Escape Analysis for Java

J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar,
S. Midkiff, IBM T.J. Watson Research

OOPSLA ‘99

December 12, 1999

Outline

- Motivation
- Contributions
- Analysis Framework
- Algorithm
- Exceptions and Finalization in Java
- Experimental Results
- Conclusions

Motivation

- Stack allocation vs. heap allocation
  - No garbage collection overhead
  - No synchronization
  - More optimization possibilities
- Lock elimination
  - Reduce synchronization overhead
  - Allocation in thread's local memory
  - Possibly avoid memory flush

Contribution

- Interprocedural framework
- Escape analysis for lock elimination
- Exception handling
- Connection graph abstraction
- Extensive experimental data

Escape Analysis Framework

- Definitions
  - Thread escape - object instance *O* escapes thread *T* if visible by other thread
  - Method escape - object instance *O* escapes method *M* if the lifetime of *O* exceeds the lifetime of *M*

Framework, cont.

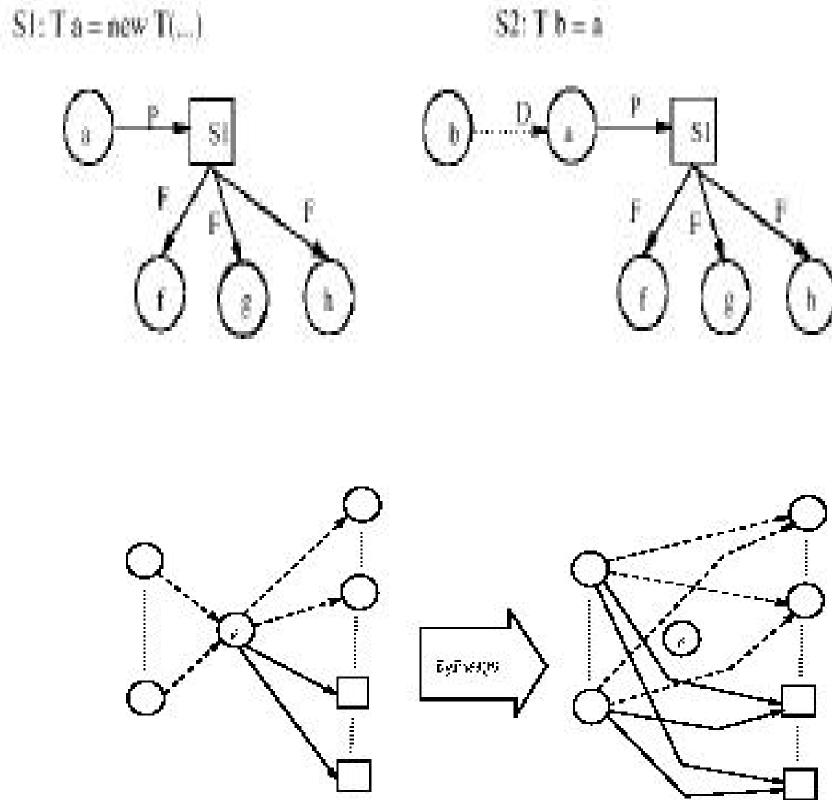


Figure 2: Illustrating *ByPass*(*p*) function

- Connection graph
  - 4 types of nodes - Reference, Object, Field, and Global nodes
  - 3 types of edges - Points to, Deferred, and Field
- *ByPass* function

Algorithm Outline

- An iterative dataflow algorithm
- Dataflow elements:
  CG annotated with *EscapeStates*
- Dataflow equations:

$$C_o^s = f^s(C_i^s), \qquad C_i^s = \wedge r\ Pred(s)(C_o^r)$$

- Until dataflow solution converges
  Iterate over methods in reverse topological order

Intraprocedural Analysis

- Basic statements
  - **p = new *t*()**
    - Create new object node *O*
    - Apply *ByPass(p)* (for FS analysis)
    - Add a points-to edge from *p* to *O*
  - **p = q**
    - Apply *ByPass(p)* (for flow-sensitive)
    - Add a deferred edge from *p* to *q*

