List resources = getUniqueResource("/home/OOPSLA");
  List dirs = computeAllDirs();
  for (Dir dir : dirs) {
    if (resources.          (dir))
      System.out.println("Accessible Resource");
  }
}
```

{LinkedList.contains,
ArrayList.contains,
HashSet.contains,
...}

{LinkedList.add,
ArrayList.add,
HashSet.add,
...}

{LinkedList.contains,
ArrayList.contains,
HashSet.contains,
...}

2 * CS(HashSet.contains) + CS(HashSet.add) is minimal

Theoretical Results

- Theorem 1: The new program P' is the behavioral equivalent to the original program P.

- Theorem 2: The new program P' has container complexity superiority over the original program P.

- Theorem 3: The time complexity of the algorithm is $O(|S_a| \cdot |S_c|)$ for given container types.
  - S_a and S_c contain container allocation statements and container method calls, respectively.

Implementation of Cres

- Implement *Cres* based on *Pinpoint* [Shi, PLDI 18]
  - Flow-sensitive points-to analysis for container property analysis

- Analyze containers in Java Collection Framework
  - List: ArrayList, LinkedList
  - Set: HashSet, LinkedHashSet, TreeSet
  - Map: HashMap, LinkedHashMap, TreeMap

Research Questions

- **RQ1: Effectiveness**
  - Performance improvement brought by *Cres*
- **RQ2: Replacement patterns**
  - Kinds and numbers of replacements *Cres* synthesize
- **RQ3: Overhead**
  - The time and space costs of *Cres*

Evaluation: Effectiveness

- What is the improvement *Cres* achieves for real-world programs?

Project	Description	Size (KLoC)	Medium (%)	95% CI (%)
bootique	Microservice platform	18.6	4.5	[4.4, 4.6]
mapper	Server application	22.4	7.3	[7.0, 7.6]
incubator-eventmesh	Eventing infrastructure	24.9	4.1	[3.9, 4.3]
google-http-java-client	Web client	25.2	27.1	[25.9, 28.3]
light-4j	Microservice platform	44.3	5.2	[5.0, 5.4]
roller	Server application	54.4	9.5	[9.2, 9.8]
IginX	Data management system	68.1	3.5	[3.4, 3.6]
sofa-rpc	RPC framework	76.4	3.7	[3.4, 4.0]
Glowstone	Server application	85.6	13.1	[12.9, 13.3]
dolphinscheduler	Eventing infrastructure	89.5	5.3	[5.1, 5.5]
dubbo	RPC framework	196.5	7.5	[7.2, 7.8]
iotdb	Data management system	384.2	6.3	[6.2, 6.4]
			8.1	[7.8, 8.4]

Speedup: On average 8.1%, up to 27.1%

Evaluation: Replacement Pattern

- Which kinds of container replacements does *Cres* synthesize?

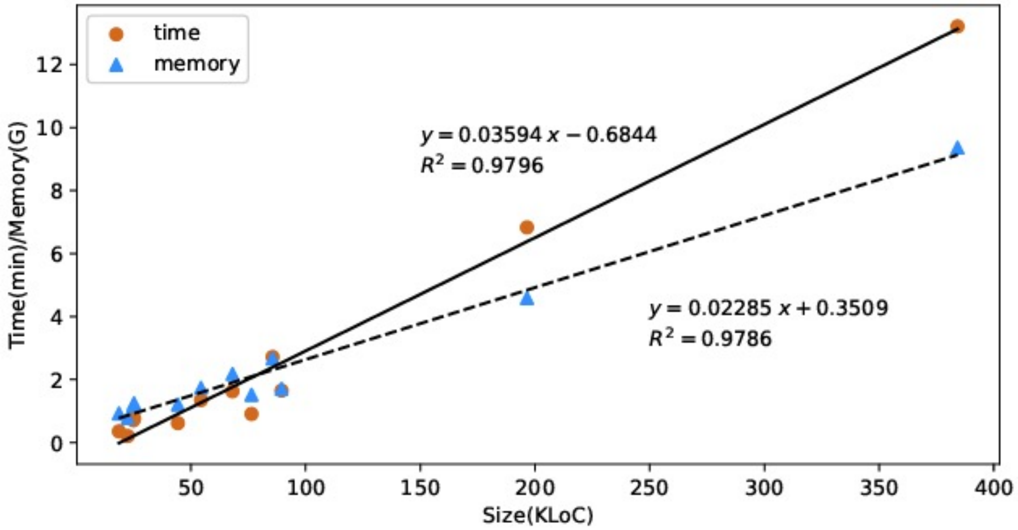
Project	#Conf/#Total	#R1	#R2	#R3	#R4	#R5	#R6
bootique	0/4			4			
mapper	0/6		5		1		
incubator-eventmesh	19/19	1	16	2			
google-http-java-client	0/4		4				
light-4j	0/5		2	3			
roller	0/6			5	1		
IginX	11/11		9		1		1
sofa-rpc	12/12		5			2	5
Glowstone	0/11		6	3	1		1
dolphinscheduler	7/7		6	1			
