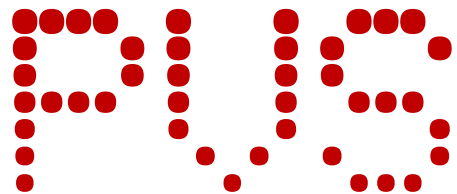


CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection

Diego Costa, Artur Andrzejak

Heidelberg University



UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

International Symposium on Code Generation and Optimization (CGO 18)

Feb 26th 2018, Vienna Austria

CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection

Diego Costa, Artur Andrzejak

Heidelberg University

Published as:

Diego Costa and Artur Andrzejak. 2018. CollectionSwitch: a framework for efficient and dynamic collection selection. In Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018). Vienna, Austria — February 24 - 28, 2018

Paper/slides available at: <https://pvs.ifi.uni-heidelberg.de/publications/>

Programs = Data Structures + Algorithms

Niklaus Wirth (1984)

Collections

- In Java, the **Collections** framework provides a reusable set of data structures
 - Widely used and well tested
 - One Interface → multiple implementations
- Developers rarely select/tune their collections [Cha++11]
 - Top-4 most used implementations are selected **95%** of the cases [Cos++17]
 - Only **20%** of the ArrayList instantiations specify the initial capacity [Cos++17]

Performance Impact of Collections

Inefficient selection of collections as the **main cause** of runtime bloat

Execution Time

+17% Improv.

Configuration of one
HashMap alloc-site
[LS09]

Memory Usage

+54% Improv.

Use of ArrayMaps
instead of HashMaps
[OME09]

Energy Consumption

+38% Improv.

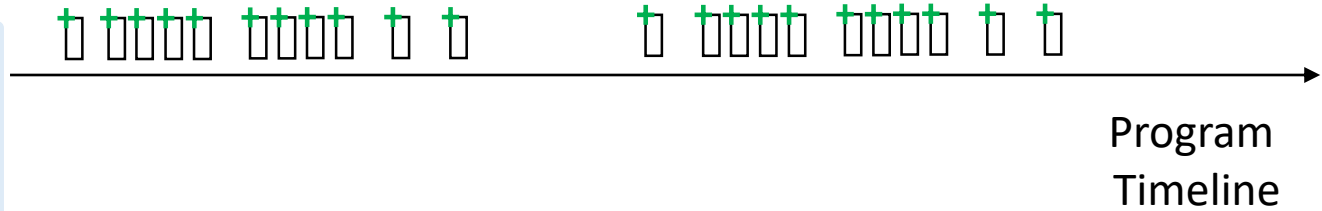
Use of ArrayList
instead of LinkedList
[Sam++16]

How to better identify and fix such performance inefficiencies?

Motivational Scenario

```
List<T> myList = new ArrayList<>();  
  
for(T elem : collection) {  
    if(!myList.contains(elem)){  
        myList.add(elem);  
    }  
}
```

```
List<T> myList = ctx.createList();  
  
for(T elem : collection) {  
    if(!myList.contains(elem)){  
        myList.add(elem);  
    }  
}
```



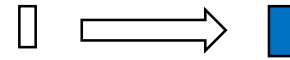
Motivational Scenario

```
List<T> myList = new ArrayList<>();  
  
for(T elem : collection) {  
    if(!myList.contains(elem)){  
        myList.add(elem);  
    }  
}
```

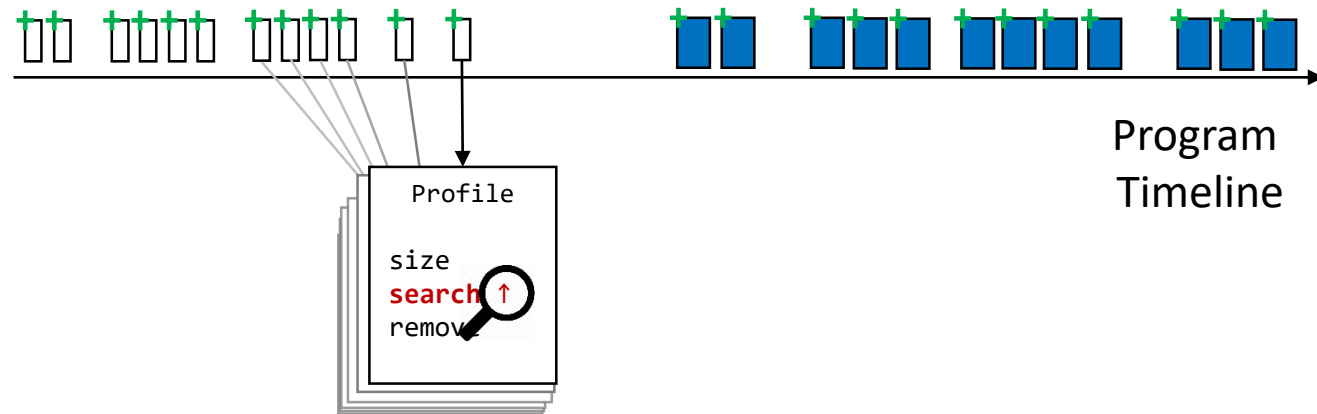
```
List<T> myList = ctx.createList();  
  
for(T elem : collection) {  
    if(!myList.contains(elem)){  
        myList.add(elem);  
    }  
}
```

