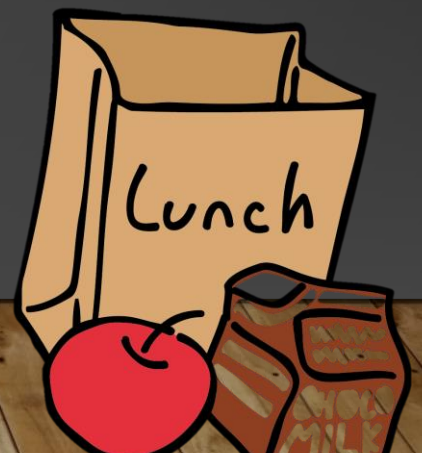
ESCAPING THE CLONE ZONE: JAVA RUNTIME-MANAGED SNAPSHOTS

MATTHEW C. DAVIS

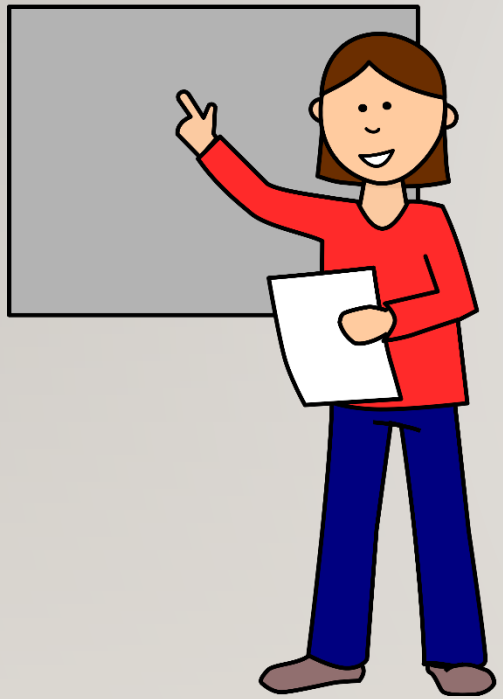
EAST CAROLINA UNIVERSITY

SERPL 2019 – AUGUSTA, GA

MAY 11, 2019



ORGANIZATION OF THIS PRESENTATION



- Background
- Motivation
- Previous work
- Current Work
- Future Work

20 Minutes

10 Minutes

BACKGROUND

IMMUTABLE OBJECTS

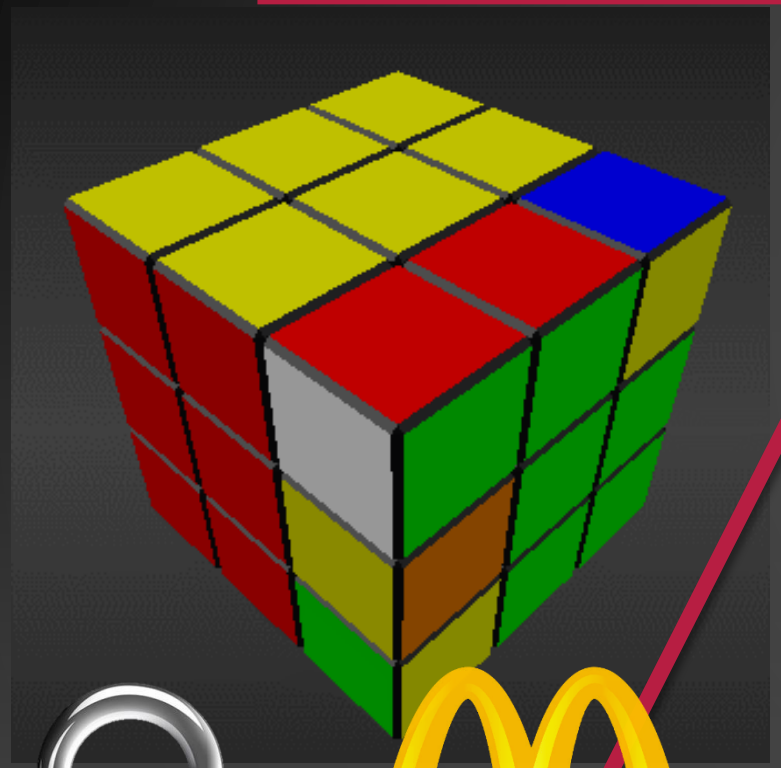
Not modified

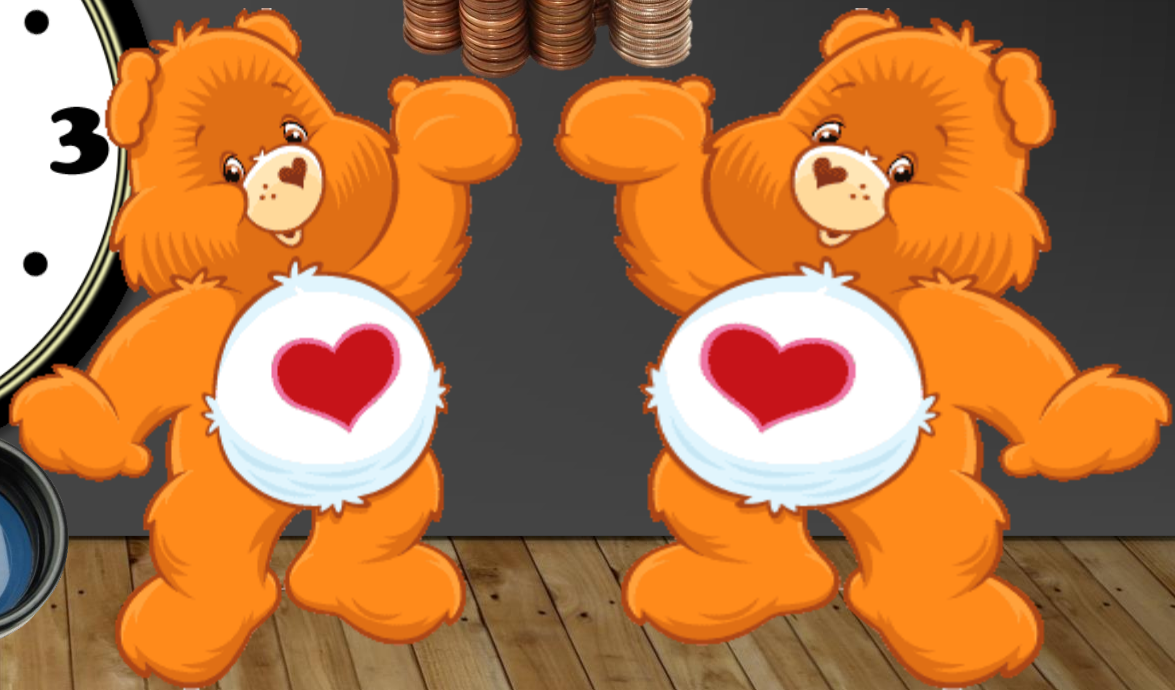
String "SER" .concat(String "PL19")

New String

String "SERPL19"

DESIGNING MUTABLE OBJECTS





USING SHARED MUTABLE OBJECTS

Worklist COPY

.addAll(COPY)

Worklist

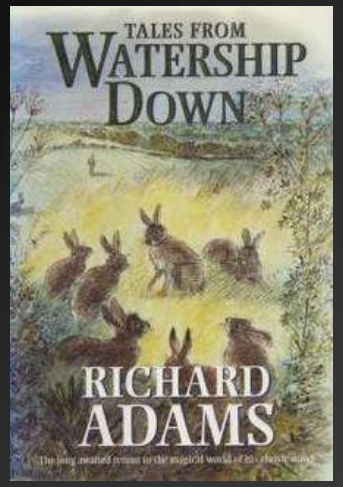
.clear()

Worklist

server

client

NATIVE JAVA OPTIONS



Under-specified semantics

(clone, copy constructor)

Not universal

(serialize, clone, copy constructor)

MOTIVATION

MOTIVATION

Propose a universal method to guarantee non-shared state with predictable semantics.

