Sharing Analysis:
Arrays, Collections, and Recursive Structures

Mark Marron, Mario Mendez-Lojo
Manuel Hermenegildo, Darko Stefanovic,
Deepak Kapur

Motivation

- Want to optimize object-oriented programs which make use of pointer rich structures
  - In an Array or Collection (e.g. java.util.List) are there any elements that appear multiple times?
  - Differentiate structures like compiler AST with/without interned symbols --- backbone is tree with shared symbol objects or a pure tree

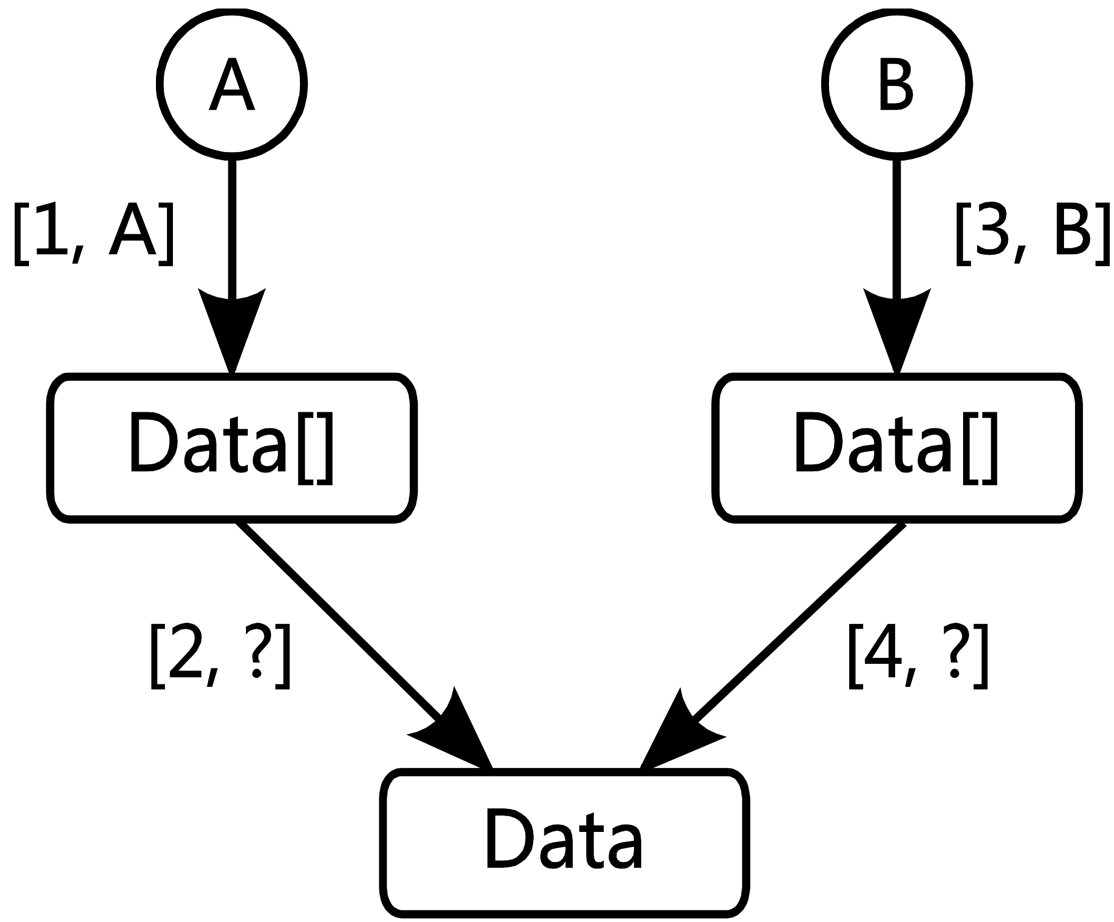
Motivation Cont.

- Ability to answer these sharing questions enables application of many classic optimizations
  - Thread Level Parallelization
  - Redundancy Elimination
  - Object co-location
  - Vectorization, Loop Unroll Schedule

Solution

- Start with classic *Abstract Heap Graph Model* and add additional instrumentation relations
  - Nodes represent sets of objects (or recursive data structures), edges represent sets of pointers
  - Has natural representation for data structures and connectivity properties
  - Naturally groups related sets of pointers
  - Efficient to work with
- Augment edges, which represent sets of pointers with additional information on the sharing relations between the pointers