dubbo	12/12	1	3	1		2	5
iotdb	10/10	2	1	6			1
	71/107	4	57	25	4	4	13

R1: LinkedList⇒ArrayList R2: ArrayList⇒LinkedList
R3: ArrayList⇒HashSet R4: TreeMap⇒HashMap
R5: LinkedHashMap⇒HashMap R6: LinkedHashSet⇒HashSet

71 confirmed replacements

Evaluation: Time and Memory Costs

- What are the time and memory costs of *Cres*?



linear scalability

~ 14 minutes analyzing 384.2 KLoC

Interesting Findings

- Equipped with flow-insensitive pointer analysis, Cres synthesizes 74 container replacements out of 107 replacements.
  - Miss several optimization opportunities due to the imprecision of container property analysis

Interesting Findings

- Using randomly generated constant factors in the method complexity specification does not affect the result as long as they conform to a specific order.

Container	Method	Complexity Function	θ
HashSet	add(O)	$tc_2(n)$	2
HashSet	contains(O)	$tc_1(n)$	1
HashSet	iterator()	$tc_1(n)$	1
HashSet	remove(O)	$tc_2(n)$	2
HashSet	size()	$tc_1(n)$	1

θ_1

θ_3

Container	Method	Complexity Function	θ
LinkedHashSet	add(O)	$tc_2(n)$	3
LinkedHashSet	contains(O)	$tc_1(n)$	1
LinkedHashSet	iterator()	$tc_1(n)$	1
LinkedHashSet	remove(O)	$tc_2(n)$	3
LinkedHashSet	size()	$tc_1(n)$	1

θ_2

θ_4

HashSet.add(O) < LinkedHashSet.add(O) $\theta_1 < \theta_2$

HashSet.remove(O) < LinkedHashSet.remove(O) $\theta_3 < \theta_4$

Drawbacks

- Container complexity superiority does not imply complexity superiority.

$$\sum_{f' \in S'} CS(f') \leq \sum_{f \in S} CS(f) \quad \not\Longrightarrow \quad \text{Time complexity of P'} \leq \text{Time complexity of P}$$

  - Complexity analysis/WCET analysis are impractical for real-world programs.
  - Need a precise and computable complexity guidance

Drawbacks

- Unaware of usage intention of loops
  - Example from IoTDB: *pageReader* is a LinkedList object

```
public Point retrieveValidLastPoint(int n) {
  List<IChunkMetadata> seqDataList = new LinkedList<>();
  for (int i = 0; i < n; i++)
    seqDataList.add(getDataFromDevice());
  for (int i = seqDataList.size() - 1; i >= 0; i--) {
    Point lastPoint = getChunkLastPoint(seqDataList.get(i));
    if (lastPoint.getValue() != null)
      return lastPoint;
  }
  return null;
}
```

Cres: LinkedList => ArrayList

Optimal solution: use iterators

Conclusions

- A new program abstraction with
  - Container property & Method semantic specification
  - Cost model & Method complexity specification
- An efficient and sound synthesis algorithm *Cres*
  - Ensuring behavioral equivalence with container property analysis
  - Improving program efficiency with complexity-guided synthesis

Thank you for your listening!