Static Analysis for Identifying and Allocating Clusters of Immortal Objects

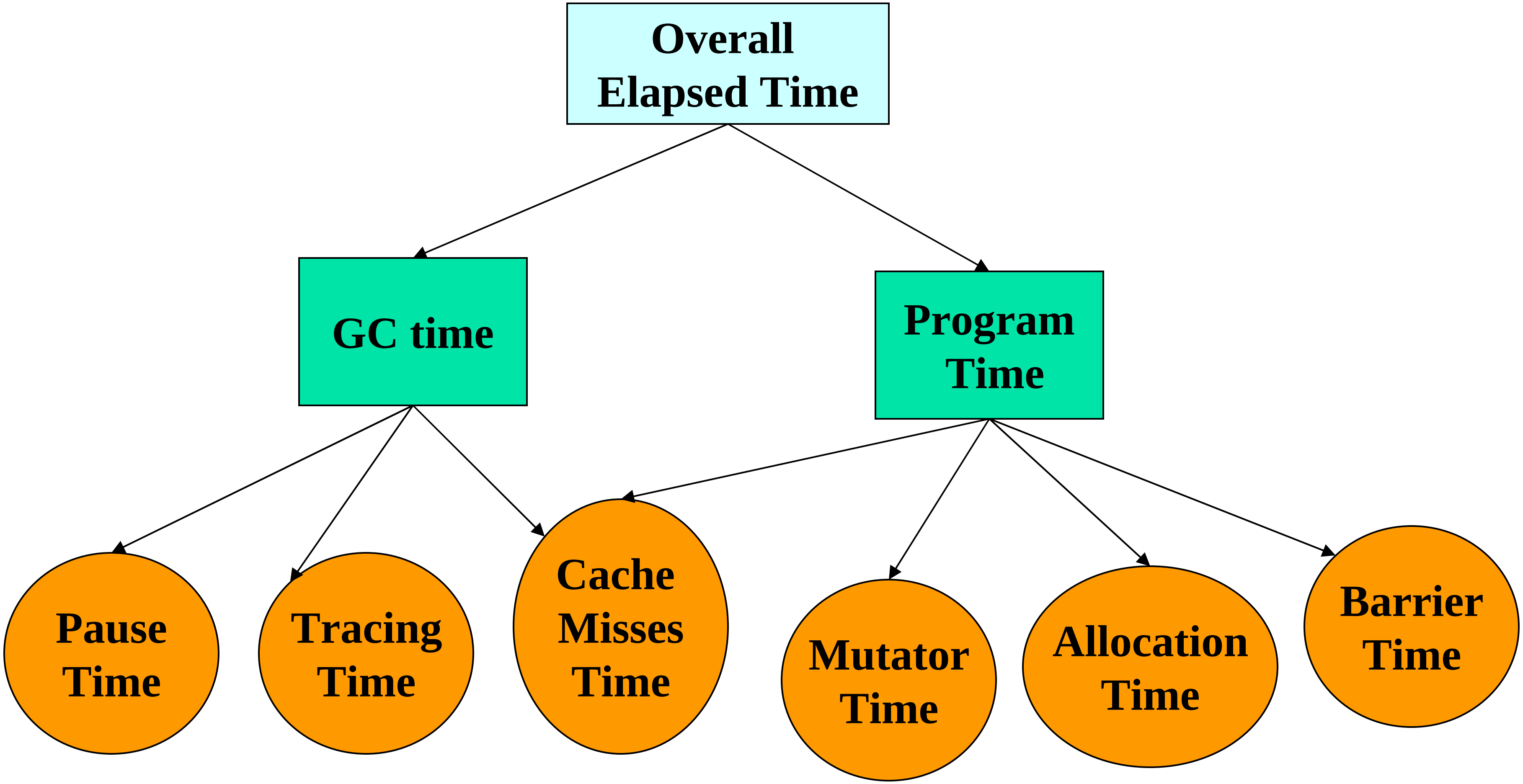
Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore

NPTEL Course on Compiler Design

Introduction

- Garbage collection (GC) is a necessity in modern O-O languages
  - Hides the problems of memory management from the programmer
- However, program incurs performance penalty due to GC
  - Generational GC and concurrent GC reduce overheads of garbage collection
- Cluster allocation reduces no. of collections and makes each collection more effective

Cost of a Tracing Collector

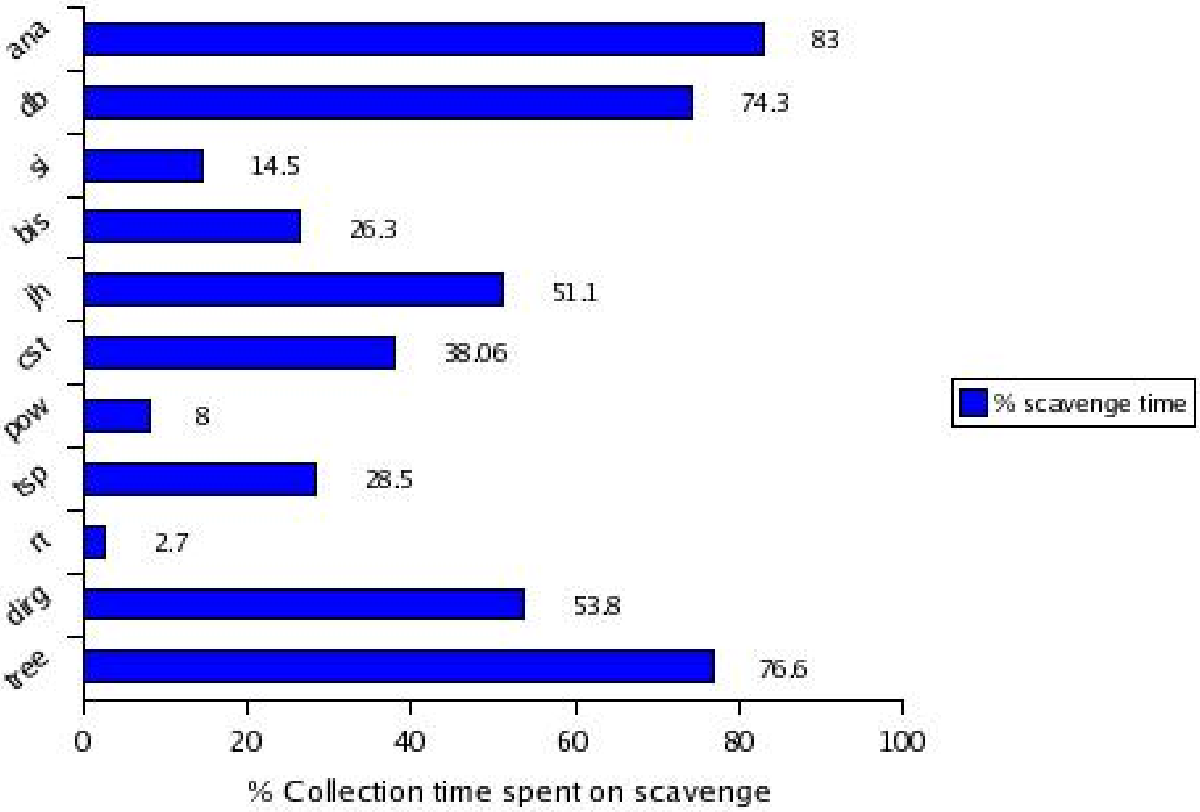


Past approaches to reduce cost of GC

- Generational collection
  - collect smaller areas at a time
- Concurrent GC
- Opportunistic collection
  - Key objects
  - schedule collection when program activity is low
- Compiler assistance for garbage collection
  - Region collection
  - Escape analysis and stack allocation