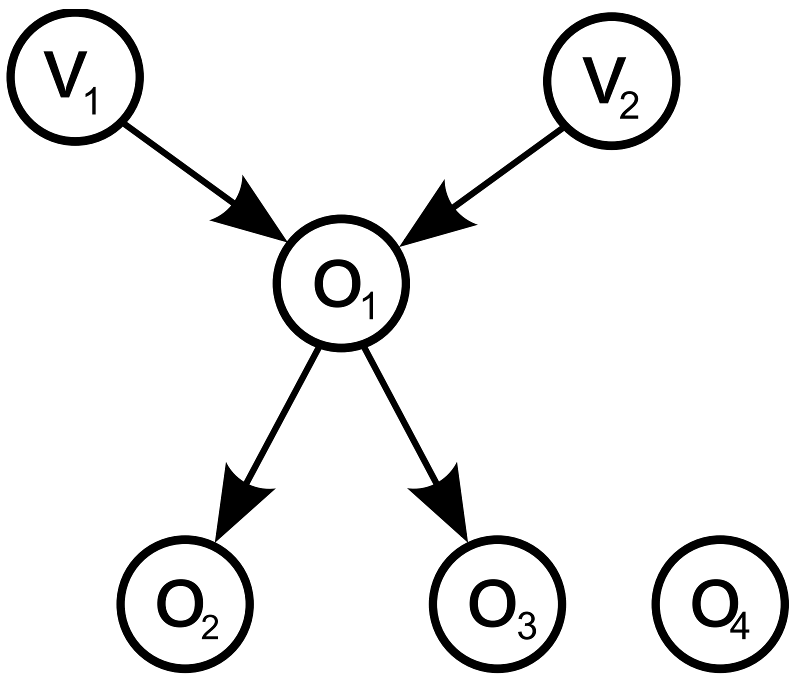
Example: Abstract Heap Graph

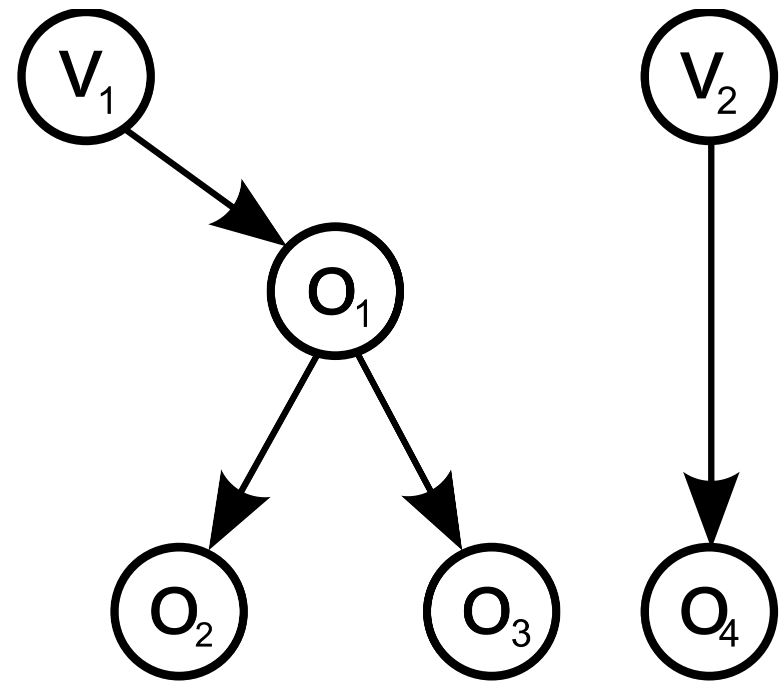


Concrete Sharing

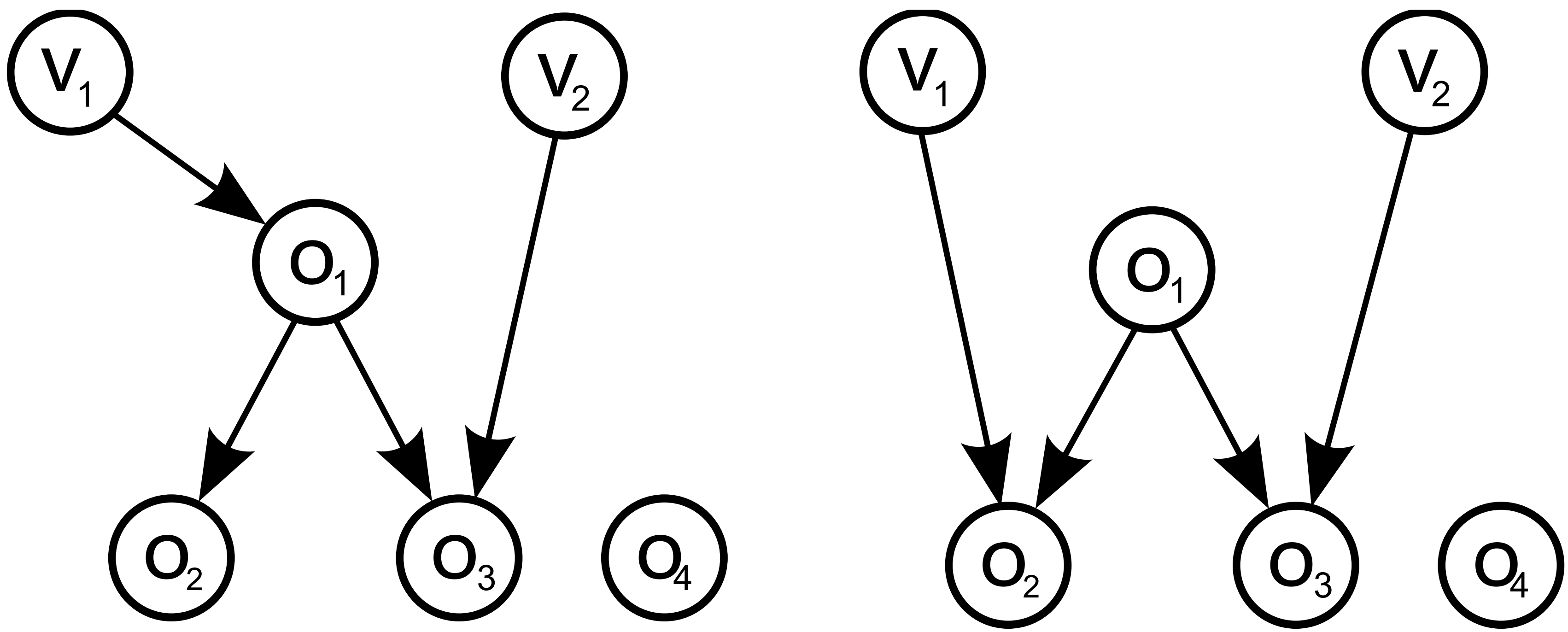
- Region of the heap (O, P, P_c)
  - O is a set of objects
  - P is the set of the pointers between them
  - P_c the references that enter/exit the region
- Given references r_1, r_2 in P_c pointing to objects o_1, o_2 respectively we say:
  - *alias:* $o_1 == o_2$
  - *related:* $o_1 != o_2$ but in same *weakly-connected* component
  - *unrelated:* o_1 and o_2 in different *weakly-connected* components