It is possible to **find** and **switch**
the collection type to a more
suitable variant?

new ArrayList<>()



new ArrayList<>()



Profile Instances Behavior

Exemplary Results

- The DaCapo benchmark of Lucene and Avrora
 - Few allocation sites generate millions of collection instances
- **Lucene:** By augmenting 12 allocation sites with our adaptive behavior
 - Reduce execution time by **15%**
- **Avrora:** By augmenting 10 allocation sites with our adaptive behavior
 - Reduce peak of memory consumption by **10%**

CollectionSwitch

A framework for Dynamic Adaptation of Java Collections

Combines two techniques:

1. Adaptive Allocation-Site

- **Profiles** collection instances
- **Searches** for a better variant
- **Switches** future instantiations to the best variant type

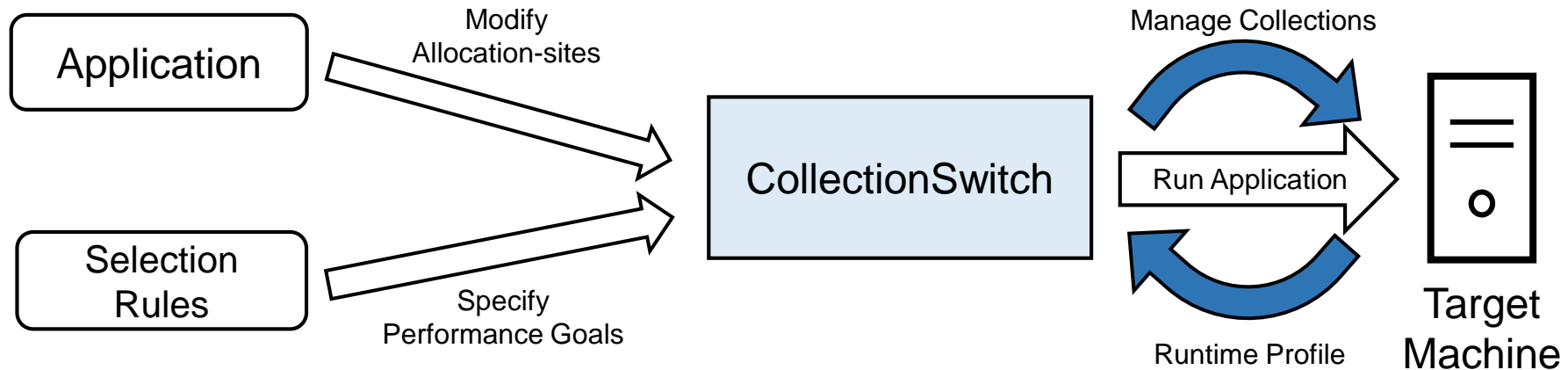
2. Adaptive Collections

- Instances that **switch themselves** to the appropriate implementation

Framework Overview

A) How to Enable Adaptive Collections?

C) How to Find a Better Implementation?



B) How to Define the Performance Goals?