Impact of Long Living Objects on Garbage Collection

Scavenging long-life objects accounts for a significant part of GC time



Ways to Reduce Scavenge Time in Garbage Collection

- Compute life times of objects and bind them to the activation record of a method that uses them last
  - Handles volatile objects well, but not long living objects
  - Stack allocation instead of heap allocation
  - Cannot handle related objects that refer to each other from different methods, but die together (dynamic data structures)
- Detect long-living clusters and allocate separately

Cluster Allocation Highlights

- Identify clusters of long living objects
  - Allocate them in a separate mature object space, neither on the runtime stack nor on the normal heap
  - No GC on mature object space, recover whole space at method termination time
  - Avoids tracing and copying of long living objects during GC
- Reduces heap size
- Heap will now contain objects with shorter life times
  - Makes collections more effective and faster
- Compiler analysis to identify clusters
  - No runtime overheads, but conservative

The Approach

- Uses information about life time of objects to construct a Points-To-Escape graph (PTE graph)
  - Based on the Compositional pointer and escape analysis framework of Whaley and Rinard
- Longest living methods that contribute to long living clusters are identified using profiling
- Objects that do not escape the longest living methods are the roots of clusters
  - All objects reachable from the roots of clusters belong to the clusters
  - Roots of clusters are Key Objects (as proposed by Hayes)

The Approach (contd.)

- All cluster objects are statically allocated in a separate mature object space
- When the method binding the life time of the root objects returns, the entire cluster is garbage and can be reclaimed in its entirety
- Evaluation using a baseline GC that can run in stop-the-world and concurrent modes
  - Implemented in Rotor
  - Both cluster and stack allocation methods have been implemented, compared and evaluated

Compositional Pointer and Escape Analysis

- Determines for every allocation site A, the method M, whose stack frame will outlive the object created at A
- An object P escapes a method M if
  - it is a formal parameter or
  - a reference to P is written into a static variable
  - a reference to P is passed to one of the callees of M, say N, and we do not know what N did to P
  - M returns P
- If none of the above, then M captures P, and P does not live beyond M; P can be allocated on the Stack of M

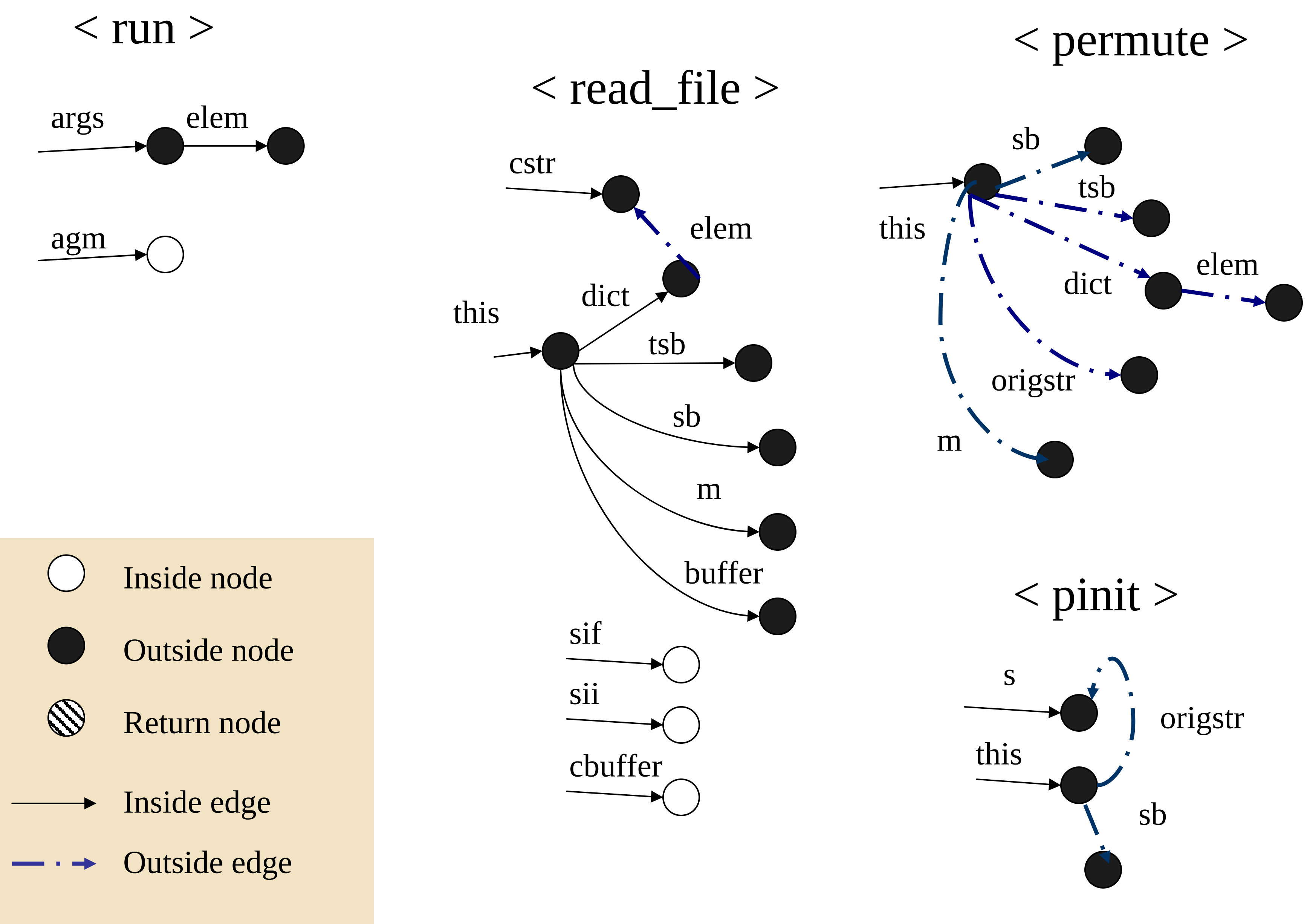
Escape Analysis (Contd.)

