A Scalable Technique for Characterizing the Usage of Temporaries in Framework-intensive Java Applications

Bruno Dufour, Barbara G. Ryder, Gary Sevitsky

In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT '08/FSE-16)

Presented by Long Cheng
November 17, 2015

About the Authors

Dr. Bruno Dufour, Assistant Professor in Department of Computer Science and Operational Research, University of Montreal

Dr. Barbara G. Ryder, J. Byron Maupin Professor of Engineering in Department of Computer Science at Virginia Tech

Dr. Gary Sevitsky, Researcher Scientist at IBM T.J. Watson Research Center

Overview

1. Background
   - Motivation
   - Escape Analysis
   - Blended Analysis Paradigm
   - Blended Escape Analysis — ISSTA 2007
2. Optimized Blended Analysis — SIGSOFT 2008/FSE-16
3. Evaluation
   - Experimental Setup
   - Experimental Results

Motivation

- *Framework-intensive applications* (e.g., Web applications) heavily use *temporary data structures* (i.e., temporaries), often resulting in performance bottlenecks.

Motivation

- *Framework-intensive applications* (e.g., Web applications) heavily use *temporary data structures* (i.e., temporaries), often resulting in performance bottlenecks.
- It is common for temporaries to occur as mini-data structures (i.e., groups of connected objects), built up with much effort only to be thrown away shortly thereafter.

Motivation

- *Framework-intensive applications* (e.g., Web applications) heavily use *temporary data structures* (i.e., temporaries), often resulting in performance bottlenecks.
- It is common for temporaries to occur as mini-data structures (i.e., groups of connected objects), built up with much effort only to be thrown away shortly thereafter.
- Understanding the contributing factors to excessive use of temporaries is critical to being able ultimately to fix these performance problems.

Motivation

- *Framework-intensive applications* (e.g., Web applications) heavily use *temporary data structures* (i.e., temporaries), often resulting in performance bottlenecks.
- It is common for temporaries to occur as mini-data structures (i.e., groups of connected objects), built up with much effort only to be thrown away shortly thereafter.
- Understanding the contributing factors to excessive use of temporaries is critical to being able ultimately to fix these performance problems.
- This paper presents an **optimized blended escape analysis** to approximate object lifetimes and thus, to identify these temporaries and their uses.

Main Contributions

An optimized blended escape analysis algorithm

Prune away unexecuted basic blocks in methods, achieving increased precision and scalability

Main Contributions

An optimized blended escape analysis algorithm

Prune away unexecuted basic blocks in methods, achieving increased precision and scalability

New metrics

for blended static and dynamic analyses that quantify key properties related to the use of temporary objects

Main Contributions

An optimized blended escape analysis algorithm

Prune away unexecuted basic blocks in methods, achieving increased precision and scalability

New metrics

for blended static and dynamic analyses that quantify key properties related to the use of temporary objects

Empirical findings

Characterize the nature and usage of temporary objects in representative, framework-intensive Java applications.

Escape Analysis

- A technique for approximating the effective **lifetime of objects**, i.e., computing if and how newly created objects become visible beyond the method in which they were created.

Escape Analysis

- A technique for approximating the effective **lifetime of objects**, i.e., computing if and how newly created objects become visible beyond the method in which they were created.
- It has been used traditionally for compiler optimizations requiring either information about an object (i) *escaping a method invocation* or (ii) *escaping an allocating thread*.

Escape Analysis

- A technique for approximating the effective **lifetime of objects**, i.e., computing if and how newly created objects become visible beyond the method in which they were created.
- It has been used traditionally for compiler optimizations requiring either information about an object (i) *escaping a method invocation* or (ii) *escaping an allocating thread*.
- Three defined escape states for each object: **globally escaping**, **non-escaping** or escaping through parameters and/or return values (**arg-escaping**).

