MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Escape Analysis in Java

Choi, Gupta, Serrano, Sreedhar, Midkiff: OOPSLA 1999

Presented by Matt, Maddy, and Krithika

Agenda

1. Introduction and Motivation
2. Framework for Escape Analysis
3. Intra- and Interprocedural Analysis
4. Results
5. Strengths and Weaknesses
6. Related Works
7. Conclusion

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

1. Introduction and Motivation

Focusing on Java

- **General purpose programming language**
  - Performance matters!
- Java is inefficient
  - Objects are always heap-allocated
  - All objects use locking
- **Motivation:** Create a dataflow algorithm that optimizes Java programs through escape analysis

What Does Escape Analysis Do?

- **Checks if an object escapes a method**
  - Is the object used outside the method that created it?
  - **Optimization:** stack allocation
- **Checks if an object escapes a thread**
  - Is the object accessed by a thread other than the one that created it?
  - **Optimization:** eliminate synchronization primitives

$$\neg Escapes(O, M) \rightarrow \neg Escapes(O, T)$$

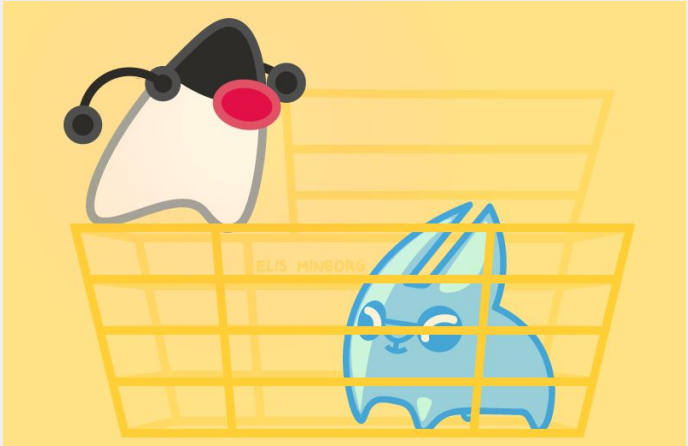
1. New framework for escape analysis in Java
2. First paper to help drop synchronization primitives
3. Less conservative exception handling
4. Experimental results

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

2. Framework for Escape Analysis

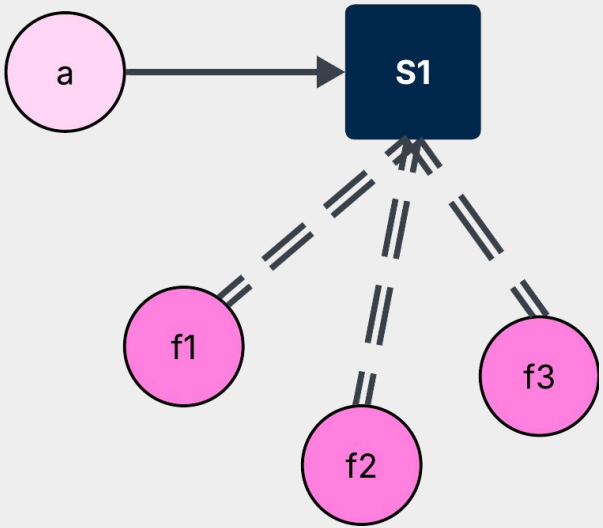
Escapement Lattice

- **NoEscape:** object does not escape the method it was created in
  - Stack allocable
  - Eliminate synchronization
- **ArgEscape:** object escapes the method, but not the thread
  - Eliminate synchronization
- **GlobalEscape:** object escapes all threads and methods