- Intra-procedural and inter-procedural algorithms
- Creates PTE (Points-To-Escape) graph
  - Allocated objects are nodes and references between objects are edges
  - Inside nodes (edges)*:* Objects (references) created within the currently analyzed region
  - Outside nodes (edges)*:* Objects (references) created outside the currently analyzed region. Nodes could become *Outside nodes* because of their access via an *Outside edge*

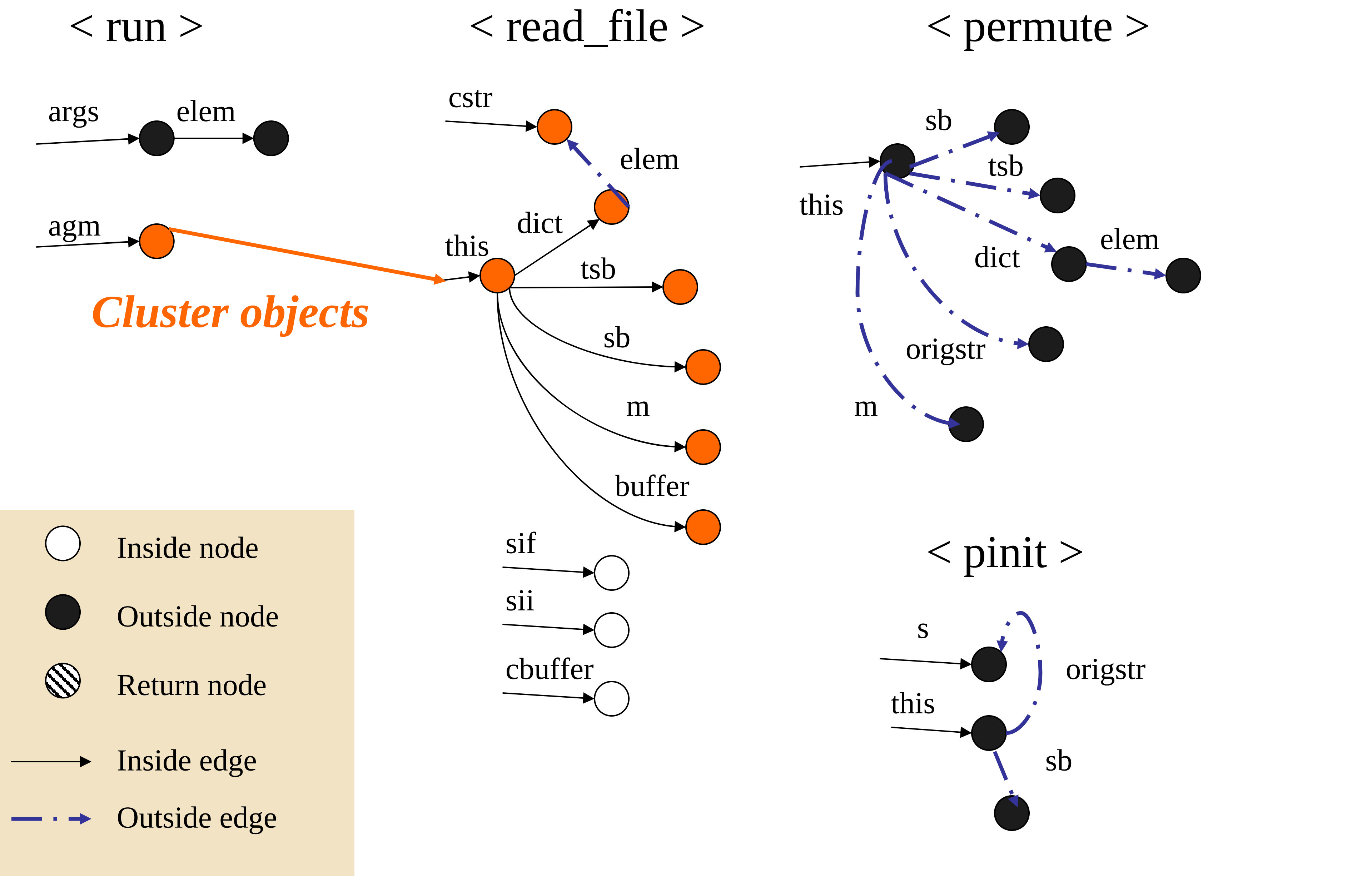
Escape Analysis (Contd.)

- Intra-procedural Algorithm for a method M
  - Incrementally computes PTE graph for M statement by statement
  - PTE graphs of some of the called methods may not be available at this stage
- Interprocedural Algorithm
  - Composes individual method (M) PTE graphs with those of the methods called from within M to form complete PTE graphs for M