Escape Analysis – An Example

```
1  public X identity(X p1) {
2    return p1;
3  }

5  public X escape(X p2) {
6    G.global = p2;
7    return p2;
8  }

10 public void f() {
11   X inst;
12   if (cond)
13     inst = identity(new Y());
14   else
15     inst = escape(new Z());
16 }
```

Example program

Escape Analysis – An Example

```
1  public X identity(X p1) {
2    return p1;
3  }

5  public X escape(X p2) {
6    G.global = p2;
7    return p2;
8  }

10 public void f() {
11   X inst;
12   if (cond)
13     inst = identity(new Y());
14   else
15     inst = escape(new Z());
16 }
```

Example program

- Objects that are *reachable through parameters* or that are *returned to caller methods* are labeled *arg escaping*.

Escape Analysis – An Example

```
1  public X identity(X p1) {
2    return p1;
3  }

5  public X escape(X p2) {
6    G.global = p2;
7    return p2;
8  }

10 public void f() {
11   X inst;
12   if (cond)
13     inst = identity(new Y());
14   else
15     inst = escape(new Z());
16 }
```

Example program

- Objects that are *reachable through parameters* or that are *returned to caller methods* are labeled *arg escaping*.
- An object is marked *globally escaping* when it becomes *globally* reachable (e.g., assigned to a static field).

Escape Analysis – An Example

```
1  public X identity(X p1) {
2    return p1;
3  }

5  public X escape(X p2) {
6    G.global = p2;
7    return p2;
8  }

10 public void f() {
11   X inst;
12   if (cond)
13     inst = identity(new Y());
14   else
15     inst = escape(new Z());
16 }
```

Example program

- Objects that are *reachable through parameters* or that are *returned to caller methods* are labeled *arg escaping*.
- An object is marked *globally escaping* when it becomes *globally* reachable (e.g., assigned to a static field).
- Objects that don't escape are marked as *captured*.

Escape Analysis Algorithm[1]

- A context-sensitive, flow-sensitive escape analysis algorithm.
- Escape analysis proceeds in a bottom-up manner on the call graph.
- A **connection graph** is generated at each call graph node to represent a summary of the relevant data structures at that node and the (current) escape state of abstract objects.

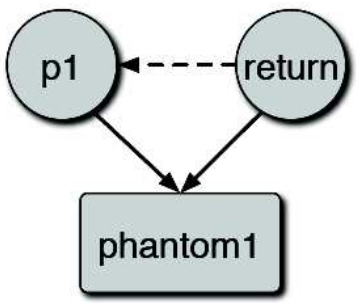
[1] Jong-Deok Choi et al. "Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis". In: *ACM Trans. Program. Lang. Syst.* 25.6 (2003), pp. 876–910.

Escape Analysis Algorithm – An Example

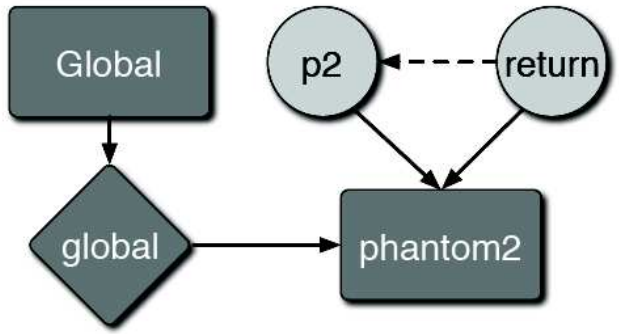
```
1  public X identity(X p1) {
2    return p1;
3  }

5  public X escape(X p2) {
6    G.global = p2;
7    return p2;
8  }

10 public void f() {
11   X inst;
12   if (cond)
13     inst = identity(new Y());
14   else
15     inst = escape(new Z());
16 }
```