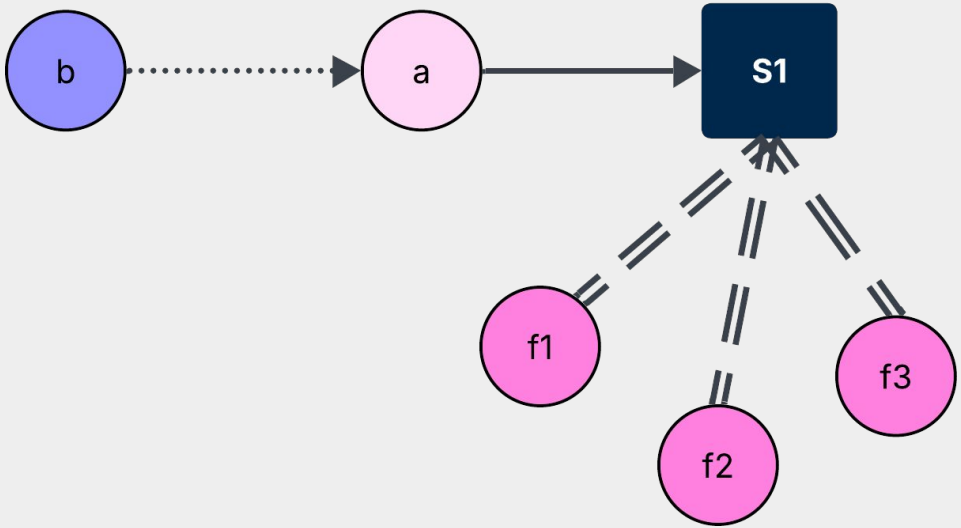


Connection Graph (CG)

S1: T a = new T(...)



S2: T b = a



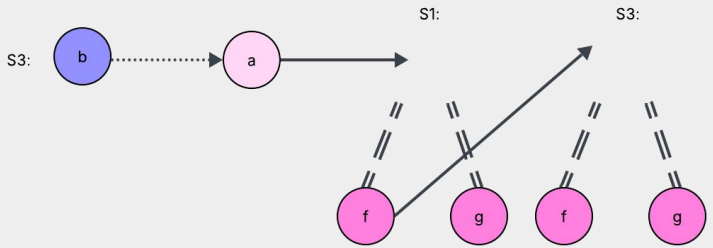
3. Intra- and Interprocedural Analysis

Intraprocedural Analysis

```
S1: T1 a = new T1(...);
S2: T1 b = a;
if()
    S3: a.f = new T1(...);
else
    S4: b.f = new T1(...);
s5: a = b.f;
```

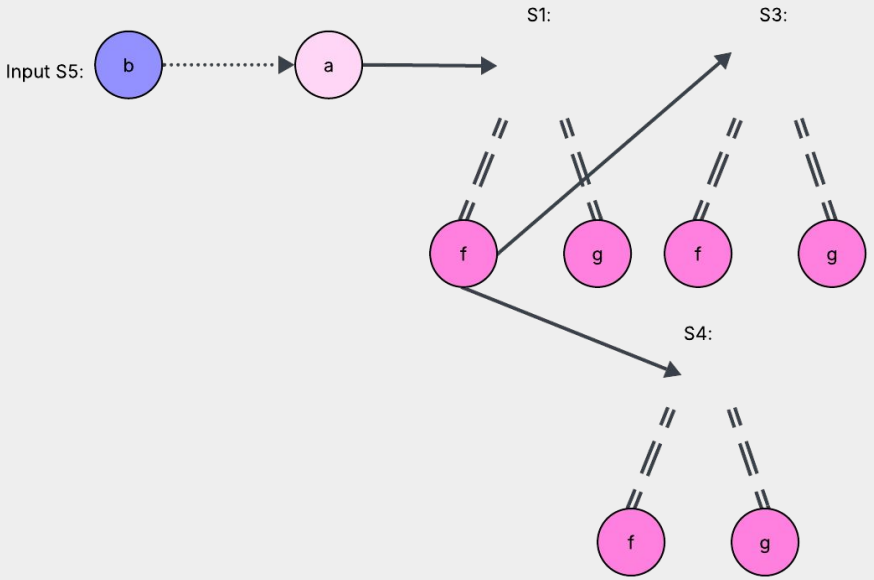
Intraprocedural Analysis

```
S1: T1 a = new T1(...);
S2: T1 b = a;
if()
    S3: a.f = new T1(...);
else
    S4: b.f = new T1(...);
s5: a = b.f;
```