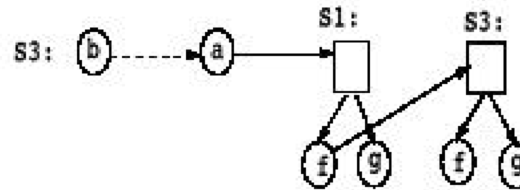
Intraprocedural Analysis

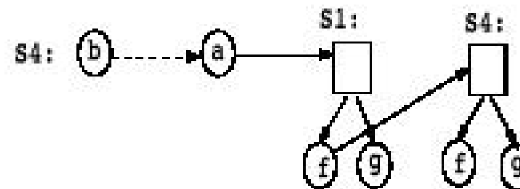
– **p.f = q**
  - Find *U* = *PointsTo(p)*
  - Find *V* = {*v* s.t. *u* -> *v*, *u* *U*, and *fid(v)* = *f*}
  - Add deferred edge from each *v* to *q*

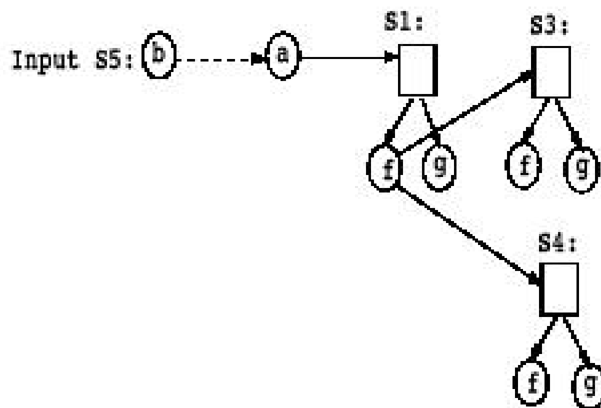
– **p = q.f**
  - Find *U* = *PointsTo(q)*
  - Find *V* = {*v* s.t. *u* -> *v*, *u* *U*, and *fid(v)* = *f*}
  - Apply *ByPass(p)* (for FS analysis)
  - Add deferred edge from *q* to each *v*