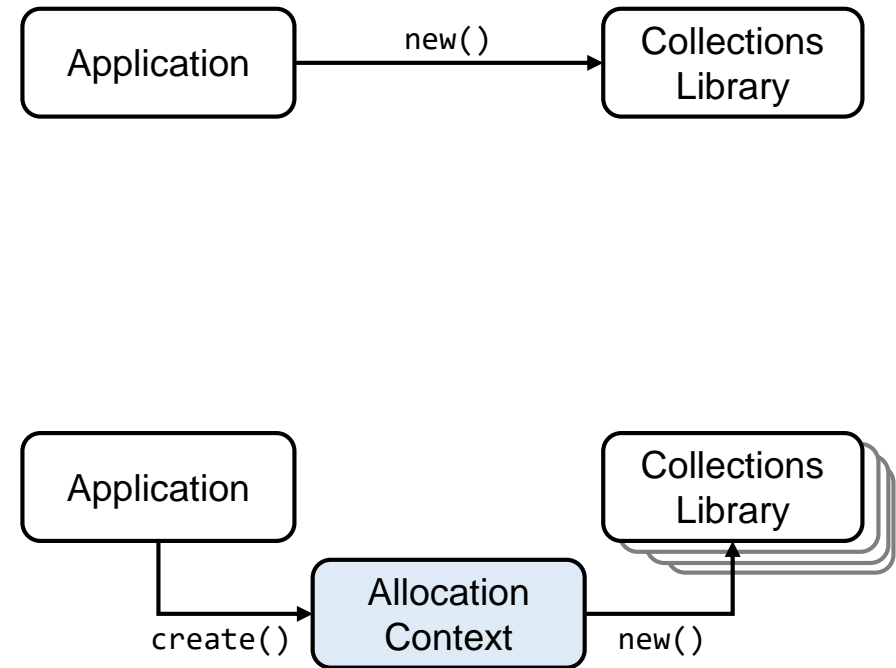
A) How to Enable Adaptive Collections

- Using CollectionSwitch in your project

```
// Original Allocation-Site  
List<T> myList = new ArrayList<>();
```

```
// Using CollectionSwitch  
static ListContext ctx = Switch.createContext(ARRAY);  
List<T> myList = ctx.createList();
```

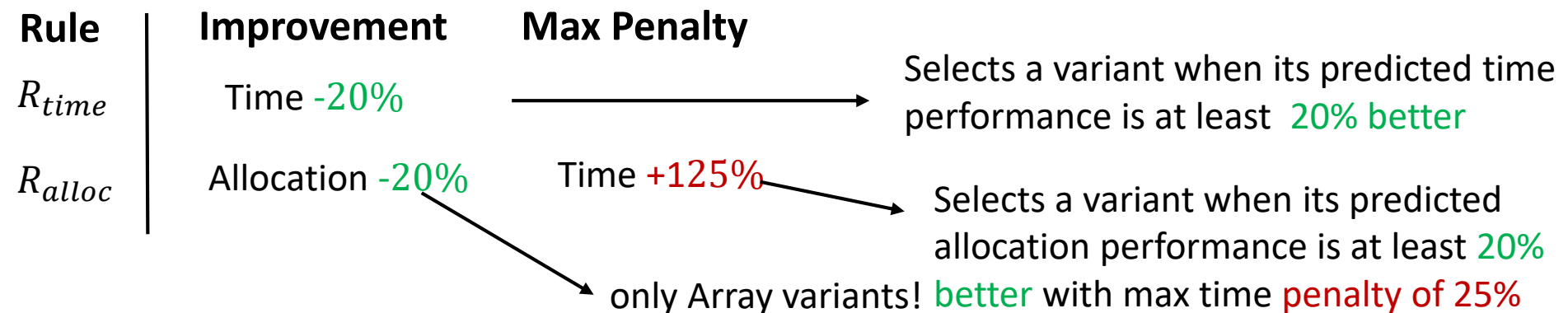
Allows **same-interface** transformations



B) How to define the Performance Goals?