PREVIOUS WORK

BASIC IDEA

```
1 // Snap/unshare objInput state prior to iterating
2 public void addAll(__snap__ ArrayList<String> objInput) {
3     for(String str : objInput) {
4         this.add(str);
5     }
6 }
```

- Formal parameter modifier: __snap__
- Runtime guarantees non-shared state
- Type independent
- Predictable depth of copy

JAVA LANGUAGE SPECIFICATION PROPOSAL

- § 3.9 – Keyword list: add __snap__
- § 8.4.1 – Formal parameter grammar

```
FormalParameterList:
  ReceiverParameter
  FormalParameters , LastFormalParameter
  LastFormalParameter

FormalParameters:
  FormalParameter {, FormalParameter}
  ReceiverParameter {, FormalParameter}

FormalParameter:
  {VariableModifier} UnannType VariableDeclaratorId

VariableModifier:
  Annotation
  final
  __snap__

ReceiverParameter:
  {Annotation} UnannType [Identifier .] this

LastFormalParameter:
  {VariableModifier} UnannType{Annotation} ...
     VariableDeclaratorId
  FormalParameter
```

Figure 4.4: JLS10 § 8.4.1 - adapted formal parameter grammar

It is a compile-time error if __snap__ appears more than once as a modifier for a formal parameter declaration or if the UnannType of the formal parameter declaration is not a reference type.

Figure 4.5: JLS10 § 8.4.1 - adapted formal parameter rules

It is neither a compile-time nor a runtime error if __snap__ is a modifier on a formal parameter with apparent or actual type enum. Rather, at runtime no snap operation is performed on enum types.

Figure 4.6: JLS10 § 8.4.1 - adapted formal parameter rules for enums

EXPERIMENT #1: TRANSFORMATION

- A transformation approach accepts code in an enriched syntax or Domain-Specific Language (DSL) and transforms the input to another, usually standard, language.

```
1 // Create and return a snapshot of obj
2 public T snap __snap__ T obj) {
3     return obj;
4 }
```

Figure 5.1: _snap_ keyword example: pre-transformation

clone()

Serialize

```
1 public T snap(T obj) {
2     obj = (SnapTree.getInstance()).snap(obj); //inserted
3     return obj;
4 }
```

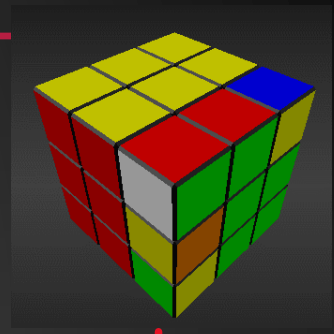
Figure 5.2: _snap_ keyword example: post-transformation

QUICKLY RAN INTO PROBLEMS

- **Not universal**
  - Not mandatory for objects to be:
    Cloneable, Serializable, or … Copy-Constructer-able
- **Not predictable**
  - Serialization and Clone() yield different results depending on type.
  - CloneNotSupported and NotSerializable exceptions.

Key point: Transformation not viable here

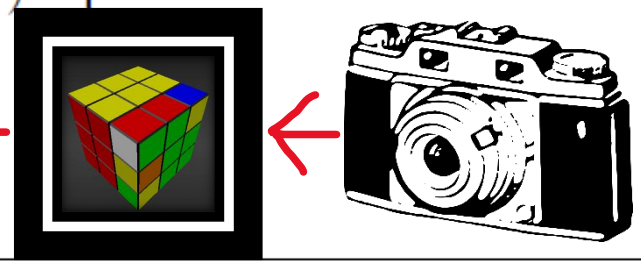
EXPERIMENT #2: MODIFY OPENJDK DIRECTLY



Basic Idea:

Implement __snap__ modifier and snapshotting directly in OpenJDK

```
1  // Snap/unshare objInput state prior to iterating
2  public void addAll(__snap__ ArrayList<String> objInput) {
3      for(String str : objInput) {
4          this.add(str);
5      }
6  }
```



EXPERIMENT #2: MODIFY OPENJDK

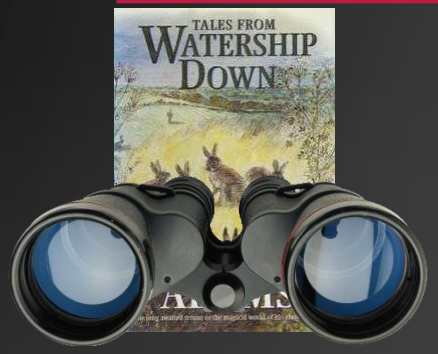
Steps for Experiment:

1. **Specify a new bytecode, 0xcb asnap**
   → Triggers object snapshotting within the Java Virtual Machine

2. **Modify the Java compiler (parse and generate phases)**
   → Emit 0xcb when loading actual parameters declared w/modifier __snap__

3. **Modify the HotSpot JVM's bytecode interpreter**
   → Snapshot object on operand stack when encountering 0xcb bytecode

WHAT IS THE RELATIVE PERFORMANCE?