Example program

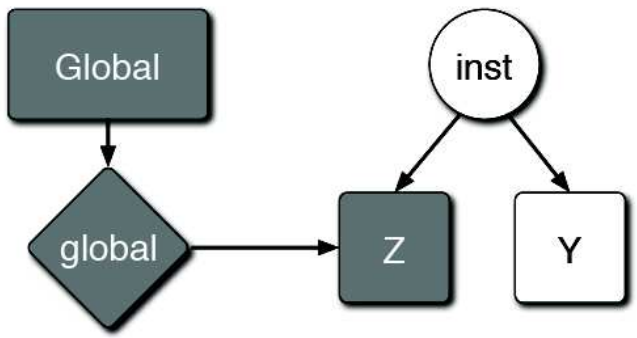
Phantom object node represents all objects that could be passed to it.



a) Connection graph for `identity`



b) Connection graph for `escape`



c) Connection graph for `f`

Summary connection graphs for methods in the example program

Blended Analysis Paradigm

- *Dynamic analysis* is used to obtain the calling structure of a *particular execution of interest* and then a *static analysis* is performed on that calling structure to obtain more detailed semantic information relevant for performance understanding.

Blended Analysis Paradigm

- *Dynamic analysis* is used to obtain the calling structure of a *particular execution of interest* and then a *static analysis* is performed on that calling structure to obtain more detailed semantic information relevant for performance understanding.
- The *hypothesis* is that blended analysis will enable a more precise and scalable analysis for performance understanding at an acceptable cost, in comparison to a purely static analysis (i.e., too imprecise) or a purely dynamic analysis (i.e., too costly because sampling will not provide sufficient precision).

Blended Escape Analysis — ISSTA 2007[3]

- Used IBM's Jinsight tool to generate a **dynamic call graph** used as input to the blended escape analysis
- The precision of the information in the connection graphs can be improved by retaining richer calling context information — dynamic calling context tree (CCT)[2].
- The postprocessing algorithm generates a **reduced connection graph** for each context in the CCT, which provides a good level of abstraction for understanding and manual exploration of temporary structures.

---

[2] Glenn Ammons, Thomas Ball, and James R. Larus. "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling". In: PLDI '97, pp. 85–96.

[3] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. "Blended Analysis for Performance Understanding of Framework-based Applications". In: *ISSTA '07*. 2007, pp. 118–128.

Optimized Blended Analysis

Declared Types

Taking advantage of knowledge of declared types, type-inconsistent edges are never added to the connection graph, which increases the precision and reduces the execution time cost.

Optimized Blended Analysis

Declared Types

Taking advantage of knowledge of declared types, type-inconsistent edges are never added to the connection graph, which increases the precision and reduces the execution time cost.

Basic Block Pruning

Prune a basic block from the control flow graph of a method if it can be shown that the block was never executed. Unexecuted basic blocks are identified using two kinds of dynamic information for each method, observed calls and allocated types of instances.

Experimental Setup

- Escape analysis was built using the WALA analysis framework.
- Used IBM's Jinsight tool to generate a dynamic call graph. The Jinsight profiler is routinely used within IBM for performance diagnosis.
- Used two well-known framework-intensive applications: *Trade* and *Eclipse*
- Four benchmarks (three configurations of Trade):
  - Trade Direct/Std
  - Trade Direct/WS
  - Trade EJB/Std
  - Eclipse JDT

Experimental Setup

- Escape analysis was built using the WALA analysis framework.
- Used IBM's Jinsight tool to generate a dynamic call graph. The Jinsight profiler is routinely used within IBM for performance diagnosis.
- Used two well-known framework-intensive applications: *Trade* and *Eclipse*
- Four benchmarks (three configurations of Trade):
  - Trade Direct/Std
  - Trade Direct/WS
  - Trade EJB/Std
  - Eclipse JDT

Benchmark	Alloc'ed Types	Alloc'ed Instances	Methods	Calls	Max Stack Depth
Direct/Std	30	186	710	4,484	26
Direct/WS	166	5,522	3,308	127,794	53
EJB/Std	82	1,751	1,978	60,936	62
Eclipse	168	53,191	1,411	1,081,927	53