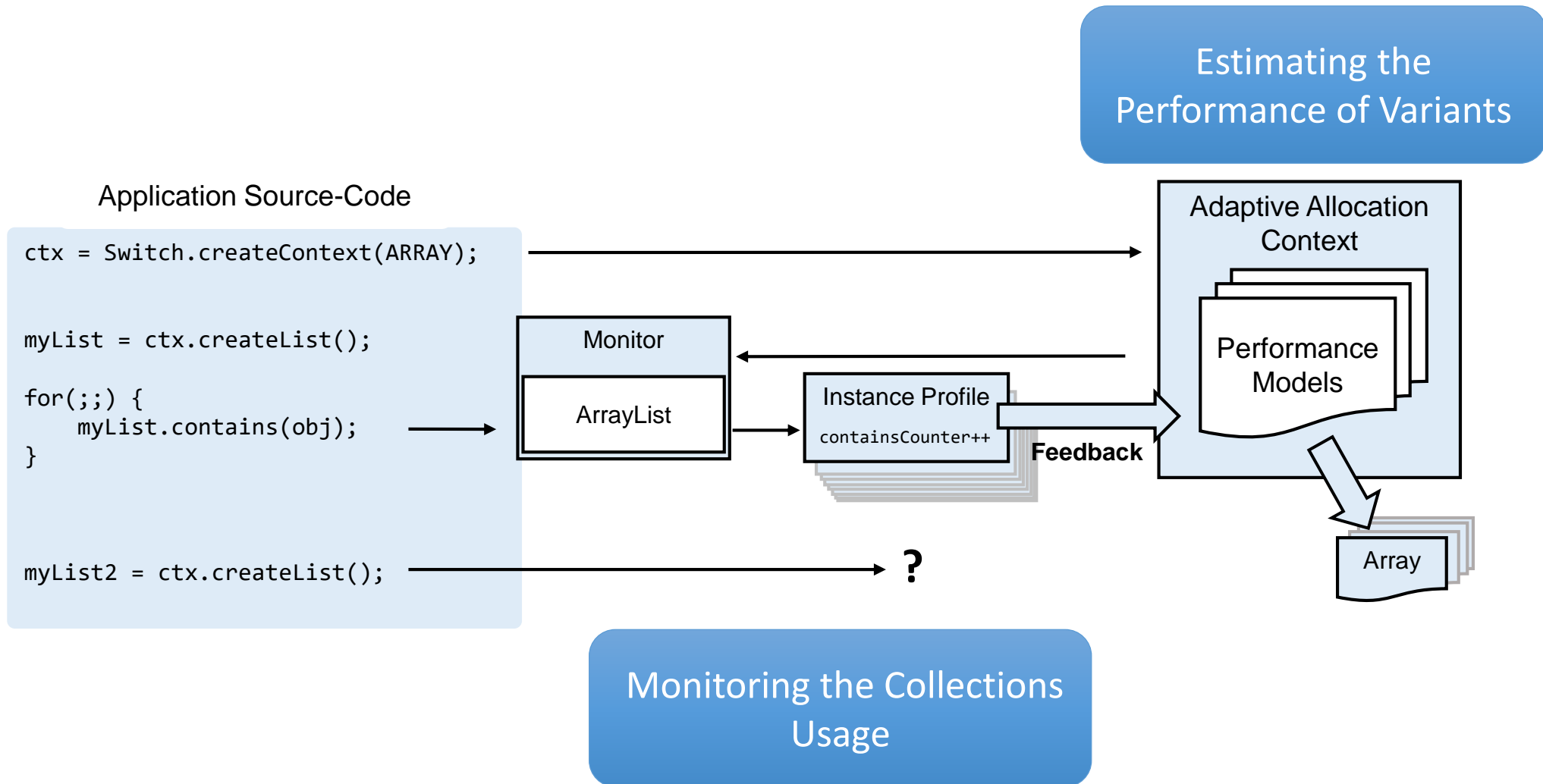
Configurable Selection Rules

- Space and time trade-offs
- Developers input the **threshold** for selecting a different variant

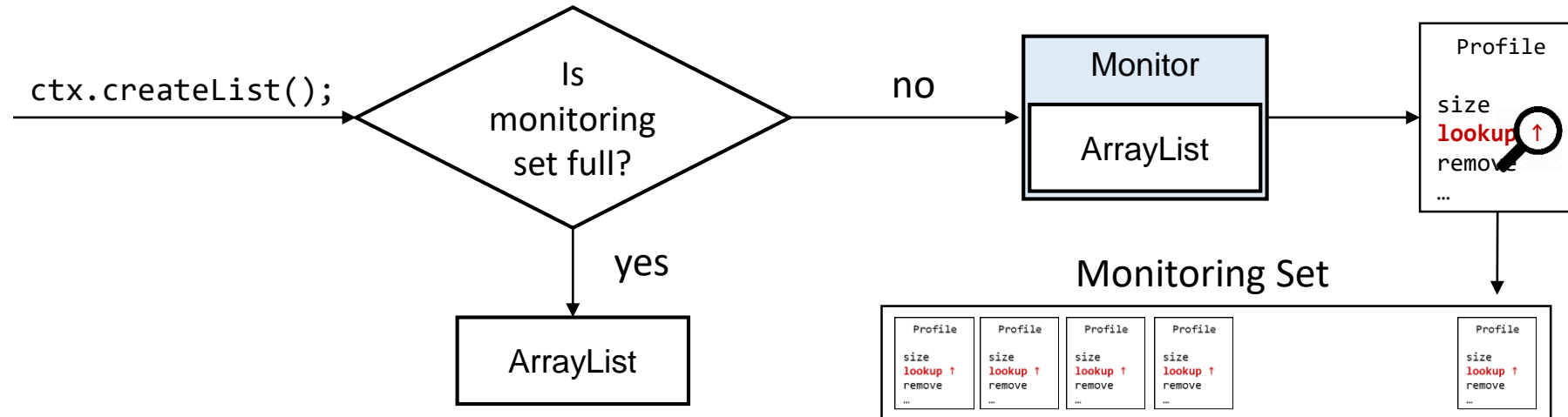


- A variant is selected when it satisfies the rule
 - Criteria satisfied by multiple variants? Select the variant with biggest improvement

