

# Algorithms for Dynamic Memory Management (236780)

## Lecture 10

---

Lecturer: Erez Petrank



# Last Week

---

- Reference counting.

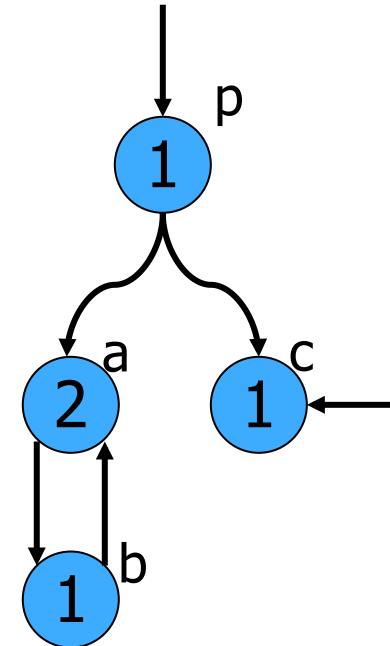


# Cycle Collection in Reference Counting Collectors

---

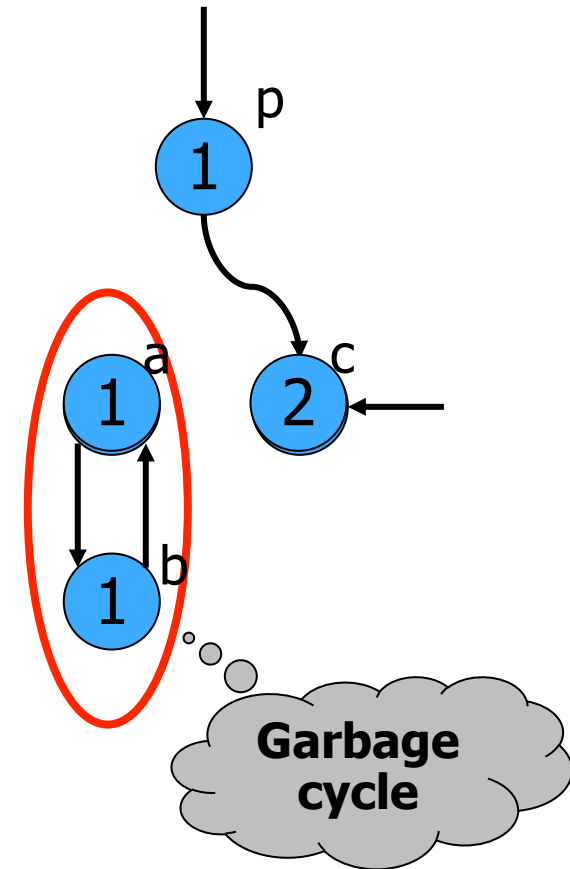
# Cyclic Structures Reclamation Problem

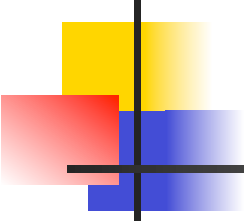
- Inability to reclaim cyclic data structures is a major drawback of reference counting (first noticed by McBeth 1963).
- A “**garbage cycle**” is an unreachable strongly connected component in the objects graph.



# Cyclic Structures Reclamation Problem

- Inability to reclaim cyclic data structures is a major drawback of reference counting (first noticed by McBeth 1963).
- A “**garbage cycle**” is an unreachable strongly connected component in the objects graph.





# Hybrid Memory Manager [Weizenbaum, 1969]

---

- The popular solution:
  - Use reference counting until heap exhausted.
  - Then, a tracing collector is invoked:
    - Reclaim all cyclic garbage,
    - Updates the reference count of live objects.
- ⇒ Benefits:
  - ⇒ Circular structures can be reclaimed.
  - ⇒ Tracing allows small rc fields to be used, and it restores stuck values when run.



# Cycle Collection

---

- **We'll go briefly through:**
  - Reference count garbage collection. Christopher [1984].
  - Cyclic reference counting with local mark-scan. Martinez, Wachenchauzer and Lins [1990].
  - Cyclic reference counting with lazy mark-scan. Lins [1992].
- **And concentrate on:**
  - **Stop-the-world version:** Concurrent cycle collection in reference counted systems. Bacon and Rajan [2001].
  - **Concurrent version:** Efficient on-the-fly cycle collection. Paz, Bacon, Kolodner, Petrank and Rajan [2005].