Benchmark characteristics

Experimental Results

Pruning Effects

- Measure the impact of the pruning technique on the scalability of the analysis

- Metric 1: Pruned basic blocks
- Metric 2: Execution time

Experimental Results

Pruning Effects

- Measure the impact of the pruning technique on the scalability of the analysis

- Metric 1: Pruned basic blocks
- Metric 2: Execution time

Benchmark	Pruned BBs	Running time (h:m:s)		Speed-up
		Orig	Pruned	
Trade Direct/Std	38.8%	0:00:22	0:00:11	2.0
Trade Direct/WS	36.0%	3:01:52	0:19:31	9.3
Trade EJB/Std	41.0%	6:49:54	0:13:50	29.6
Eclipse JDT	30.9%	43:13:20	2:01:39	21.3
Average	36.7%			15.6

Pruning effects

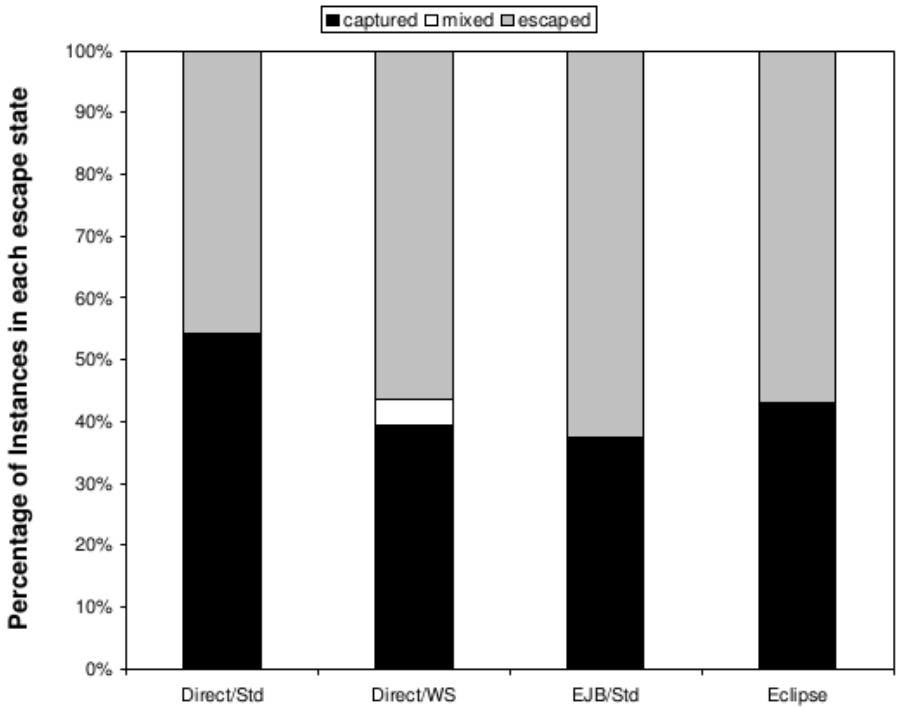
Experimental Results

Disposition

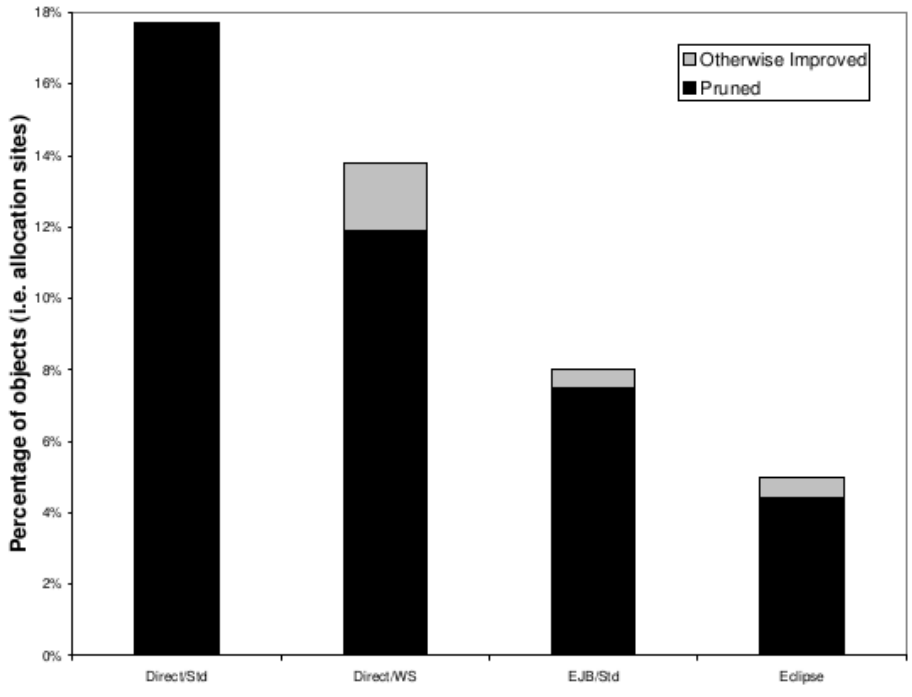
- In escape analysis, every object receives a disposition, i.e., final escape state.