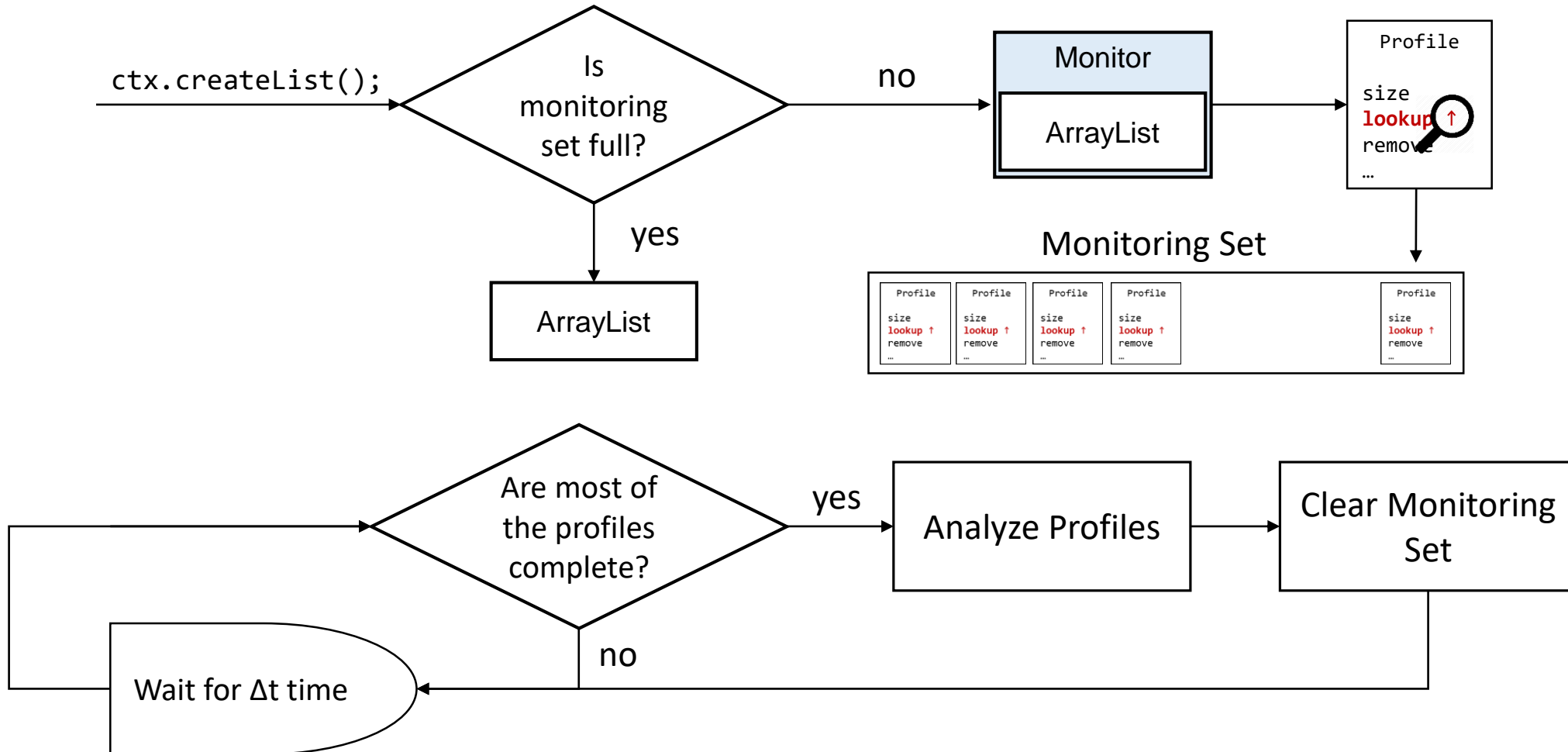
C) How to Find a Better Variant?