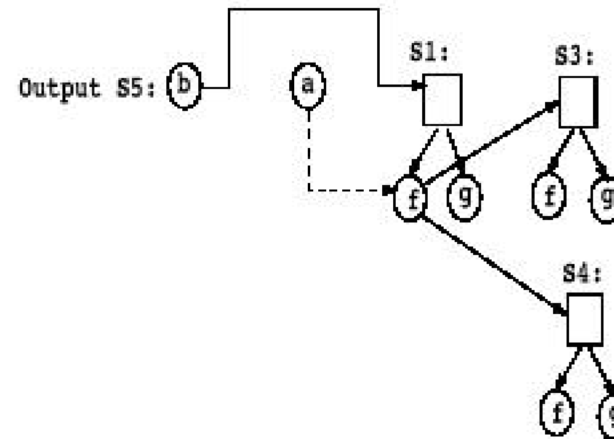
Intraprocedural Analysis, Example

```
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

- Connection graph at method entry
  - Process $f_i = a_i$, for each formal f_i; a_i is a phantom reference node
  - $EscapeState[f_i] = NoEscape$,
  - $EscapeState[a_i] = ArgEscape$
- Connection graph at method exit
  - *return=ref*, for each returned reference
  - *ByPass* each reference node
  - Do reachability analysis over CG of *M*

Reachability Analysis

- At method exit!
- A two phase transitive closure
  1. Find globally escaping nodes
  2. Find argument escaping nodes
- Partition CG of *M* into three parts
  – Global - set reachable from a *GlobalEscape* node
  – Arg - set reachable from an *ArgEscape* node
  – Local - set which retains *NoEscape*

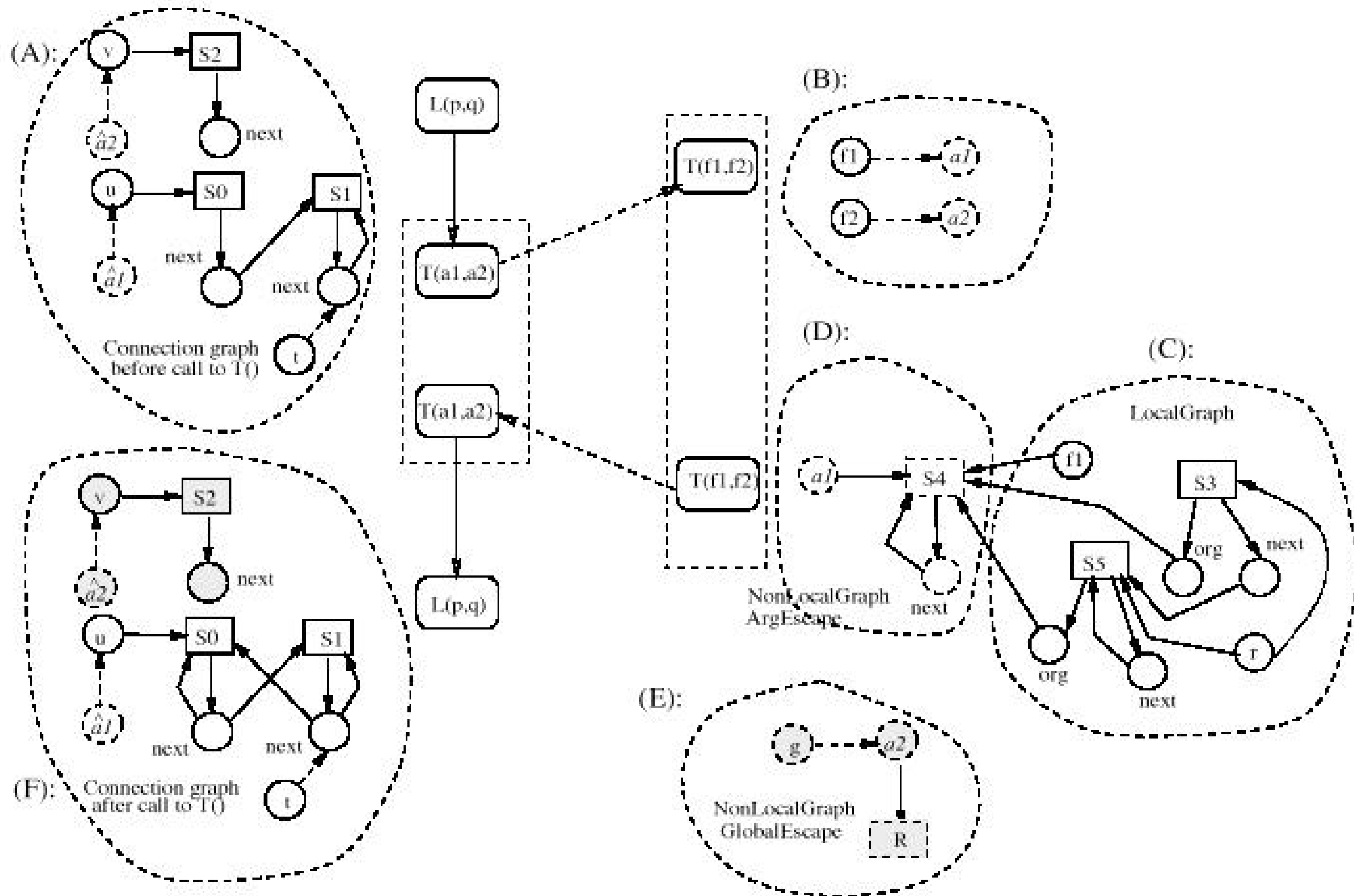
Interprocedural Analysis, cont.

- Connection graph before a call site
  - Process $\hat{a}_i = u_i$,
    $\hat{a}_i$ is a dummy node, u_i is an actual
  - $\hat{a}_1$ and u_1 reference the receiver object
- Connection graph after a call site