- In this evaluation:
  - Focused on steady-state performance
  - Disabled JIT compilation (adapting c1 and c2 compilers are future wo
  - Garbage collected before each measured operation
  - Evaluation is x64-only due to native x64 assembly code
  - Large & Small inputs evaluated.

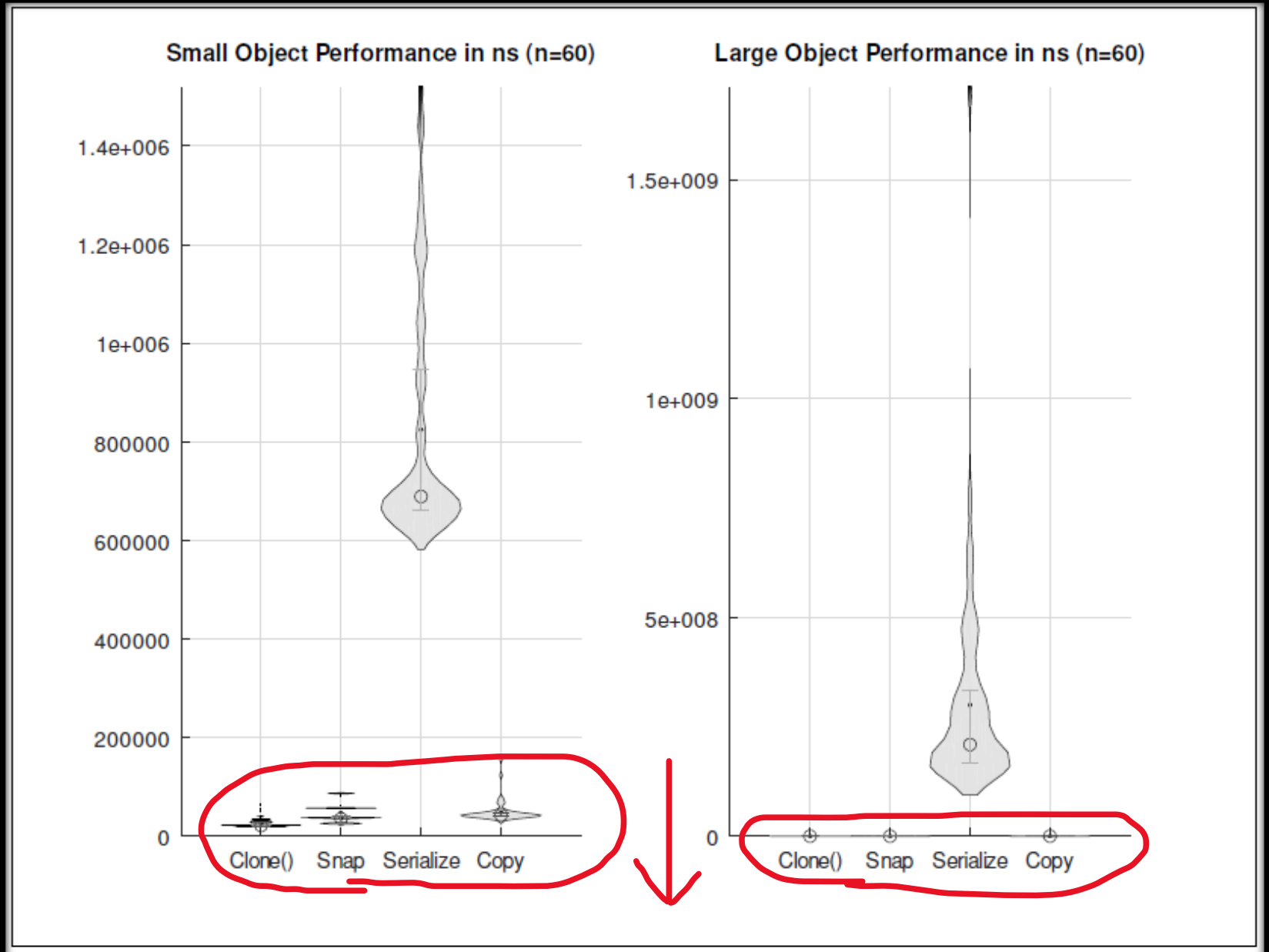