- Dynamic imprecision sometimes introduces ambiguity regarding the path in the dynamic CCT traversed by an instance. a state henceforth is marked as mixed (both escaping and captured).

- Metric 3: Disposition breakdown
- Metric 4: Disposition improvement – percentage of objects whose disposition is improved by the pruning algorithm

Experimental Results



Disposition breakdown



Disposition improvement

Experimental Results

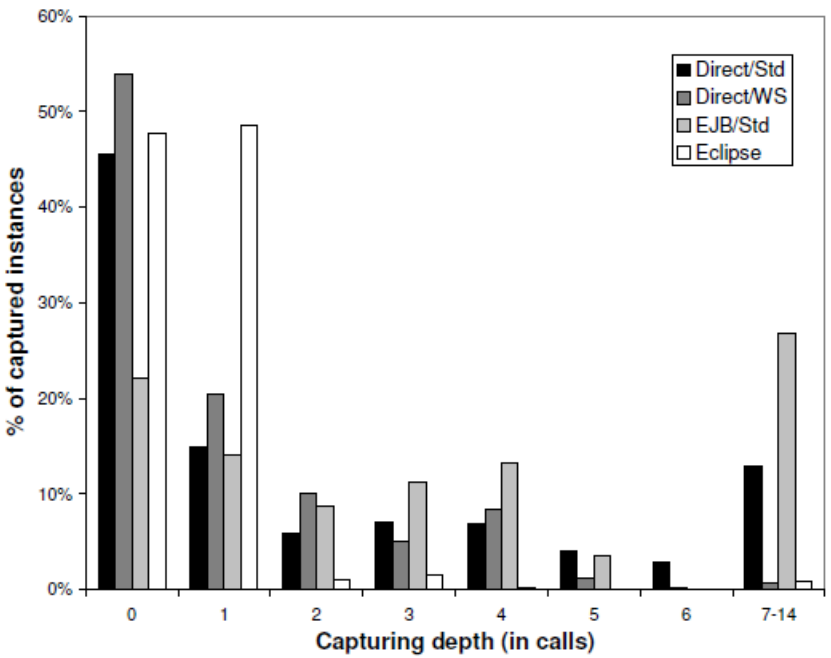
Capturing Depth

- A measure of the nature of the individual regions in the program calling structure that use temporaries
- The length of the shortest acyclic path from its allocating context to its capturing context

Experimental Results

Capturing Depth

- A measure of the nature of the individual regions in the program calling structure that use temporaries
- The length of the shortest acyclic path from its allocating context to its capturing context