  - "Map" $\hat{a}_i$'s to a_i's
  - Add nodes to caller
  - Add edges to caller

MapsTo

- Propagates effects of the callee to caller
- Recursive rules
  - a_i -> $\hat{a}_i$
  - O_p -> O_q O_p *PointsTo(p)*, O_q *PointsTo(q)*,
    p and *q* are a_i, and $\hat{a}_i$,
    or *p* and *q* are corresponding fields in
    two mapping objects O_p and O_q

Example



Java Special Features

- Bottom methods
  - Most conservative CG, using type information
- Exceptions
  - Kill edges from "*try*-local" nodes only
- Finalization
  - If object class overrides *finalize* - mark *GlobalEscape*

Experiments

- IBM HP (static) Java Compiler
- 333MHz Power PC, AIX 4.1.5, 1 Mb L2, 512 Mb RAM
- 10 medium to large sized Java programs
- Two optimizations
- Four variants of the algorithm
  - Flow Sensitive
  - Bounded Field FS - *k mod m*, *m=3*
  - Flow Insensitive
  - Bounded Field FI

Benchmarks

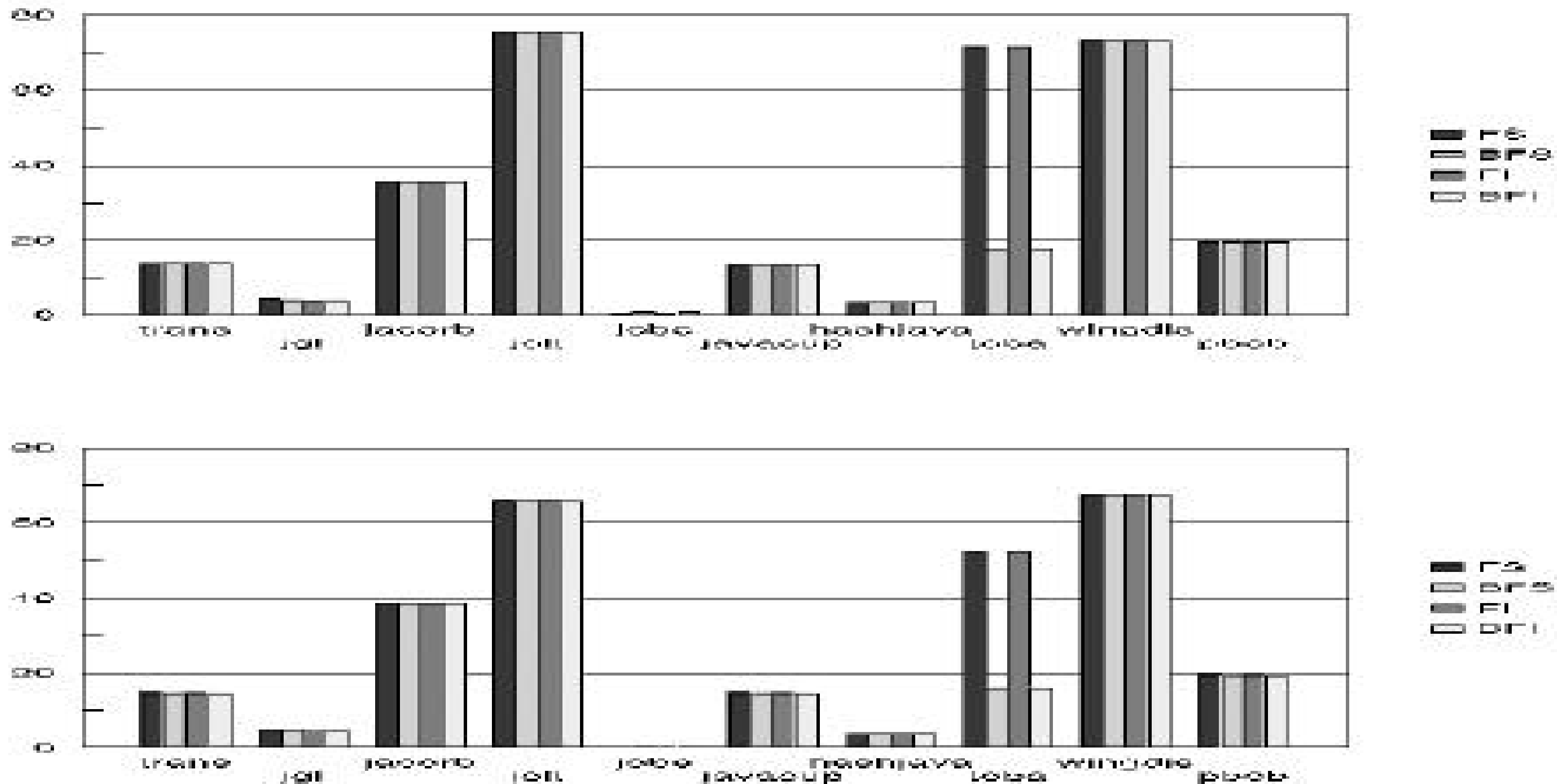
Program	Number of classes	size of classes	Number of objects allocated		Size of objects in bytes allocated		Total number of locks	
			user	user + library	user	user + library	user	user + library