Sharing: Alias and Unrelated





Sharing: Related



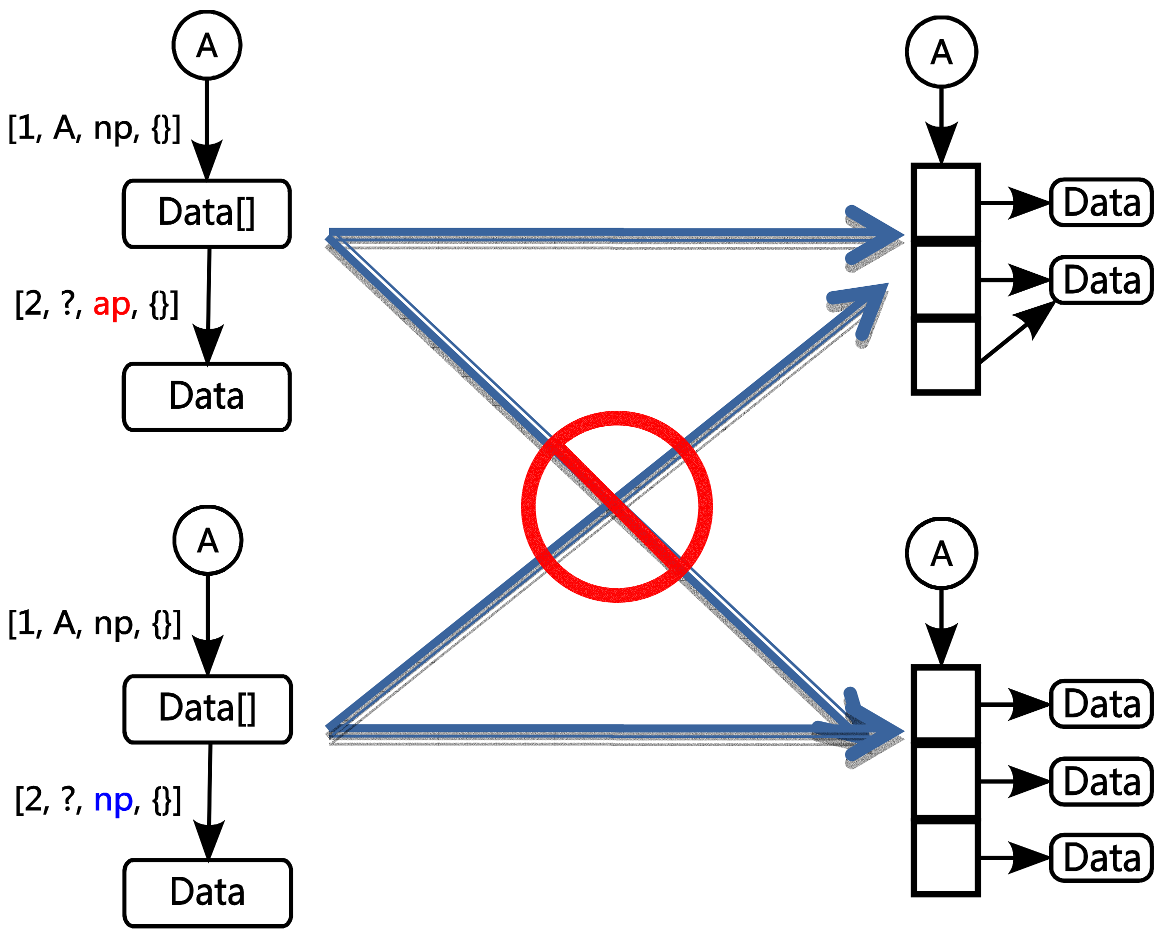
Abstract Representation

- Edges abstract sets of references (variable references or pointers)
- Introduce 2 related abstract properties to model sharing
  - *Interference:* Does a single edge (which abstracts possible many references) abstract only references with disjoint targets or do some of these references alias/related?
  - *Connectivity:* Do two edges abstract sets of references with disjoint targets or do some of these references alias/related?

Interference

- For a single edge how are the targets of the references it abstracts related
- Edge e is:
  - *non-interfering:* all pairs of references r_1, r_2 in $\gamma(e)$ must be *unrelated* (there are none that *alias* or are *related*).
  - *interfering:* all pairs of references r_1, r_2 in $\gamma(e)$, may either be *unrelated* or *related* (there are none that *alias*).
  - *share:* all pairs of references r_1, r_2 in $\gamma(e)$, may be *aliasing*, *unrelated* or *related*.