Monitoring the Collections Usage



Monitoring the Collections Usage



Monitoring the Collections Usage

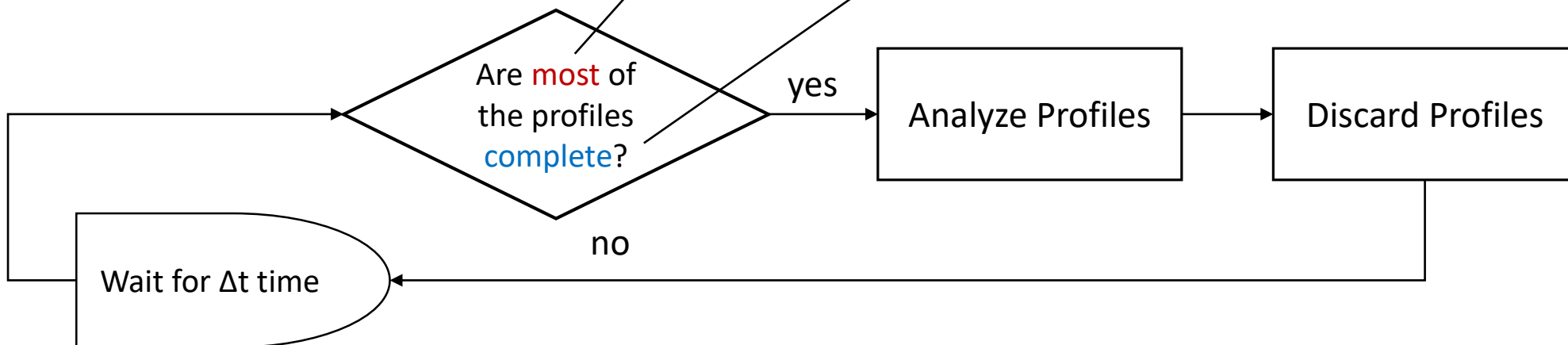
Parametrized ratio r

$r = 0.3$ too unstable

$r = 1.0$ too slow

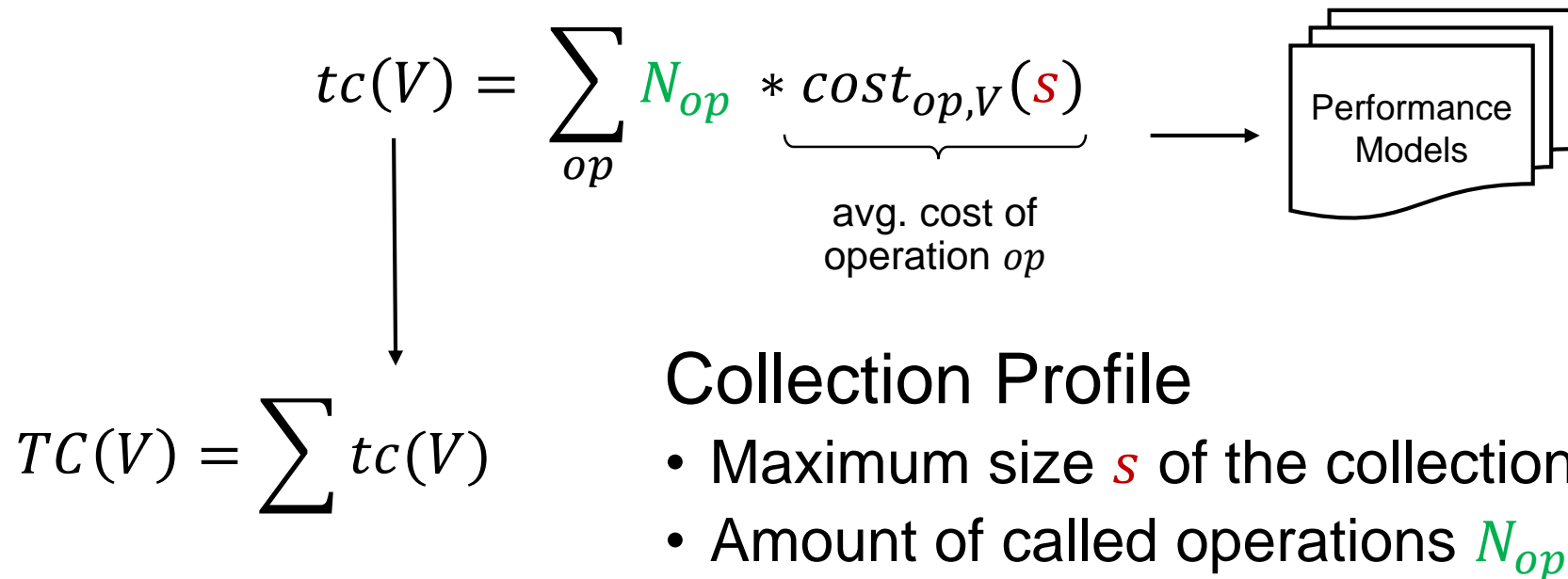
$r = 0.6$ good compromise

Instances have been assigned for
garbage collection



Estimating the Performance

We compare variants V performance according to the total cost metric $TC(V)$



Estimating the Performance

- Polynomial function of the collection size s

$$cost_{op,v}(s) = \sum_{k=0}^d a_k s^k$$

- We design a series of **benchmarks** to calculate the coefficients


- + 30 variants
- Single Operation Scenario
- Measurement Variables
 - Execution Time
 - Memory allocation

Factor	Levels/Categories
Size	[10, 100,200,...,1M]
Operations	populate, contains, iterate, middle, remove
Data Type	Integer
Data Distribution	Uniform

Estimating the Performance