Capturing depth histogram

Experimental Results

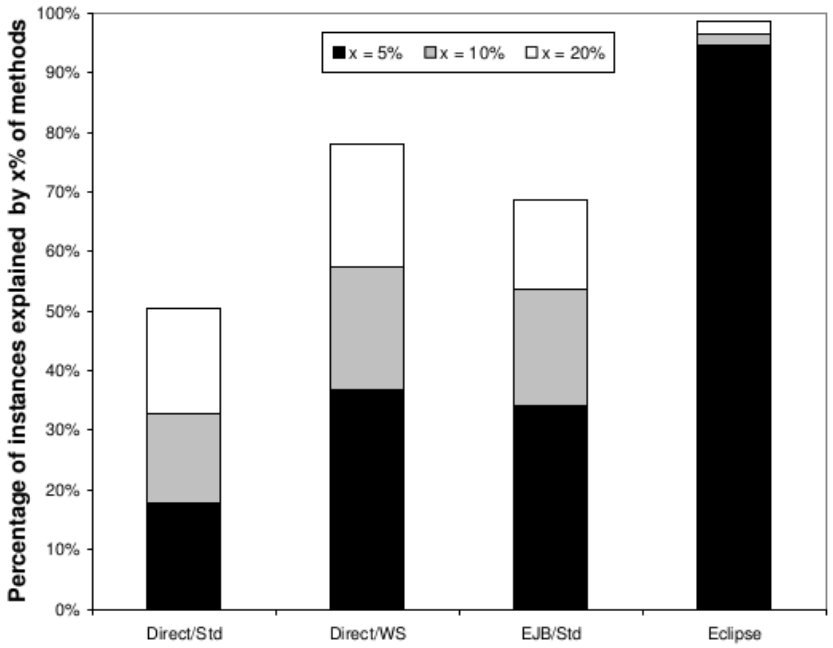
Concentration

- Understand whether object churn behavior is typically concentrated in a few regions, or is spread out across many regions
- The percentage of captured instances that are explained by X% of the top capturing methods

Experimental Results

Concentration

- Understand whether object churn behavior is typically concentrated in a few regions, or is spread out across many regions
- The percentage of captured instances that are explained by X% of the top capturing methods