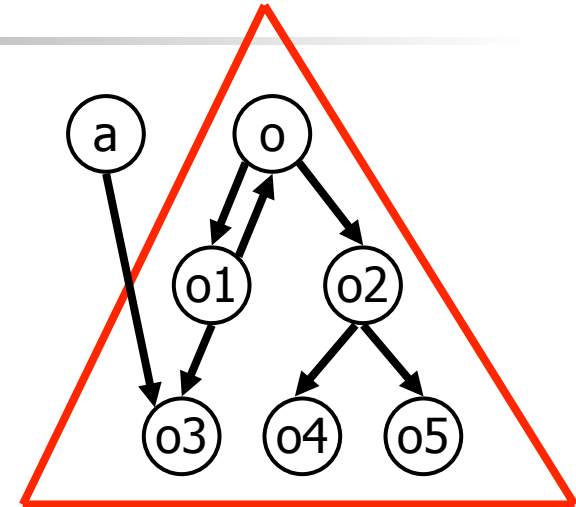
# Cycle Collection Basic Idea - 1

---

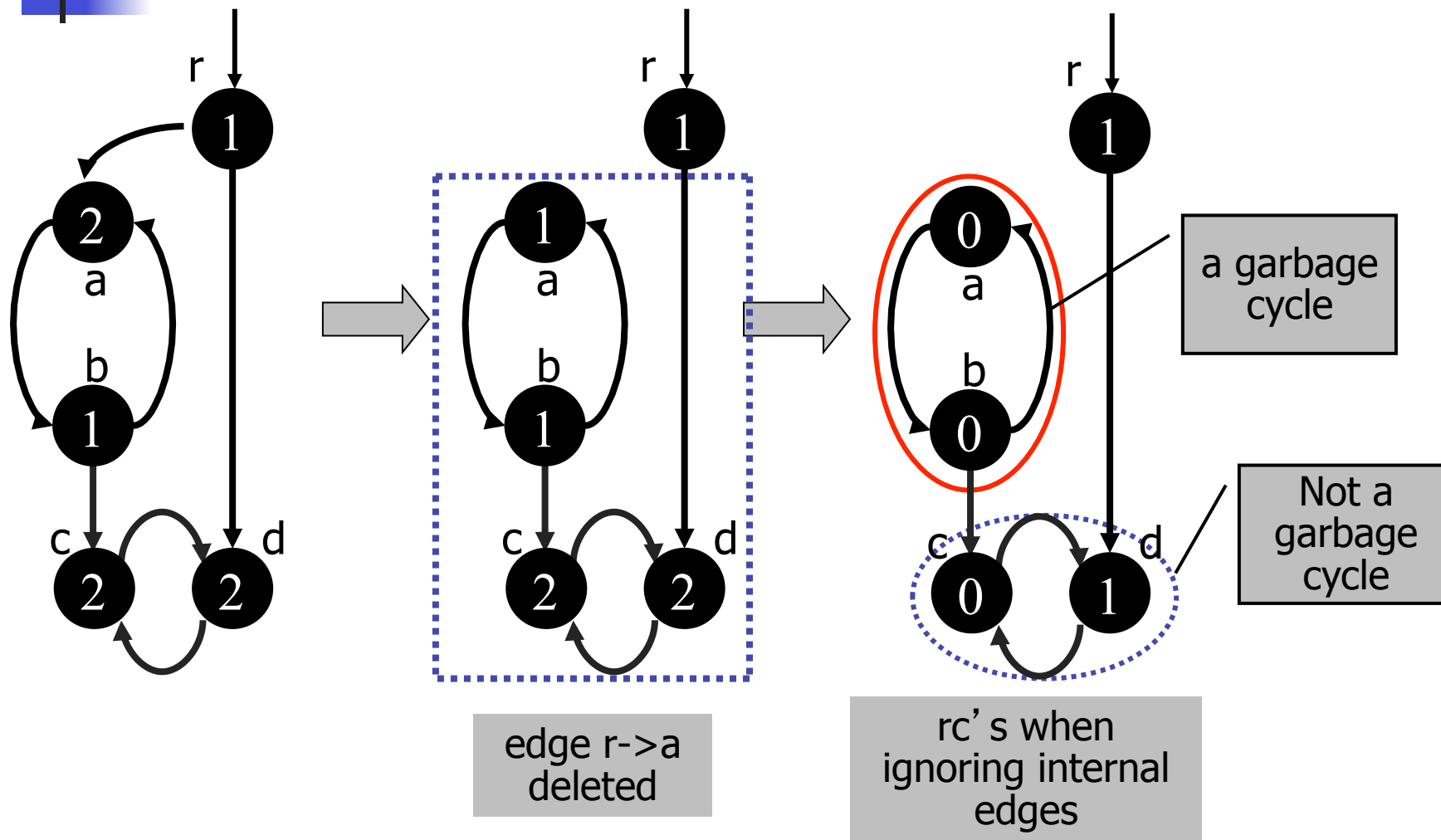
- Observation 1: Garbage cycles can only be created when an rc is decremented to a non-zero value.
- ⇒ Objects whose rc is decremented to a non-zero value become **candidates**.