- Selects a variant V_{new} to replace the current V_{cur} when it satisfies the Performance Rule

$$R_{alloc} \left| \begin{array}{ll} \text{Improvement} & \text{Max Penalty} \\ \text{Allocation} = -20\% & \text{Time} = +125\% \end{array} \right.$$



$$\left(\frac{TC_{alloc}(V_{new})}{TC_{alloc}(V_{cur})} \leq 0.8 \right) \wedge \left(\frac{TC_{time}(V_{new})}{TC_{time}(V_{cur})} \leq 1.25 \right)$$

Adaptation on Instance Level

- CollectionSwitch can also switch to an adaptive variant
 - Second level of adaptation
- Adaptive Collections
 - Small sizes: Memory efficient implementation (array)
 - Large sizes: Time efficient implementation (hash)

Variant	Transition	Threshold
AdaptiveSet	Array -> Hash	40
AdaptiveMap	Array -> Hash	50

Transition is done by **copying** the elements

Evaluating the Model I

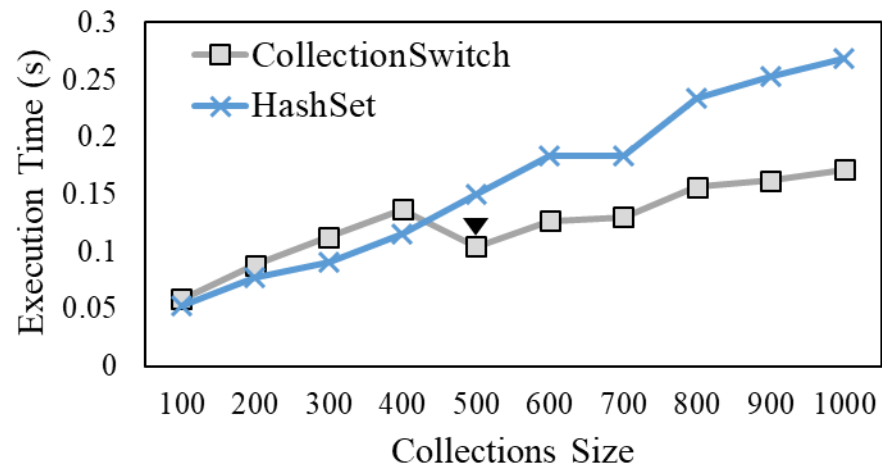
- Micro-benchmarks
 - Population of the collection
 - 100 searches of a random element