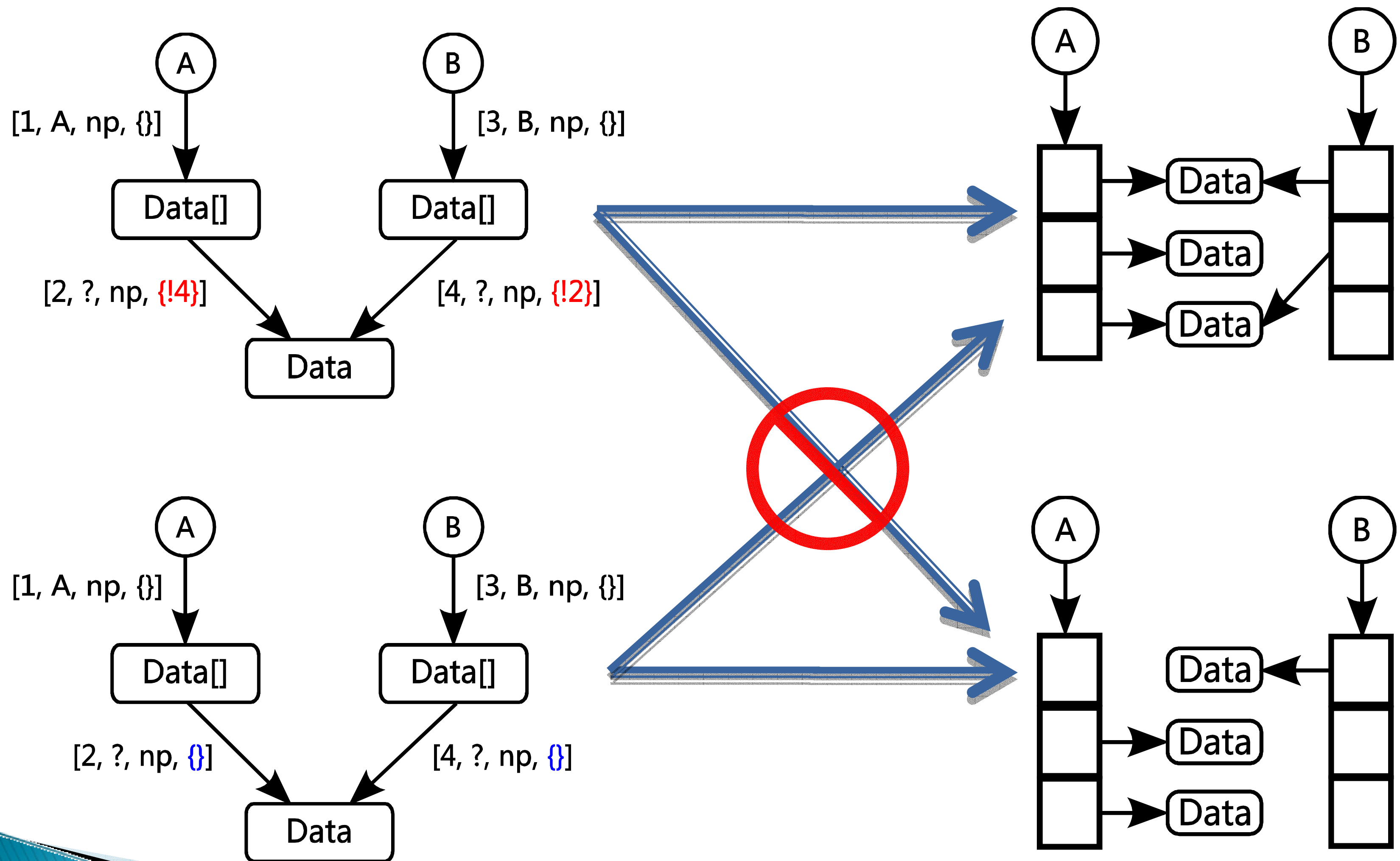
Interference Example



Connectivity

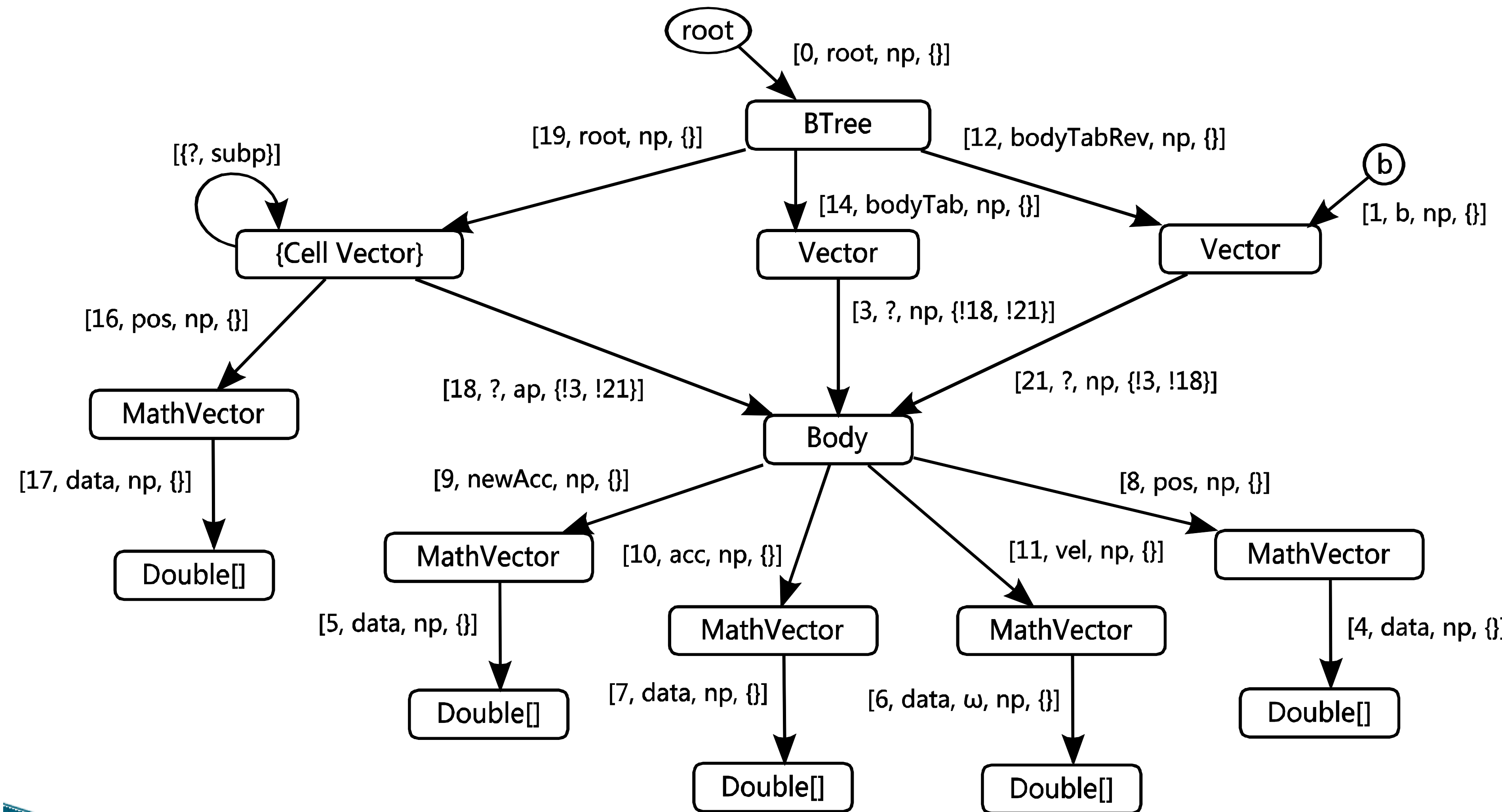
- For two different edges how are the targets of the references they abstract related
- Edges e_1, e_2 are:
  - *disjoint:* all pairs of references r_1 in $\gamma(e_1)$, r_2 in $\gamma(e_2)$ are *unrelated* (there are none that *alias* or are *related*).
  - *connected:* all pairs of references r_1 in $\gamma(e_1)$, r_2 in $\gamma(e_2)$ may either be *unrelated* or *related* (there are none that *alias*).
  - *share:* all pairs of references r_1 in $\gamma(e_1)$, r_2 in $\gamma(e_2)$ may be *aliasing*, *unrelated* or *related*.