# Cycle Collection Basic Idea - 2

- Terms:
  - **Sub-graph** of  $O$ : graph of objects reachable from  $O$ .
  - **External pointer (to a sub-graph)**: a pointer from a non sub-graph object to a sub-graph object.
  - **Internal pointer (of a sub-graph)**: a pointer between 2 sub-graph objects.
- **Observation 2**: In a garbage cycle all the reference counts are due to internal pointers of the cycle.
- ⇒ For each candidate's sub-graph, check if all reference counts are due to internal pointers.



# Goal: Compute Counts for External Pointers Only





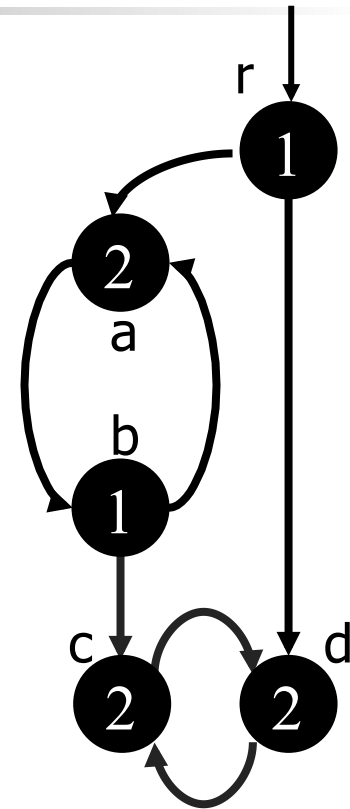
# The Original Algorithm

---

- ⇒ Whenever an rc of an object  $O$  is decremented to a non-zero value, perform 3 **local** traversals over the sub-graph of  $O$ .
  - Update the rc's to reflect only pointers that are external to the graph of  $O$ .
  - Restore the rc of the externally reachable objects.
  - Reclaims all objects which are not externally reachable.

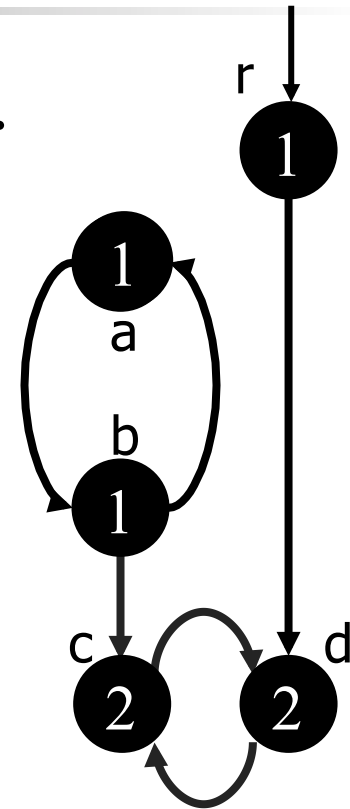
# Implementation: an Example

- Use three colors: black, gray, white.
- ⇒ Whenever an rc of an object 'a' is decremented to a non-zero value, perform 3 **local** traversals over the sub-graph of 'a'.



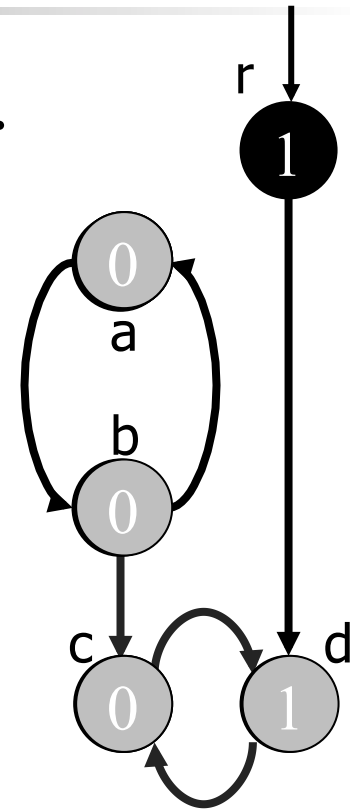
# Implementation: an Example

- Use three colors: black, gray, white.
- ⇒ Whenever an rc of an object 'a' is decremented to a non-zero value, perform 3 **local** traversals over the sub-graph of 'a' .
  - **Mark:** Updates rc's to reflect only pointers that are external to the graph of 'a', marking nodes in gray.