An Example: _211_anagram



An Example: _211_anagram



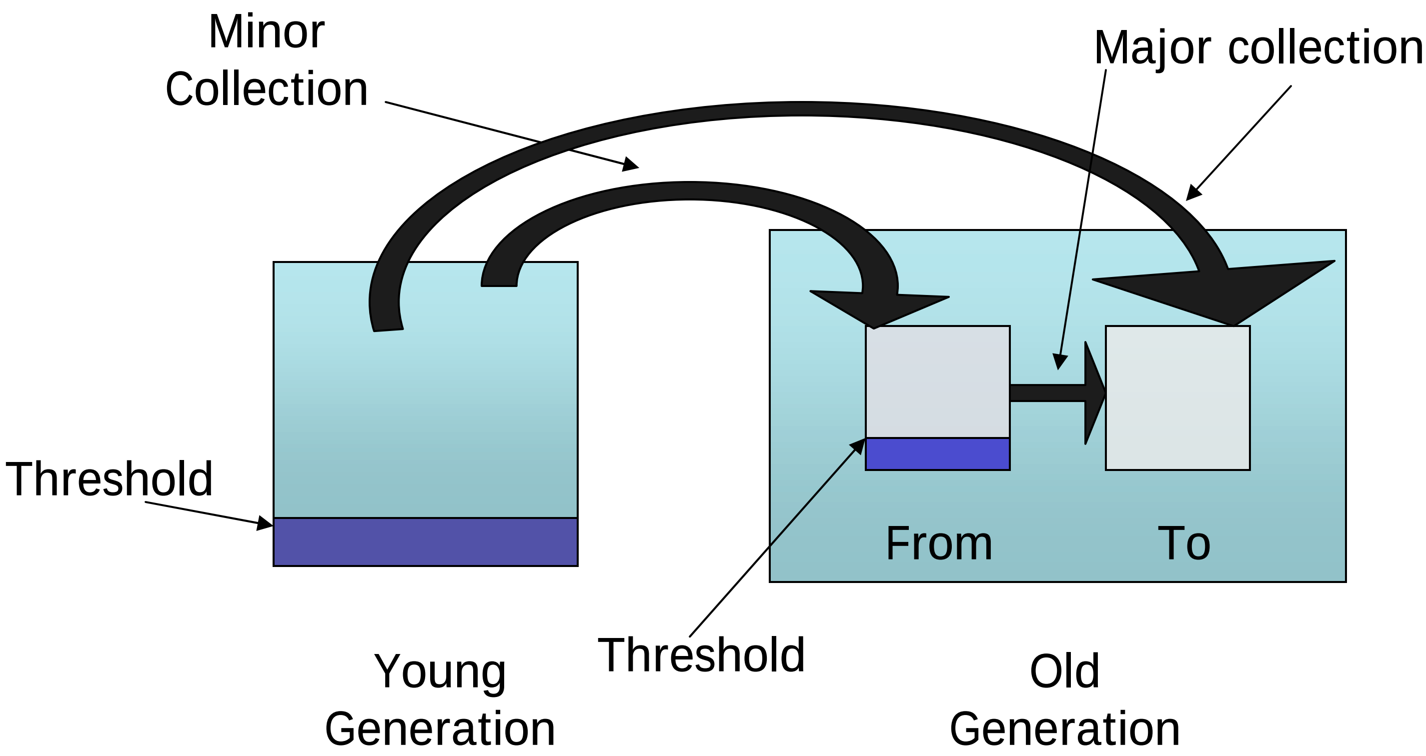
Cluster Identification

- Apply profiling and get a list of methods that have a long life and are close to main
- Emphasis is on those methods that live long *and* allocate objects that are potential *roots* of clusters
- Nodes with no incoming edges are *roots*
- Depth First Search on PTE graphs is used to identify clusters
  - Only edges corresponding to *new* statements are considered
  - All objects created by such statements are allocated using *new1*, instead of *new*, placing them in a separate mature object space

Concurrent Garbage Collector (baseline) for ROTOR

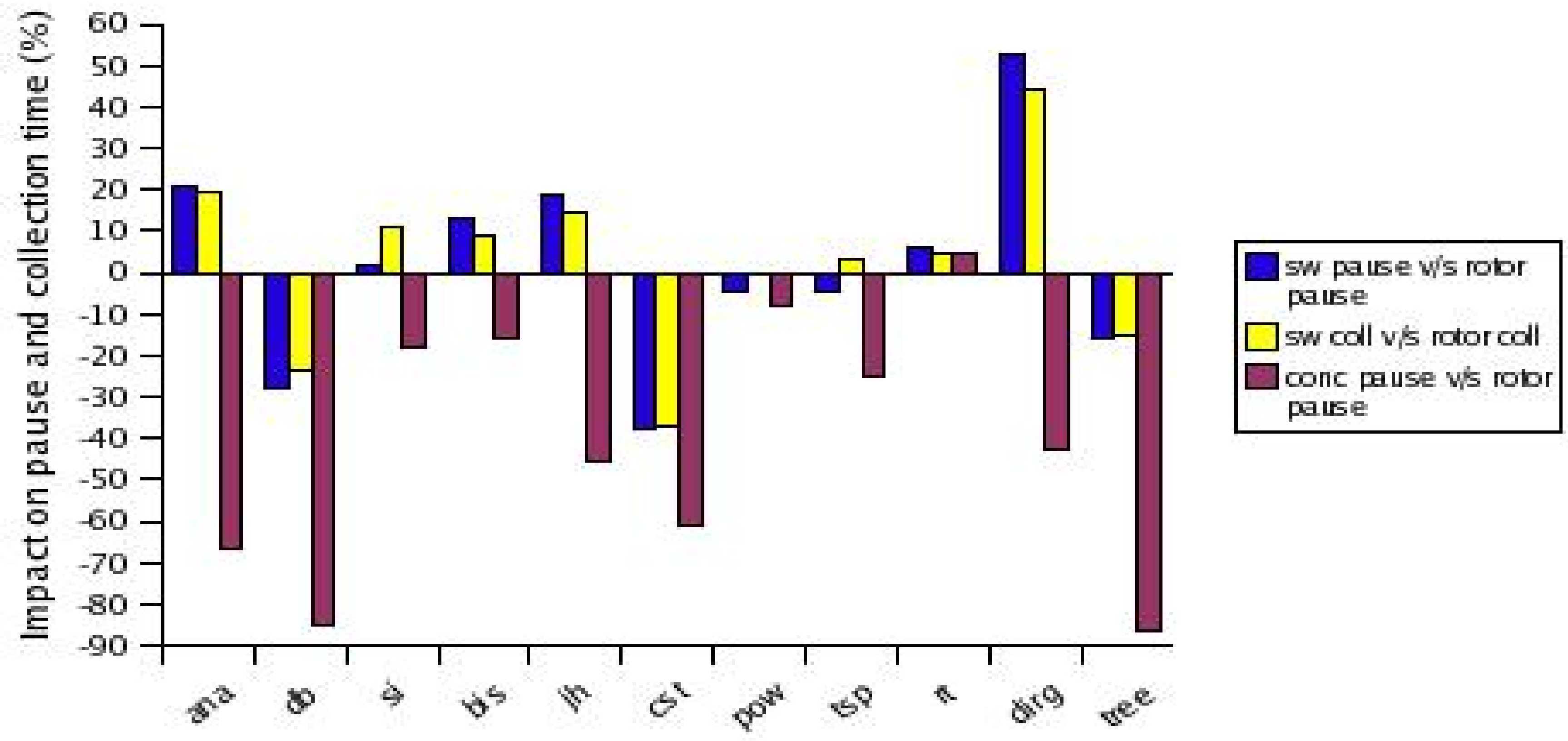
- Existing ROTOR GC is a Stop-The-World GC
- Ours is a generational concurrent GC
  - Permits mutator (application) and GC to run concurrently; GC is yet another thread
  - Based on the algorithm of Nettles & O'Toole
  - Two generations – Young and Old
    - Young stores 'recent' objects, most of which would become garbage quickly
    - Old stores 'permanent' objects
  - Has the same GC interface as ROTOR GC