Connectivity Example



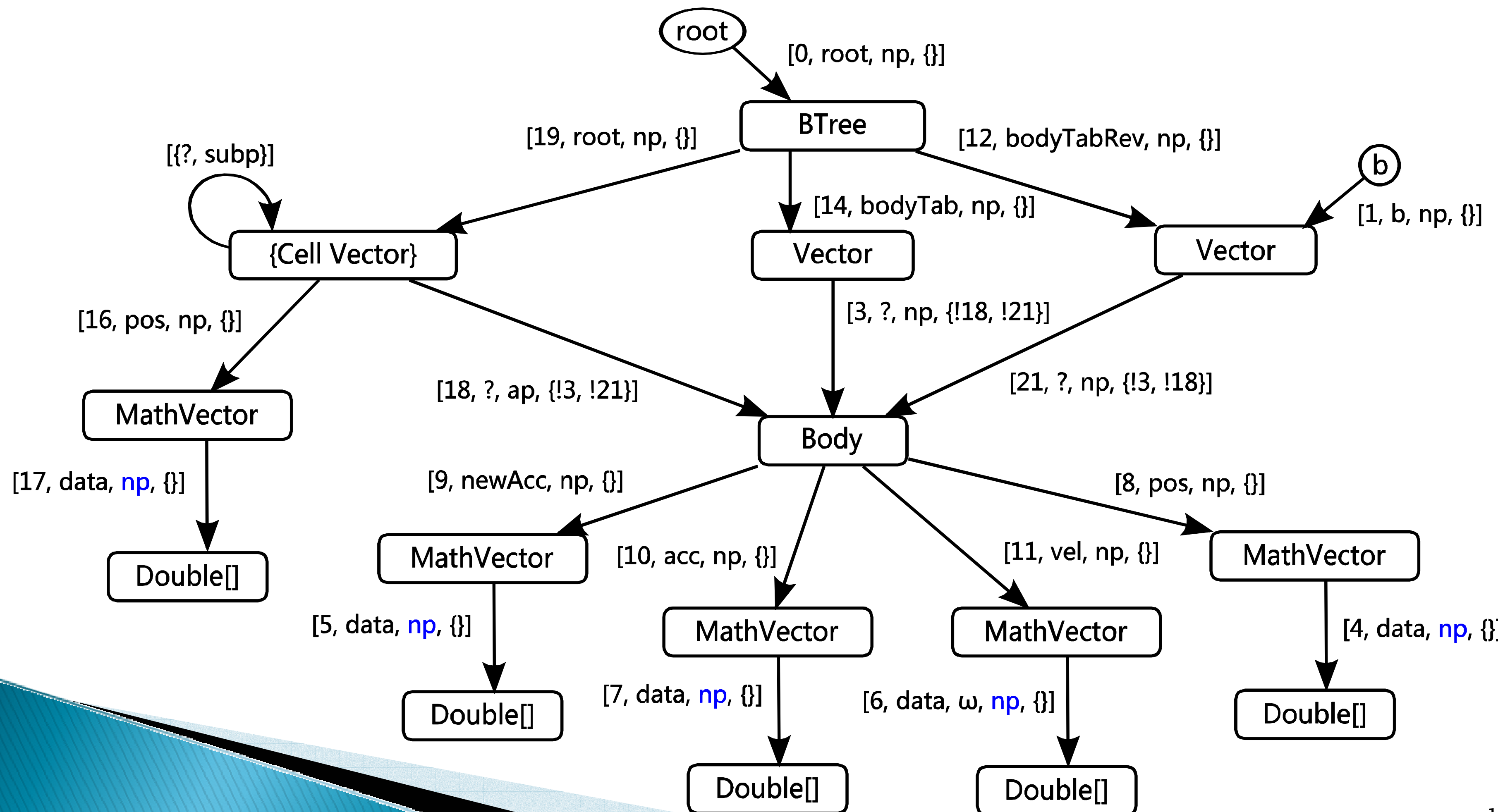
Case Study BH (Barnes-Hut)