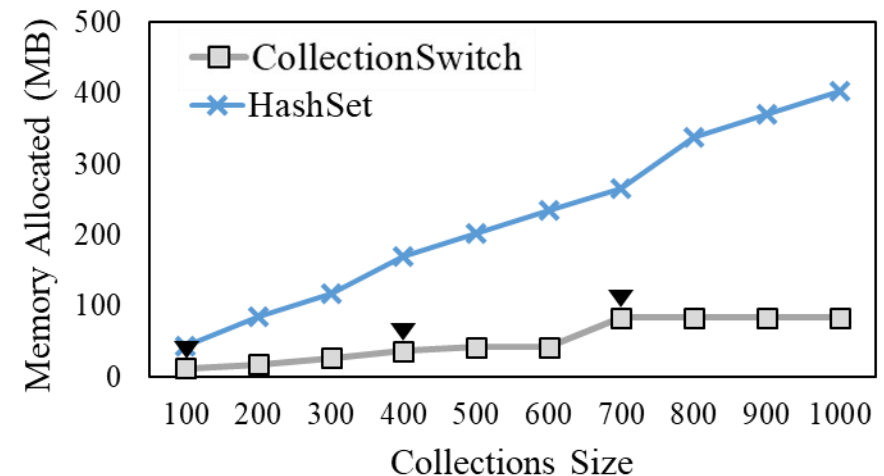
Rule	Improvement
R_{time}	Time = 20%

Evaluating the Model II

- Micro-benchmarks
 - Population of the collection
 - 100 searches of a random element



Rule	Improvement
R_{time}	Time = 20%



Rule	Improvement	Max Penalty
R_{alloc}	Allocation = 20%	Time = 20%

Evaluating the Performance Improvement

- DaCapo benchmarks
 - Real applications

Bench	Input Size	#Target. Alloc.	Original Run		CollectionSwitch							
			T(s)	M(MB)	R_{time}				R_{alloc}			
					$T_1(s)$		$M_1(MB)$		$T_2(s)$		$M_2(MB)$	
avrora	large	7	4.1	72.4	4.2	–	72.1	–	4.4	+7%	65.4	-10%
bloat	large	17	30.3	96.7	28.9	–	96.9	–	26.6	-12%	89.4	-8%
fop	default	15	0.5	53.4	0.5	–	57.0	+7%	0.5	–	53.9	–
h2	large	10	40.1	509.0	38.3	-6%	508.7	–	44.6	+11%	470.1	-8%
lusearch	large	12	3.6	282.4	3.1	-15%	269.4	-5%	3.4	-6%	268.0	-5%

Evaluating the Overhead

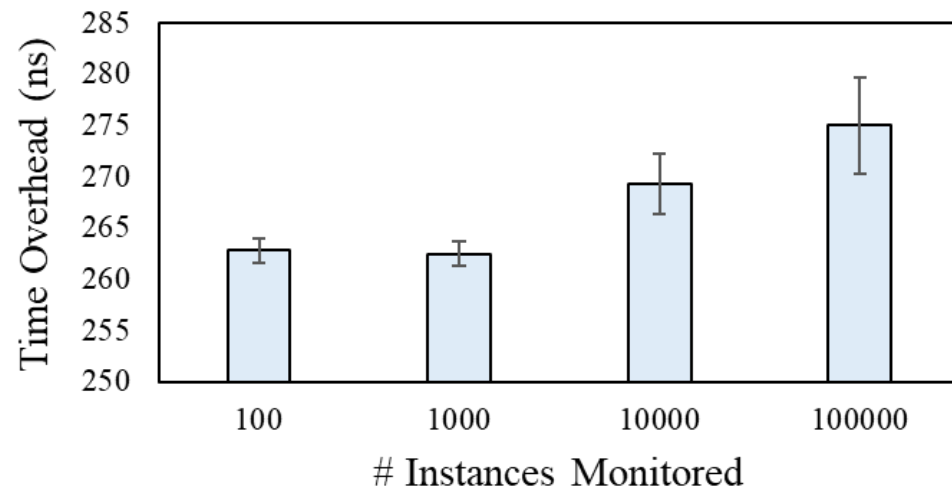
- DaCapo

- No significant overhead

Rule	Improvement
R_{imp}	Time = $\infty\%$

- Estimation of Variants Performance

- Below 300 ns



- Memory Overhead

- Footprint of each Allocation Context ~1Kb.

Summary

- Selecting the appropriate collection is critical for designing efficient Java applications
- CollectionSwitch selects collection at runtime through:
 - Adaptive [allocation-sites](#)
 - Adaptive [collections](#)
- Improvement on execution time and memory of real applications

Thank You!

diego.costa@informatik.uni-heidelberg.de

References

- [Cha++11] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. 2011. **Brainy: effective selection of data structures.** (PLDI '11)
- [Cos++17] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. **Empirical Study of Usage and Performance of Java Collections.** (ICPE '17)
- [LS09] Lixia Liu and Silvius Rus. 2009. **Perflint: A Context Sensitive Performance Advisor for C++ Programs.** (CGO '09)
- [OME09] Ohad Shacham, Martin Vechev, and Eran Yahav. **Chameleon: adaptive selection of collections.** (PLDI '09)
- [Sam++16] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. **Energy profiles of Java collections classes.** (ICSE '16)