Concurrent Generational Garbage Collector

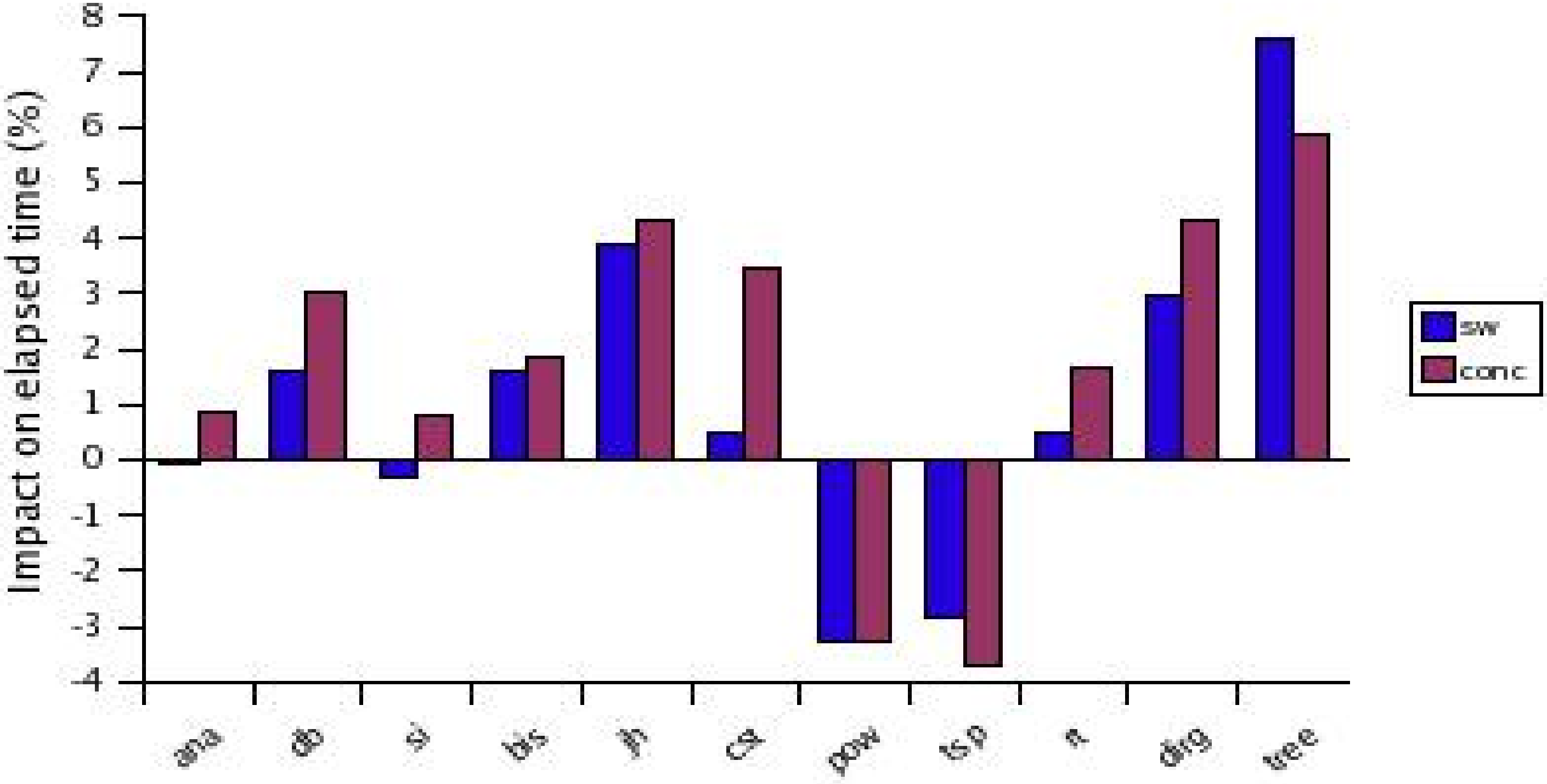


After major collection, From and To swap their roles

Performance of Concurrent GC: Pause and Collection Times



Performance of Concurrent GC: Elapsed Time



Performance of Clustering: Heap and Mature Object Spaces

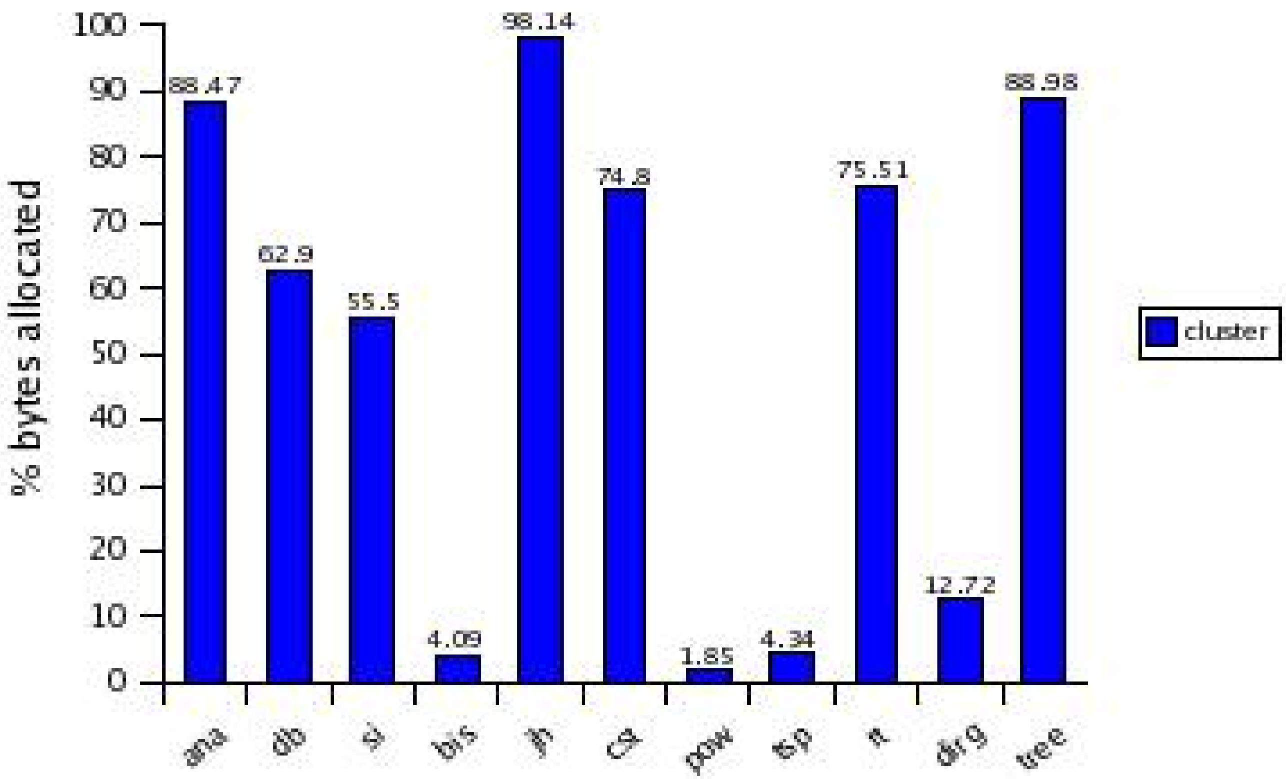
Average reduction in heap requirement is 12.6%