- N-Body Simulation in 3-dimensions
- Uses Fast Multi-Pole method with space decomposition tree
  - For nearby bodies use naive n^2 algorithm
  - For distant bodies compute center of mass of many bodies and treat as single point mass
- Dynamically Updates Space Decomposition Tree to Account for Body Motion
- Has not been successfully analyzed with other existing shape analysis methods


root
[0, root, np, {}]
BTree
[19, root, np, {}]
[12, bodyTabRev, np, {}]
[{?, subp}]
b
[14, bodyTab, np, {}]
[1, b, np, {}]
{Cell Vector}
Vector
Vector
[16, pos, np, {}]
[3, ?, np, {!18, !21}]
[18, ?, ap, {!3, !21}]
[21, ?, np, {!3, !18}]
MathVector
Body
[17, data, np, {}]
[9, newAcc, np, {}]
[8, pos, np, {}]
MathVector
[10, acc, np, {}]
[11, vel, np, {}]
MathVector
Double[]
[5, data, np, {}]
MathVector
MathVector
[4, data, np, {}]
Double[]
[7, data, np, {}]
[6, data, ω, np, {}]
Double[]
Double[]
Double[]
Double[]

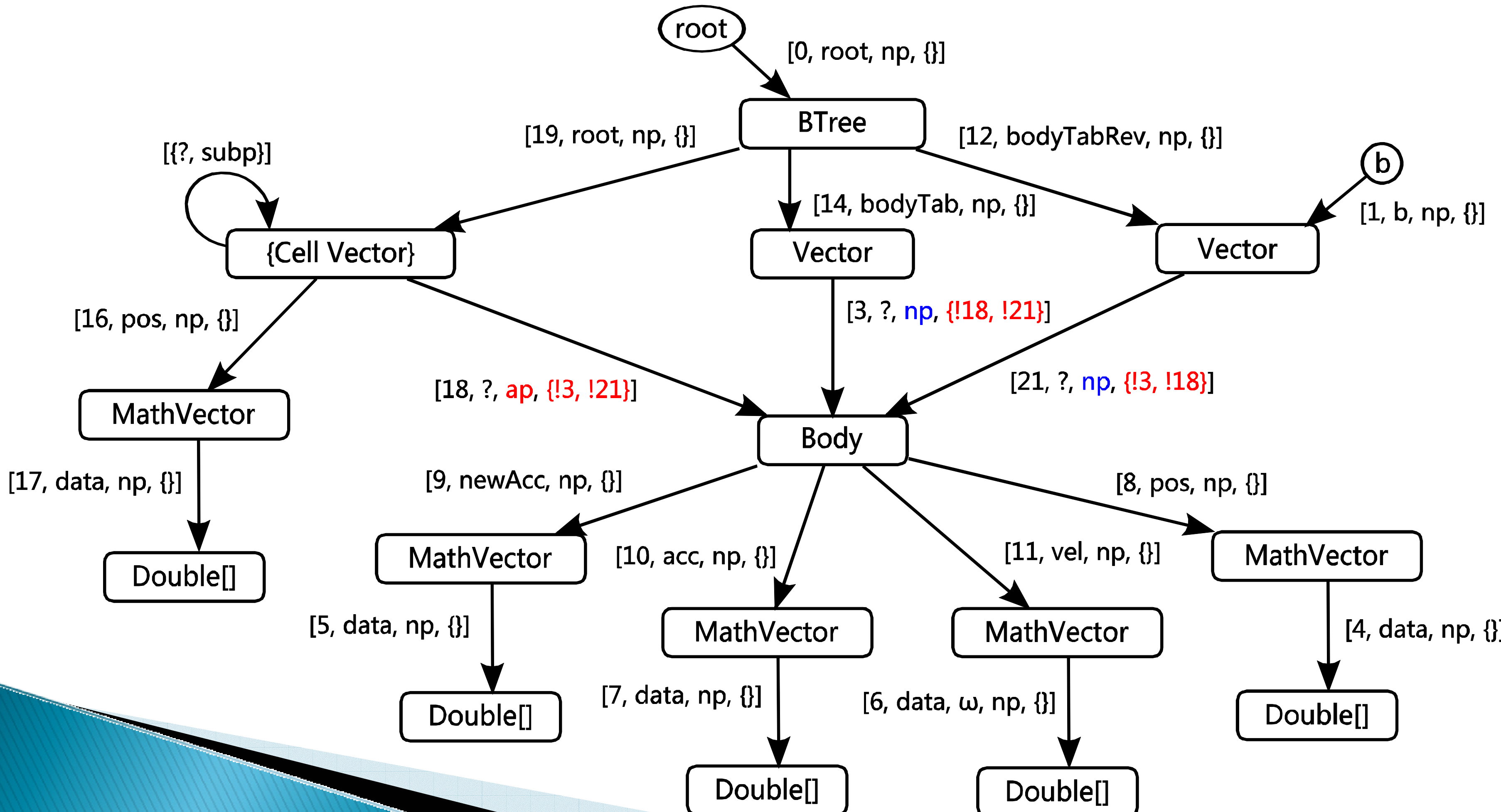
BH Optimizations Memory

- Inline *Double[]* into *MathVector* objects, 23% serial speedup 37% memory use reduction