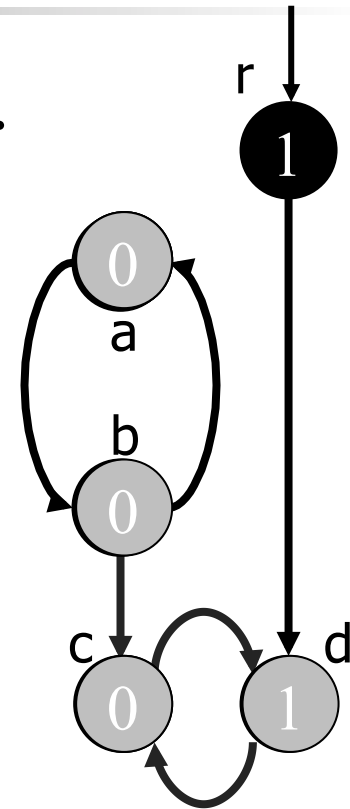
# Implementation: an Example

- Use three colors: black, gray, white.
- ⇒ Whenever an rc of an object 'a' is decremented to a non-zero value, perform 3 **local** traversals over the sub-graph of 'a'.
- **Mark:** Updates rc's to reflect only pointers that are external to the graph of 'a', marking nodes in gray.



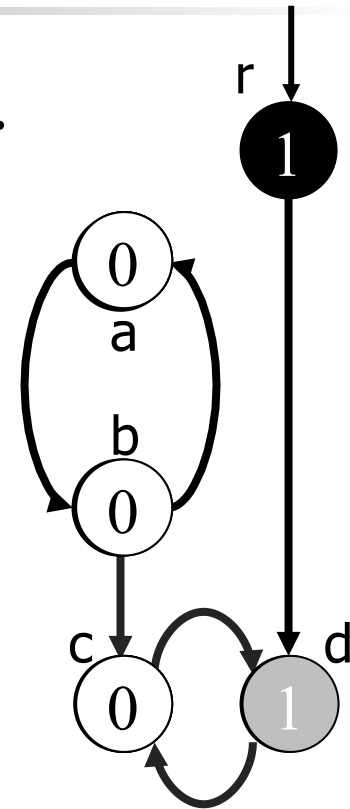
# Implementation: an Example

- Use three colors: black, gray, white.
- ⇒ Whenever an rc of an object 'a' is decremented to a non-zero value, perform 3 **local** traversals over the sub-graph of 'a'.
  - **Mark:** Updates rc's to reflect only pointers that are external to the graph of 'a', marking nodes in gray.
  - **Scan:** Restores rc's of the externally reachable objects, coloring them in black. Rest of the nodes are marked as garbage (white).



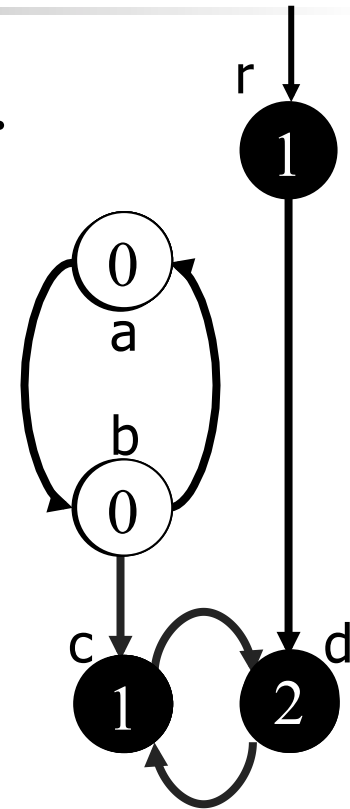
# Implementation: an Example

- Use three colors: black, gray, white.
- ⇒ Whenever an rc of an object 'a' is decremented to a non-zero value, perform 3 **local** traversals over the sub-graph of 'a'.
  - **Mark:** Updates rc's to reflect only pointers that are external to the graph of 'a', marking nodes in gray.
  - **Scan:** Restores rc's of the externally reachable objects, coloring them in black. Rest of the nodes are marked as garbage (white).