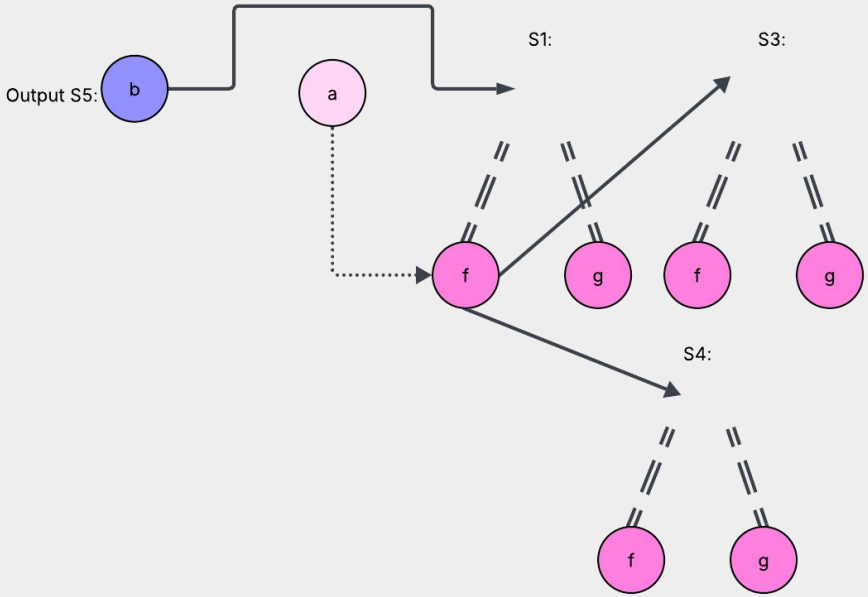
Intraprocedural Analysis

```
S1: T1 a = new T1(...);
S2: T1 b = a;
if()
    S3: a.f = new T1(...);
else
    S4: b.f = new T1(...);
------------
s5: a = b.f;
```



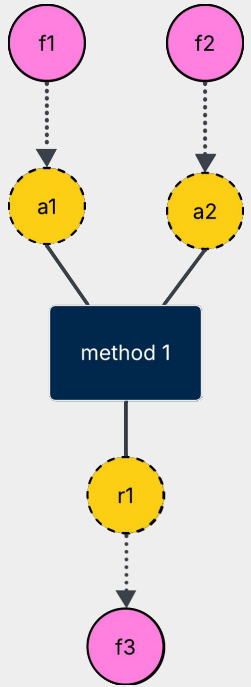
Intraprocedural Analysis

```
S1: T1 a = new T1(...);
S2: T1 b = a;
if()
    S3: a.f = new T1(...);
else
    S4: b.f = new T1(...);
s5: a = b.f;
```



Interprocedural Analysis

- Static analysis for multiple methods
- **Key Idea**: combine summary info created from intraprocedural analysis
- Iterate over call graph in reverse topological order until data flow solution converges