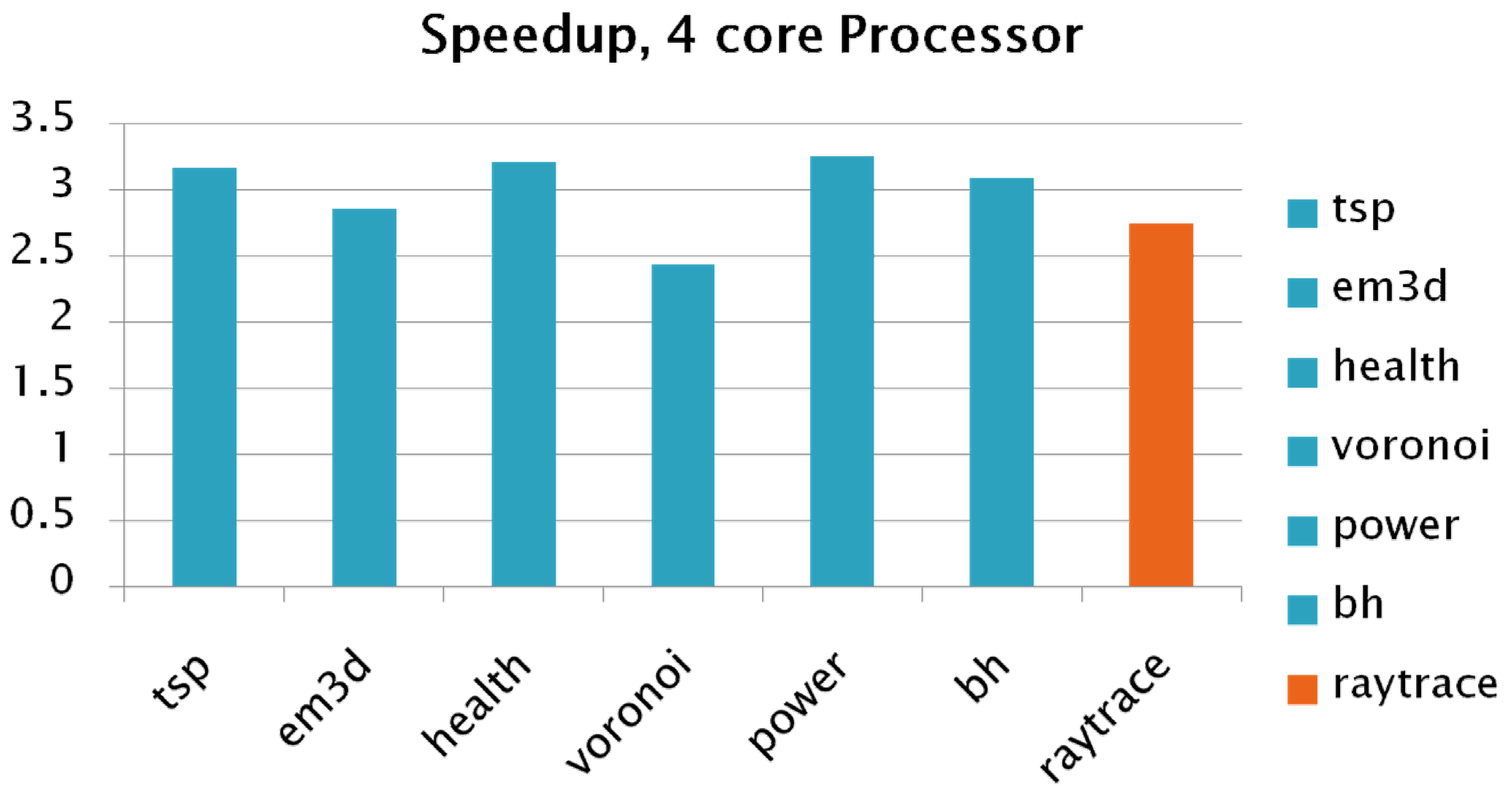


BH Optimizations TLP

- TLP update loop over *bodyTabRev*, factor 3.09 speedup on quad-core machine



General TLP Results



Benchmark Analysis Statistics

Benchmark	LOC	Analysis Time
bisort	560	0.26s
mst	668	0.12s
tsp	910	0.15s
em3d	1103	0.31s
health	1269	1.25s
voronoi	1324	1.80s
power	1752	0.36s
bh	2304	1.84s
db	1985	1.42s
raytrace	5809	37.09s

Conclusions

- Presented a practical abstraction for modeling sharing in programs
- Allows us to accurately model how objects are stored arrays (or Collections from java.util)
- This information can be usefully applied to compiler optimizations
  - Thread-Level Parallelization
  - Vectorization or Loop Unrolling
  - Various memory locality optimizations

Demo of the (shape) analysis available at:

www.cs.unm.edu/~marron/software.html