Figure 7.7: Benchmarking Results - Violin Plot

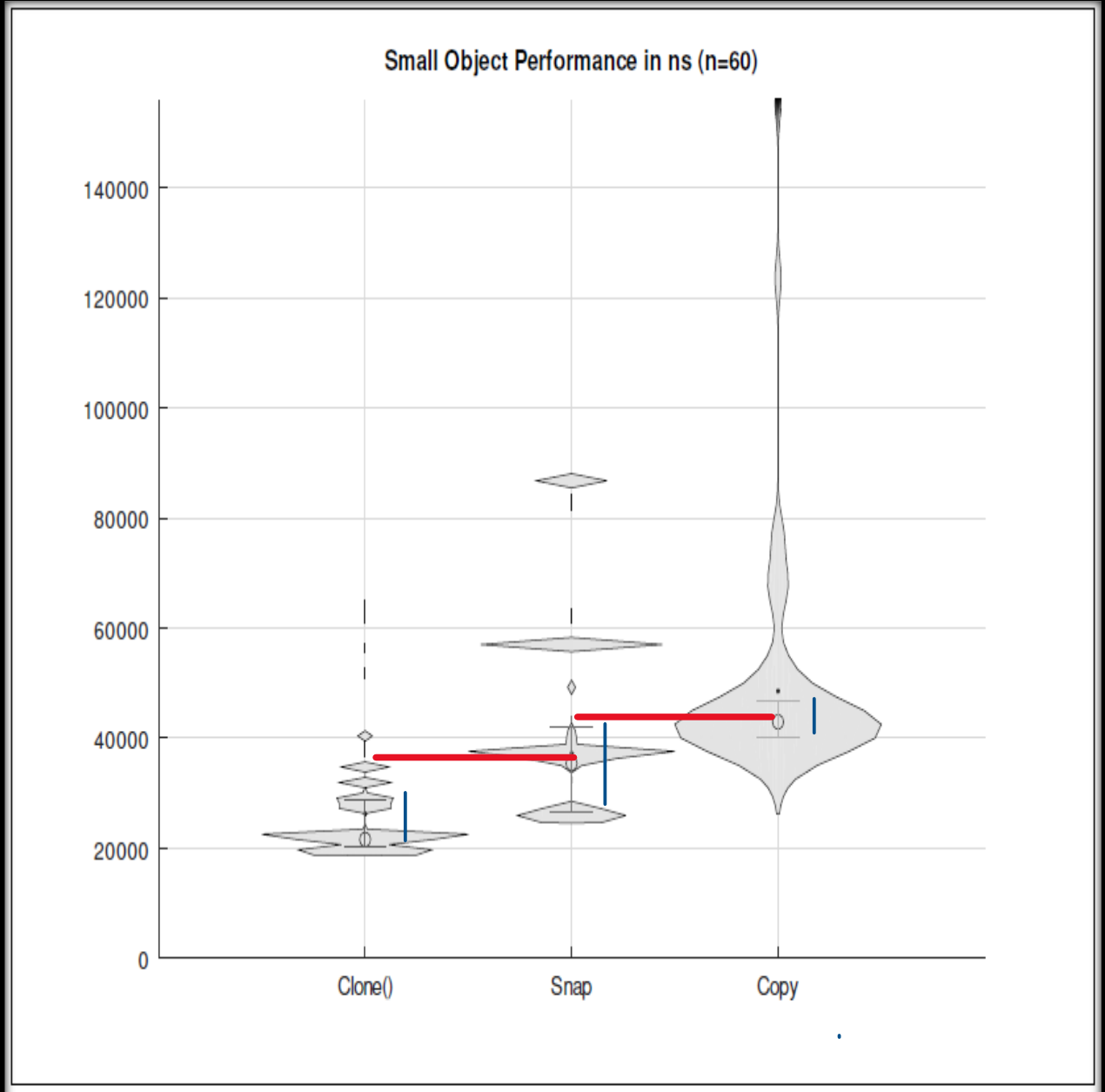


Figure 7.8: Benchmarking Results - Violin Plot - Small Object w/o Serialize

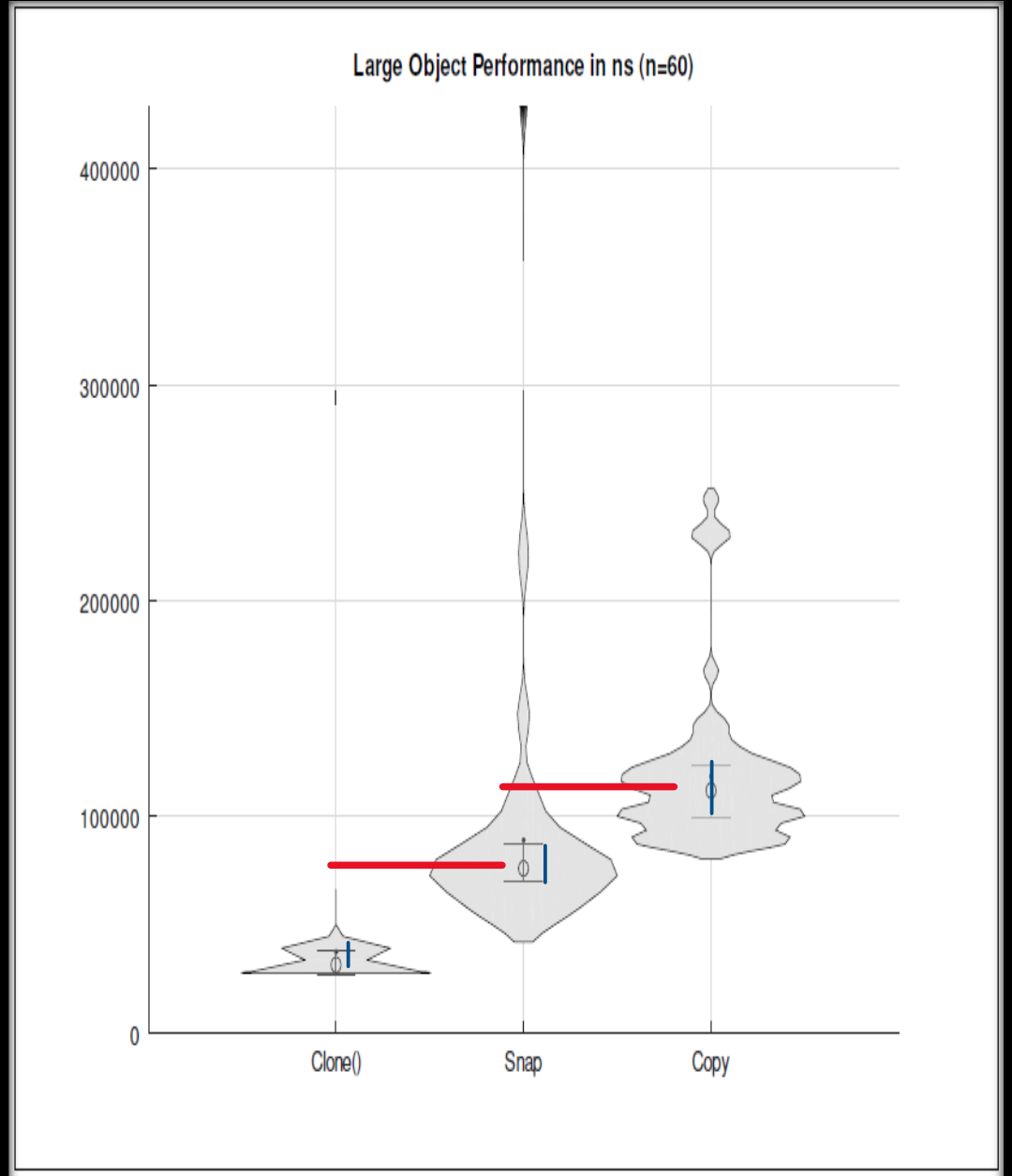


Figure 7.9: Benchmarking Results - Violin Plot - Large Object w/o Serialize

PREVIOUS WORK SUMMARY

At the end:

1. **Had a working prototype based on OpenJDK 10**
   → Implemented __snap__ and 0xcb in javac and HotSpot JVM

2. **That was performant**
   → Relative to extant methods

3. **And provided consistent semantics for all objects**

CURRENT WORK

CURRENT RESEARCH QUESTIONS

R1. How frequently are clone(), serialize(), copy constructor, copy libraries used?