trans	142	503K	263K	727K	7656K	31333K	868K	885K
jgl	135	217K	3808K	4157K	124409K	139027K	10391K	10434K
jacorb	436	308K	103K	48036K	2815K	3423323K	546K	672K
jolt	46	90K	94K	593K	3006K	17511K	1030K	1348K
jobe	46	60K	204K	339K	7957K	13331K	77K	106K
javacup	59	101K	67K	330K	1672K	8454K	191K	287K
hashjava	98	183K	173K	248K	4671K	827K	158K	165K
toba	19	86K	154K	2201K	5878K	59356K	1060K	1246K
wingdis	48	178K	840K	2561K	25902K	92238K	2105K	2299K
pbob	65	333K	19787K	48206K	639980K	2749520K	35691K	171189K

Table 2: Benchmarks characteristics

Experimental Results

- Stack Allocation of *NoEscape* objects

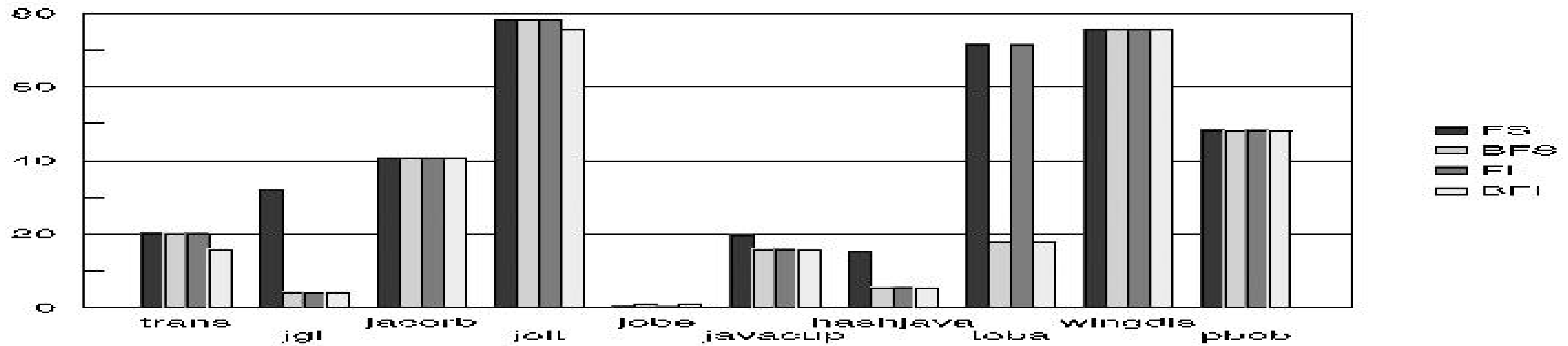
HPCJ native *alloca* routine

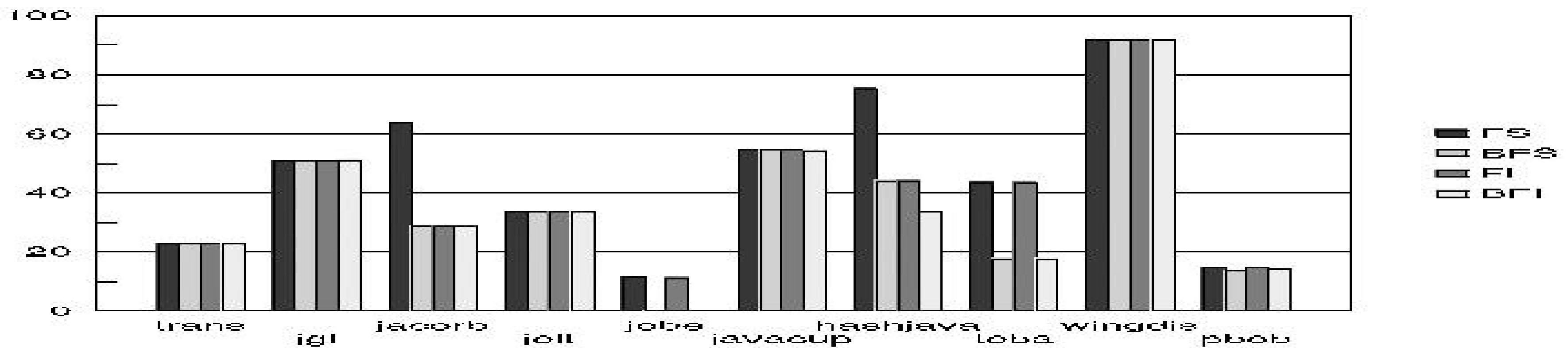


Experimental Results

- Lock Elimination

  At object use site bypass *compare &swap*





Conclusions

- 2 - 23% speedup, 8% avg
- Lots of stack-allocable objects
- 11 - 92% of locks eliminated
- FS as bad as FI?
- Very similar to point-to Graphs
- Need benchmarks with larger footprint
- Need algorithm time and space

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 %