Concentration

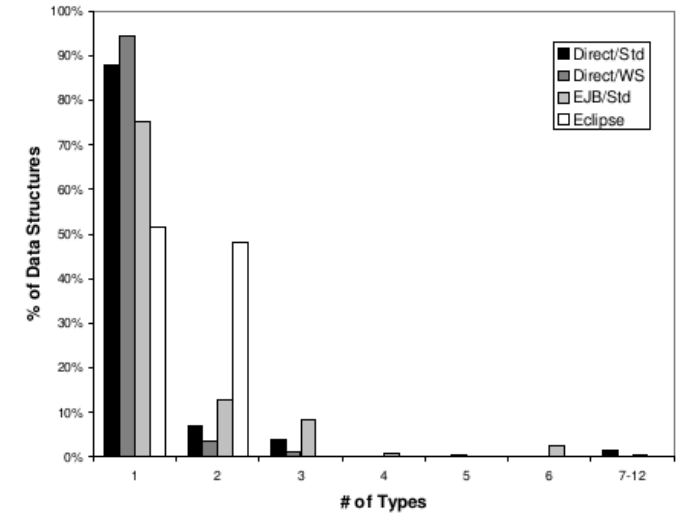
Experimental Results

Complexity of Data Structures

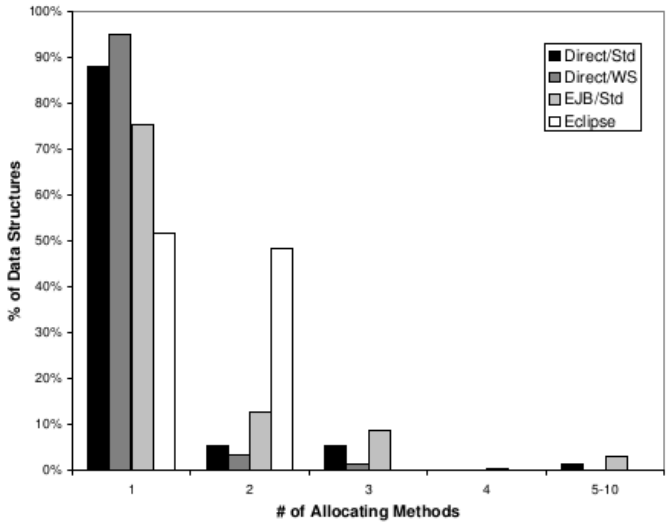
A data structure in the reduced connection graph of a calling context in the CCT consists of a root (i.e., an object with no incoming edges) and those nodes reachable from it that have the same escape state.

- Metric 7: # of types – number of distinct object types in each data structure
- Metric 8: # of allocating methods – number of distinct methods that allocate instances that are part of this data structure
- Metric 9: Height of data structure – length of the longest acyclic path in the reduced connection graph from a given data structure root to any other object in the data structure
- Metric 10: Maximum capturing distance – the longest capturing call chain corresponding to an instance contained in the data structure

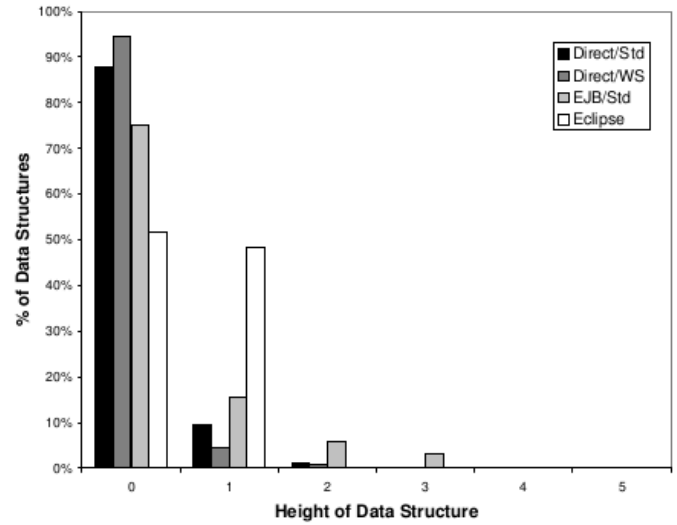
Experimental Results

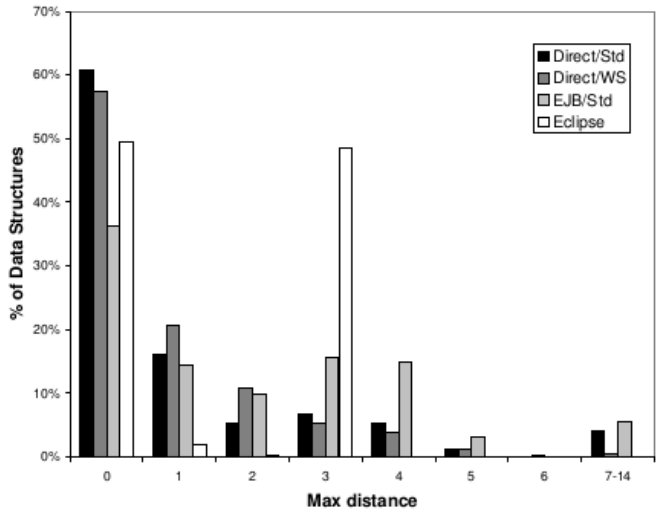


a) # of types



b) # of allocating methods





Complexity of data structures (by occurrences)

Conclusions

An optimized blended escape analysis algorithm, achieving increased precision and scalability

- Prune away type-inconsistent edges in the connection graph
- Prune away unexecuted basic blocks in methods in the connection graph

10 Metrics

that explain characteristics of the usage of temporary data structures in framework-intensive applications.

Empirical Findings

Characterize the nature and usage of temporary objects in four framework-intensive benchmarks.

Thanks!