Not well understood

R2. What is the distribution of Java object graph depths?

Not well understood

PLAN OF CURRENT WORK

R1. *f*()

n Most Starred

Rascal MPL

Locations:

Clone
Serialize
Copy Constructor
Copy Library

Rascal: Inject Annotations Into Source

+

Annotations Trigger Depth Check

+

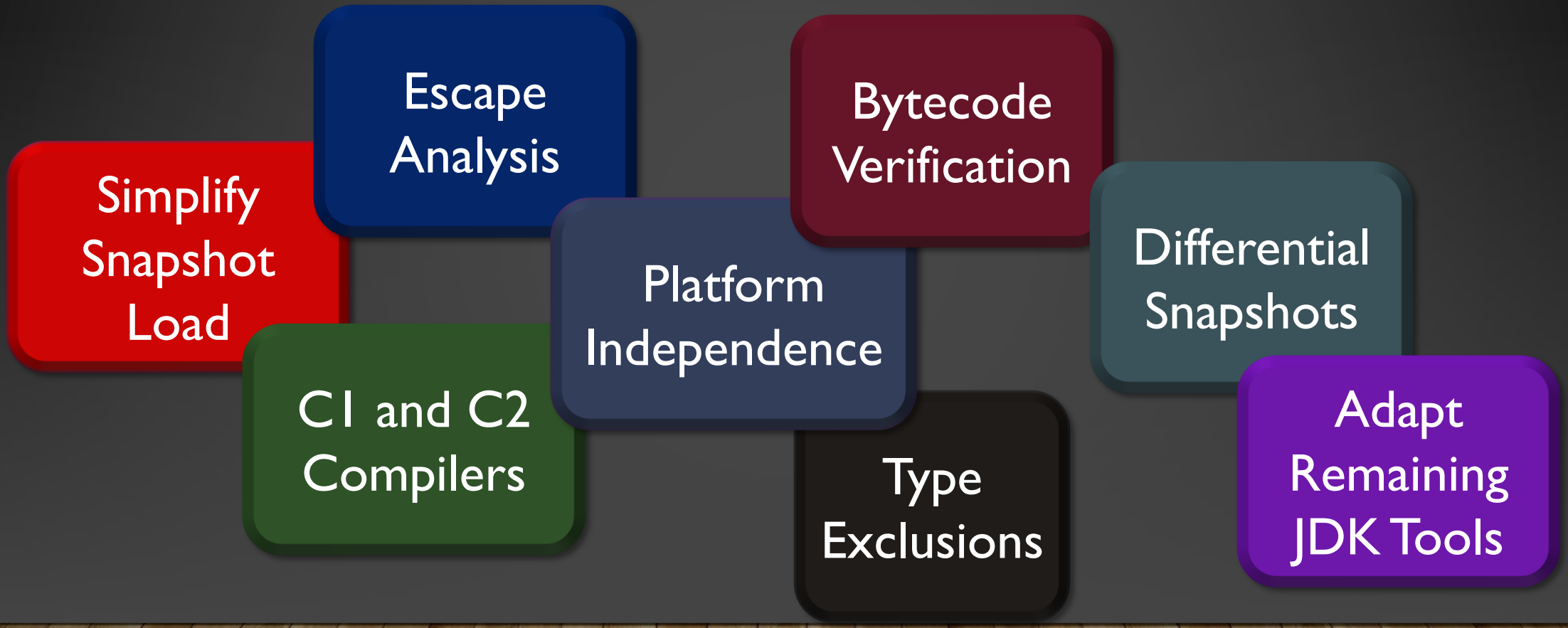
Execute: Output Depth Metrics

=

R2. Graph Depth Metrics

FUTURE WORK

PLENTY OF WORK FOR THE FUTURE



Q & A

THANK YOU