Reachability Analysis

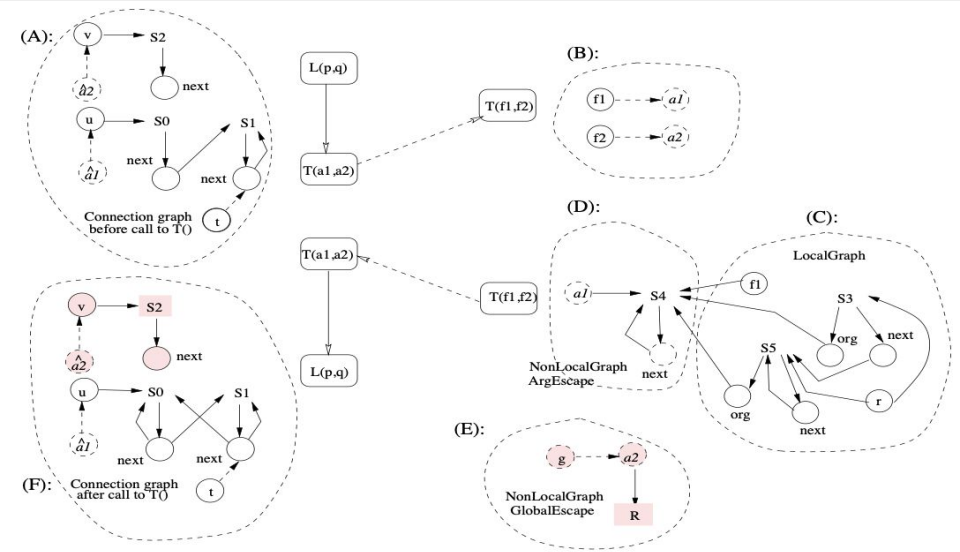


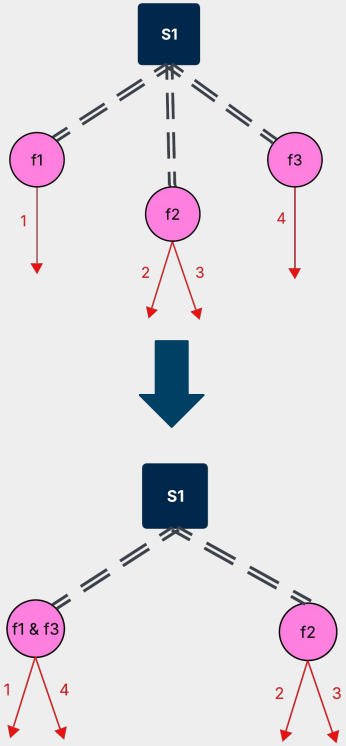
Figure 5: Connection graphs at various points in the call graph. Nodes that escape globally are shadowed.

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

4. Results

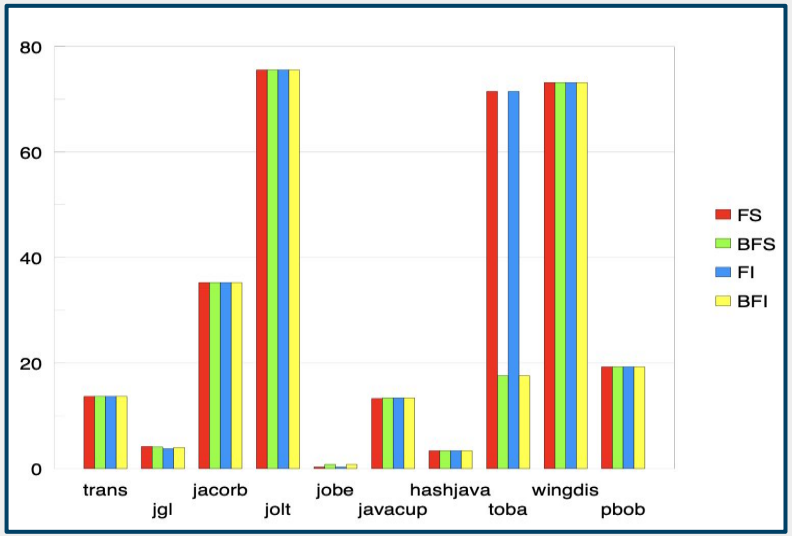
Experimental Design

- Four types of escape analysis are tested
  - Flow Sensitive (FS)
  - Flow Sensitive with Bounded Field Nodes (BFS)
  - Flow Insensitive (FI)
  - Flow Insensitive with Bounded Field Nodes (BFI)
- 10 medium- and large-sized benchmarks
- 333 MHz uniprocessor, 1 MB L2 Cache, 512 MB Memory