Program	Young gen/ Old gen-no clust. (MB)	Young gen/ Old gen-with clust. (MB)	Max clust. size (MB)
_211_anagram	2/8	0.7/1.4	3.8
_209_db	1/10	1/2	2.5
_208_cst	1/40	0.7/1.4	12.7
raytrace	0.8/1.6	0.8/1.6	3.6
treeadd	4/8	0.19/0.38	1.4

Performance of Clustering:
Fraction of Bytes Allocated in Mature Object Space

pow, tsp, and dirg will benefit from Stack Allocation

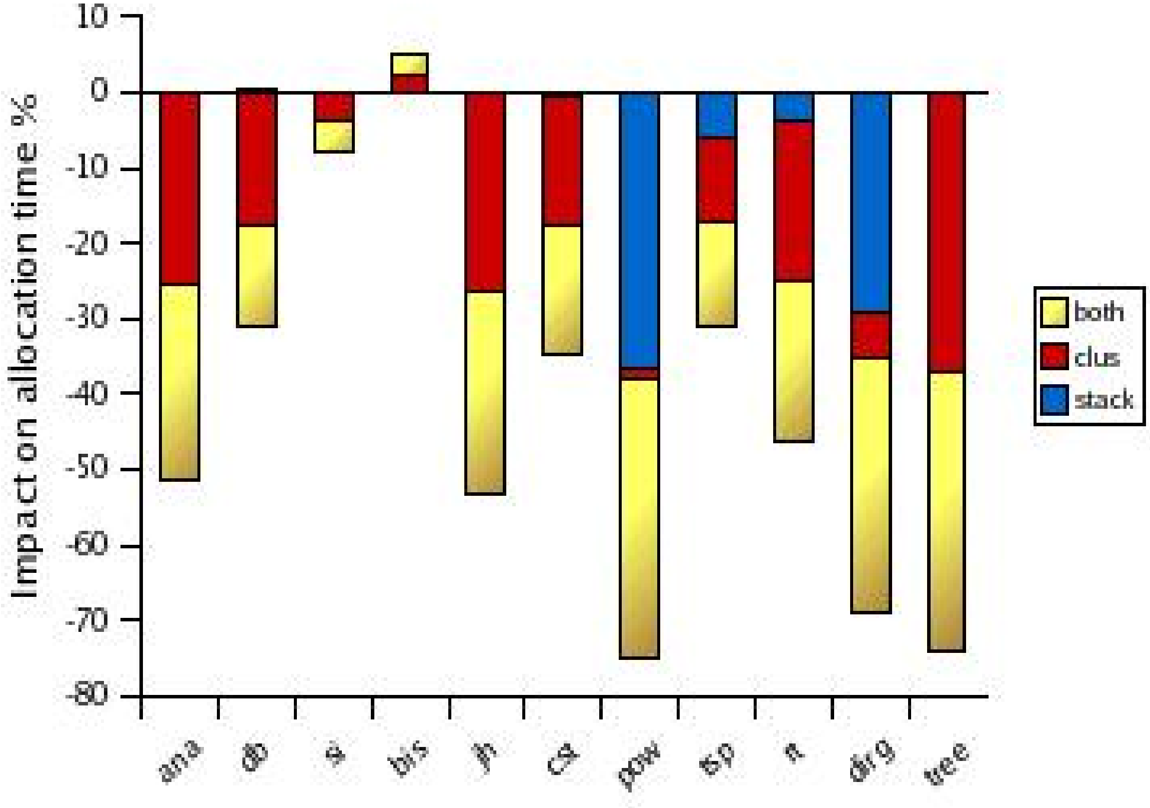
Average: 51.57%



Performance of Clustering: Inter-region References

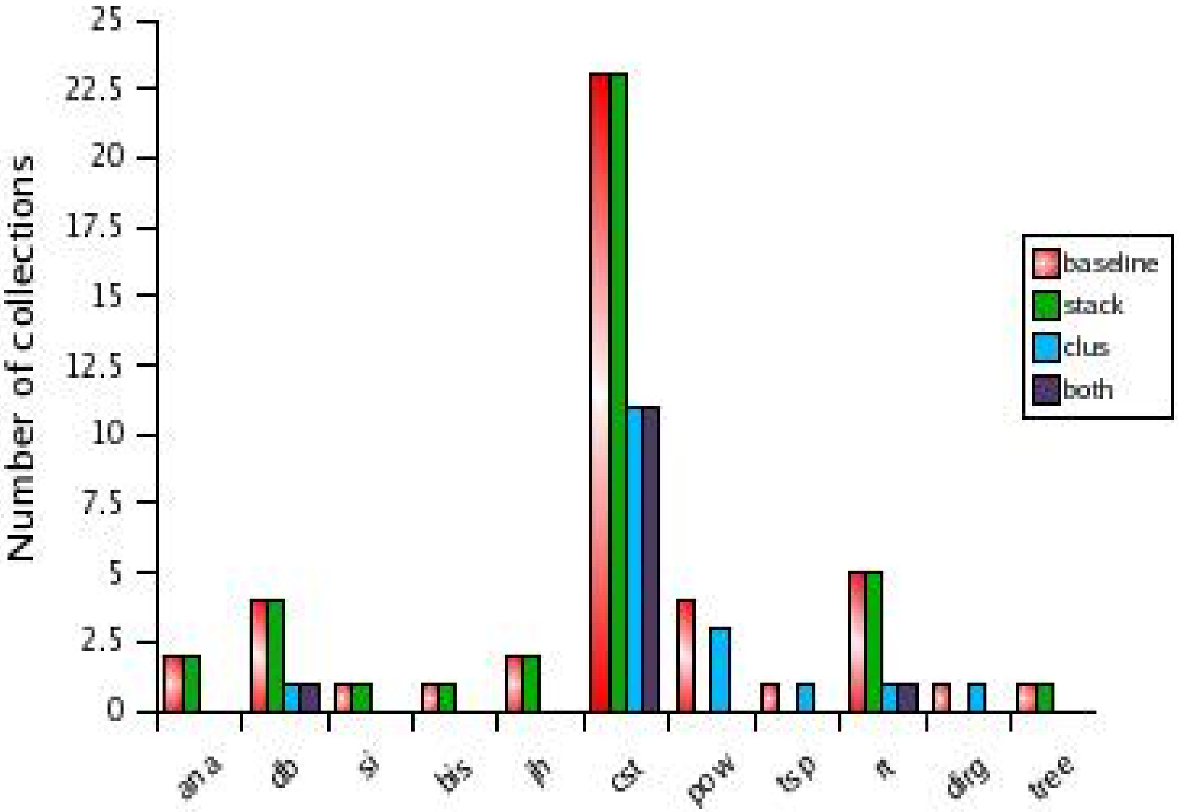
Program	Total No. of cluster to heap references	Total inter-region pointers without/with clustering	% Reduction in no. of inter-region pointers	% Garbage in cluster
_209_db	1	6916/14	99.79	11.2
_210_si	2	44731/39324	12.08	19.38
_208_cst	6	403912/169319	58.08	33.3
raytrace	3	163798/293	99.82	99.5