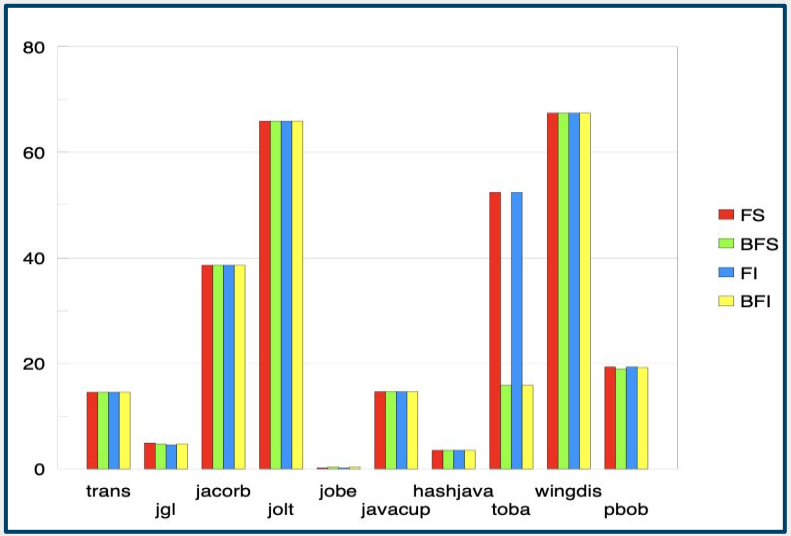


Stack Allocation Results

% User Local Objects Allocated on the Stack

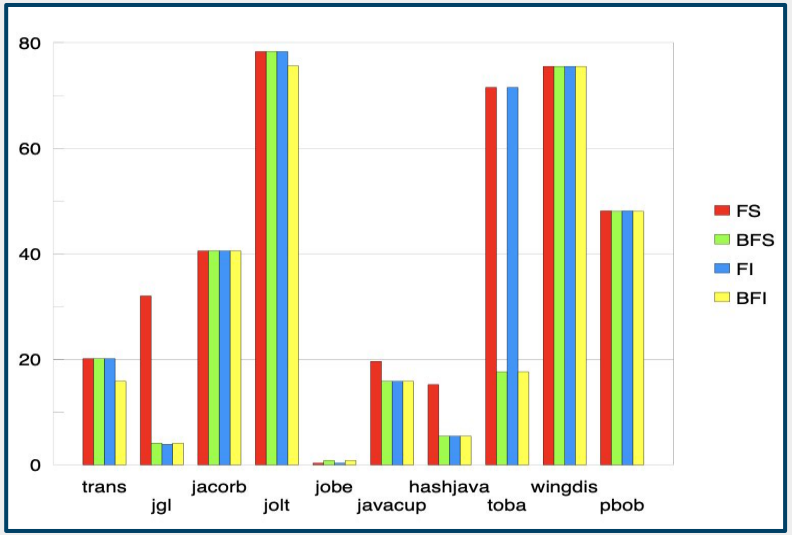


% User Local Object Space Allocated on the Stack

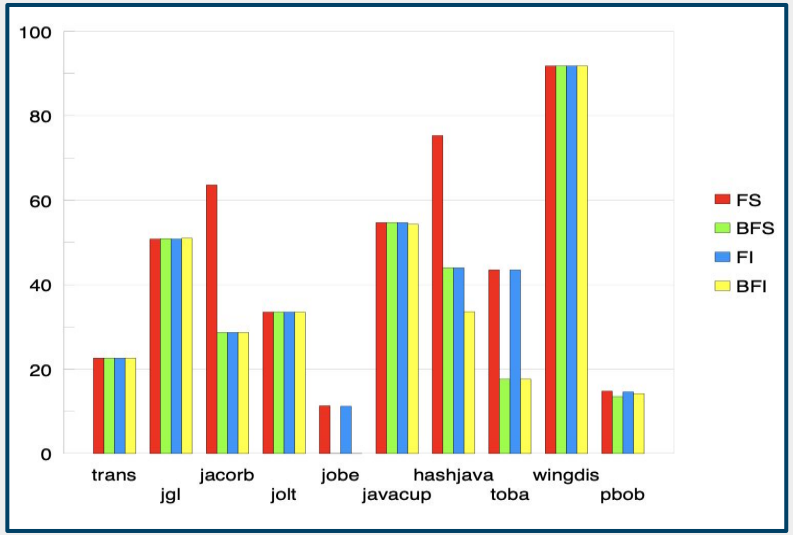


Locking Elimination Results

% Thread Local Objects



% Thread Locks Removed



Takeaways

- Number of objects allocated to the stack is highly dependent on specific programs
- Proposed methods remove 11% - 92% of lock operations
  - Remove > 50% in half of benchmarks
- Most savings come from removing lock operations

Program	Execution time (sec)	percentage reduction
trans	5.2	7 %
jgl	18.8	23 %
jacorb	2.5	6 %
jolt	6.8	4 %
jobe	9.4	2 %
javacup	1.4	6 %
hashjava	6.4	5 %
toba	4.0	16 %
wingdis	18.0	15 %
pbob	N/A	6 %

Table 3: Improvements in execution time

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

5. Strengths and Weaknesses

Strengths and Weaknesses

Strengths	Weaknesses
• Improves Java's performance in a wide variety of cases • Strong empirical evidence • Formalized arguments	• No mention of added compile time ○ Poor worst-case complexity • No ablations analysis • Limited benchmark diversity

6. Related Works

Impact on Modern Systems

- **Directly implemented in many modern JVMs**
  - **HotSpot JVM**
- Inspired implementation of languages with automatic memory management
  - **Go, Swift, C#**
- Escape analysis is still used to eliminate locks in languages like **Kotlin**

7. Conclusion

Conclusion

- Authors propose a **technique for memory and synchronization management**
- While modern techniques are more advanced, **the core ideas are still widely used today**
- Increased popularity of memory intensive workloads and use of concurrency adds significance to this area of research

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Thank You!