Performance of Clustering: Reduction in Allocation time



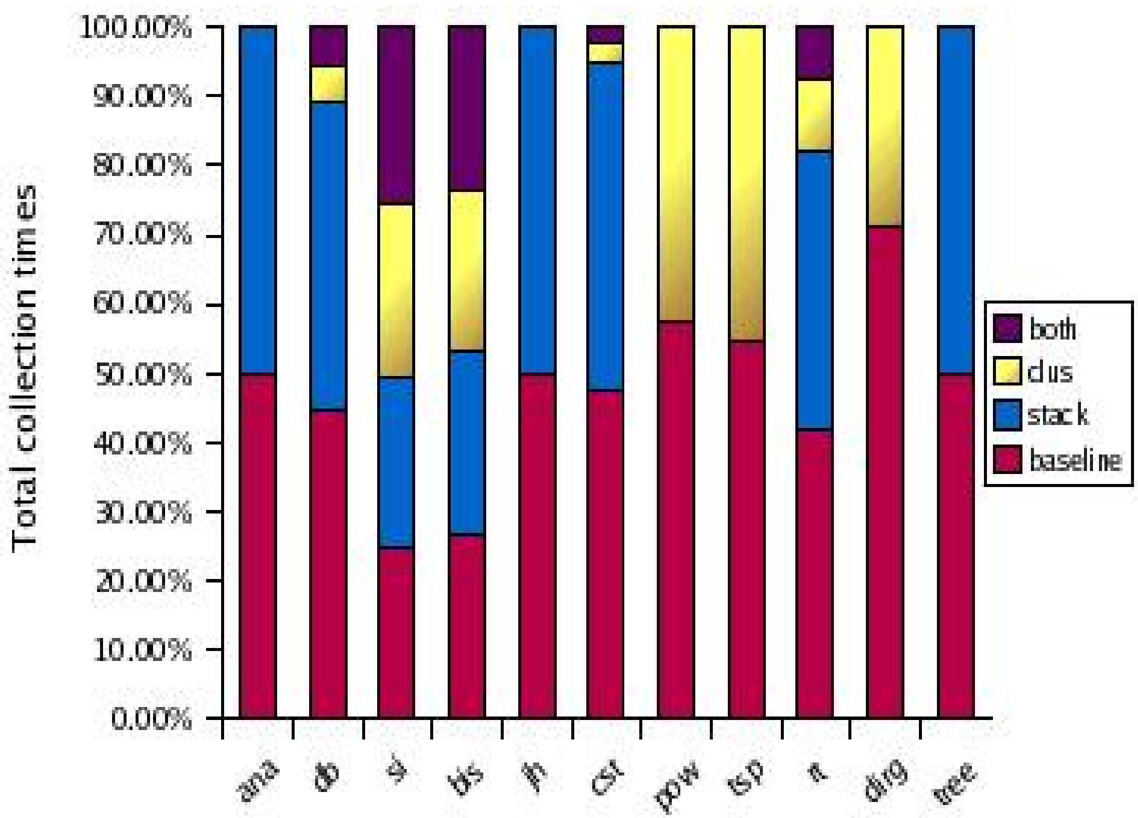
Technique	Average reduction
Stack allocation	12.61 %
Clustering	14.99 %
Both	20.63 %

Performance of Clustering: Impact on the No. of Collections



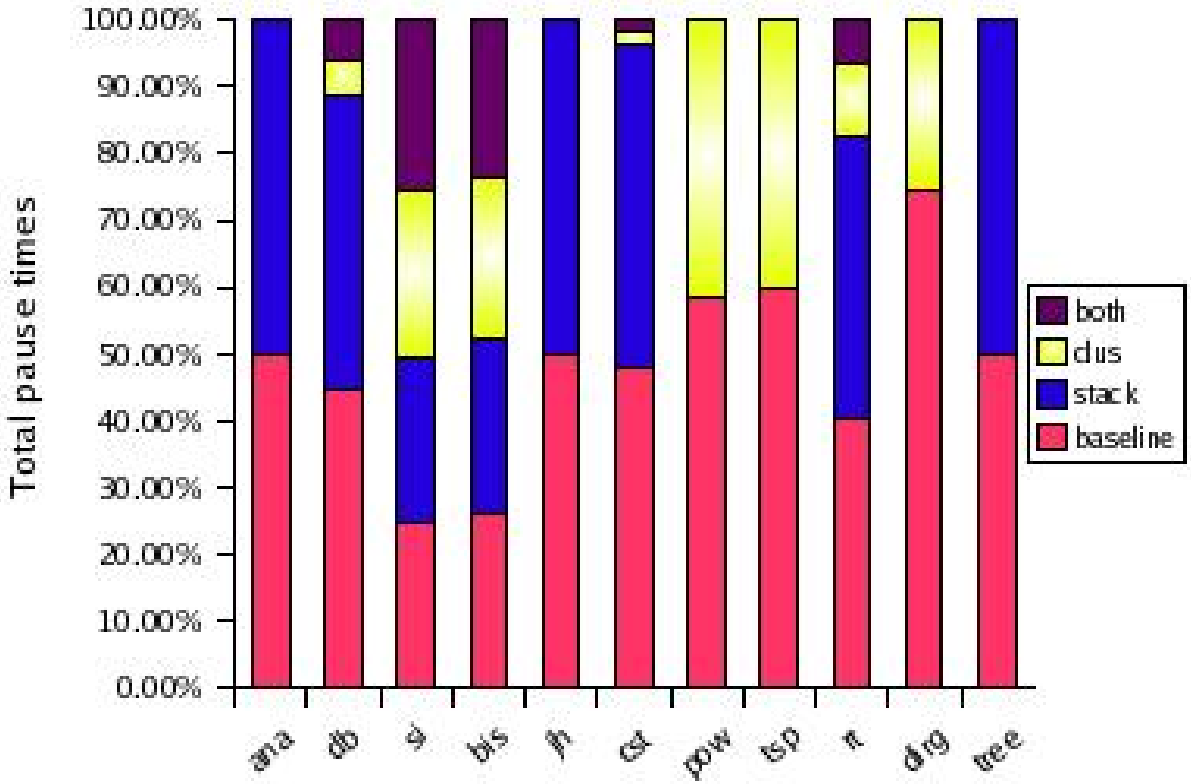
Technique	Average reduction
Stack allocation	75 %
Clustering	66.5 %
Both	91.56 %

Performance of Clustering: Reduction in collection time



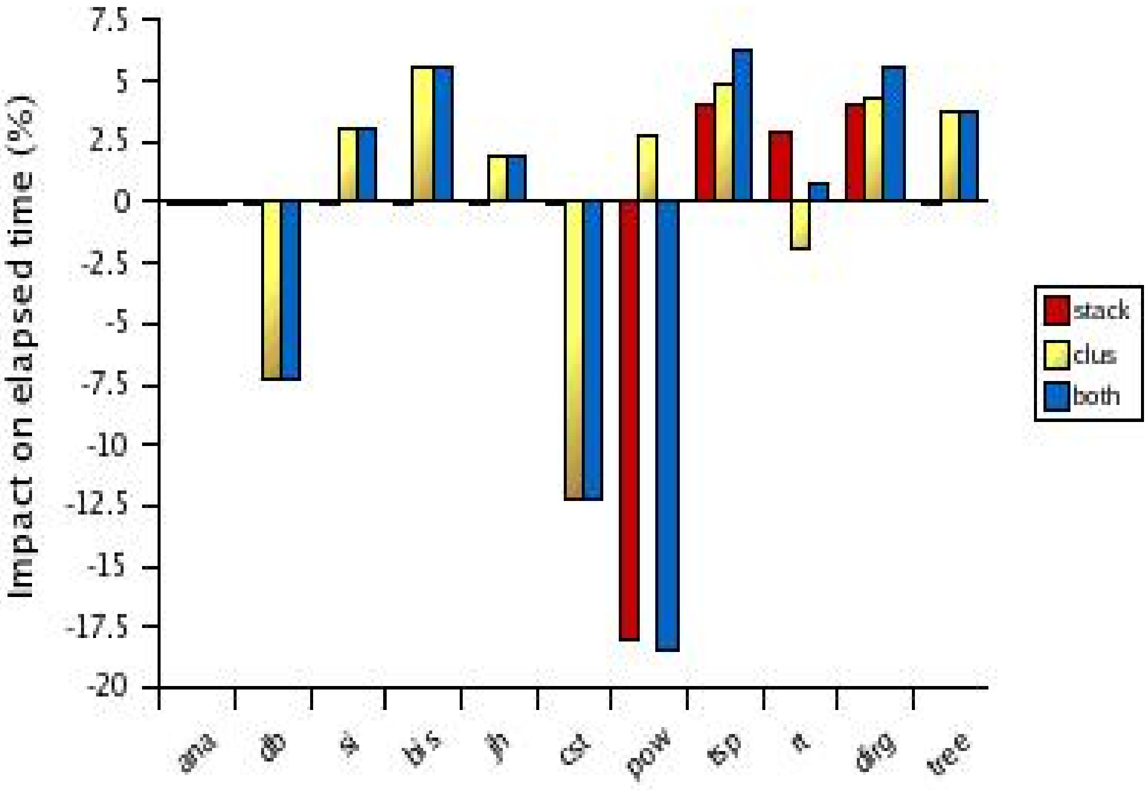
Technique	Average reduction
Stack allocation	60.9 %
Clustering	60.6 %
Both	79.27 %

Performance of Clustering: Reduction in Pause Time



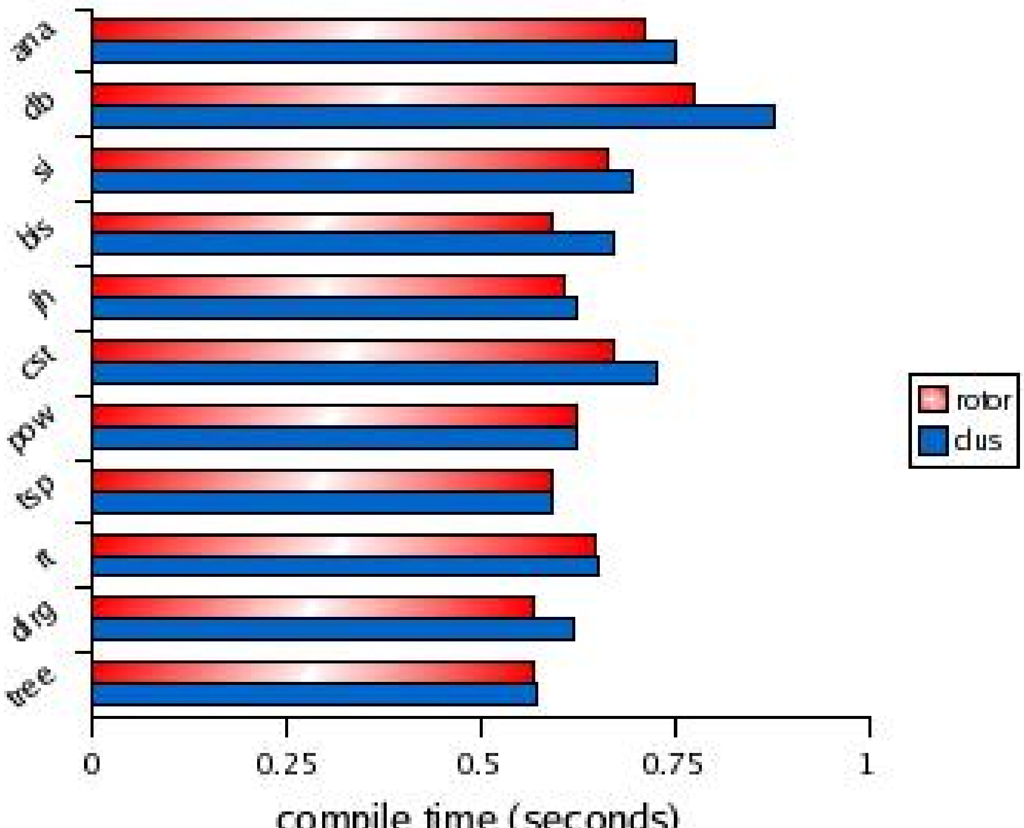
Technique	Average reduction
Stack allocation	63.55 %
Clustering	62.82 %
Both	79.33 %

Performance of Clustering: Impact on Elapsed Time



Technique	Average reduction
Stack allocation	1.75 %
Clustering	0.44 %
Both	1.018 %

Performance of Clustering: Impact on static compilation time



Average increase in the Static compilation time: 6.73%

Limitations of the Clustering Algorithm

- Analysis static and hence conservative
- Cannot handle dynamically growing data structures
- Cannot handle allocation site homogeneity

```
Void X ( .. )  {
 if(condn)
  a.f = Y();
 else
  b.f = Y();
}
```

```
Void Y() {
 return new classA();
}
```

Conclusions

- Clustering optimization
  - reduces the number of collections considerably
  - reduces individual collection times and pause times by a reasonable amount
  - reduces number of inter-region pointers
  - elapsed time is not affected much
  - reduces the total memory requirement
  - produces even better results, when applied along with stack allocation optimization

Thank You