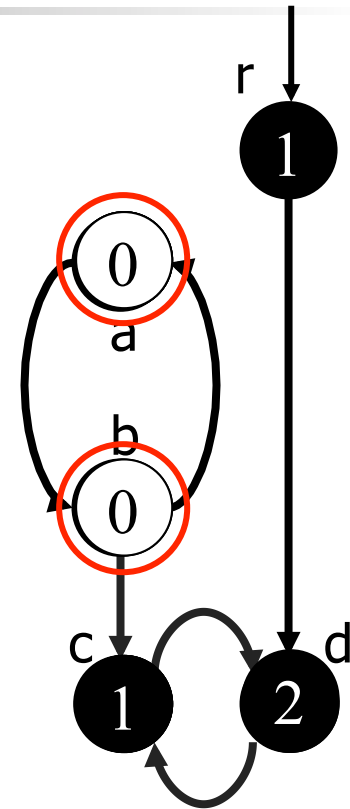
# Implementation: an Example

- Use three colors: black, gray, white.
- ⇒ Whenever an rc of an object 'a' is decremented to a non-zero value, perform 3 **local** traversals over the sub-graph of 'a'.
  - **Mark:** Updates rc's to reflect only pointers that are external to the graph of 'a', marking nodes in gray.
  - **Scan:** Restores rc's of the externally reachable objects, coloring them in black. Rest of the nodes are marked as garbage (white).



# Implementation: an Example

- Use three colors: black, gray, white.
- ⇒ Whenever an rc of an object 'a' is decremented to a non-zero value, perform 3 **local** traversals over the sub-graph of 'a'.
  - **Mark:** Updates rc's to reflect only pointers that are external to the graph of 'a', marking nodes in gray.
  - **Scan:** Restores rc's of the externally reachable objects, coloring them in black. Rest of the nodes are marked as garbage (white).
  - **Collect:** collects garbage objects (the white ones).





# Pseudo Code

## Delete(S)

```
rc(S):=rc(S)-1
if (rc(S)=0)
  for T in Sons(S)
    Delete(T)
  link S to free-list
else
  Mark(S)
  Scan(S)
  Collect_White(S)
```

Traces S' sub-graph,  
removing rc due to  
pointers internal to  
the sub-graph.

## Mark(S)

```
if (color(S)=black)
  color(S):=gray
  for T in Sons(S)
    rc(T):=rc(T)-1
  Mark(T)
```

# Pseudo Code (cont' )

At this point, finding a non-zero rc in the gray sub-graph of S, means an external reference.

## Scan(S)

```
if (color(S)=gray)
  if (rc(S)>0)
    Scan_Black(S)
  else
    color(S):=white
    for T in Sons(S)
      Scan(T)
```

## Scan\_Black(S)

```
color(S):=black
for T in Sons(S)
  rc(T):=rc(T)+1
  if (color(T)≠black)
    Scan_Black(T)
```

At this point, the sub-graph of S contains only black & white objects.

## Collect\_White(S)

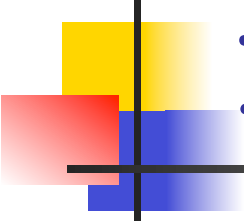
```
if (color(S)=white)
  color(S):=black
  for T in Sons(S)
    Collect_White(T)
  link S to free_list
```



# Correctness

---

- After phase 1: a non-zero rc in the gray sub-graph, is due to external references.
- In phase 2: objects with no external references become white, until an object  $A$  with an external reference is found. Then,  $A$ 's sub-graph is colored black and its rc's are restored.
- If after phase 2 there are still white objects in  $S'$  sub-graph, they must be unreachable, and are reclaimed.



# It's Good to be Lazy

---

- Most pointer deletions (to a non-zero rc) do not really indicate a garbage cyclic structure.
  - It is a waste to check all these candidates.
- Idea: lazy candidate check.
  - Keep a list of all candidates,
  - Periodically, check recorded candidates that still look good candidates.
  - Some of the candidates just reach a zero rc and are reclaimed with no cycle collection,
  - Other candidates' rc gets incremented, testifying for their liveness.
  - Very effective at eliminating repeated traversals.



# The Lazy Algorithm - Implementation

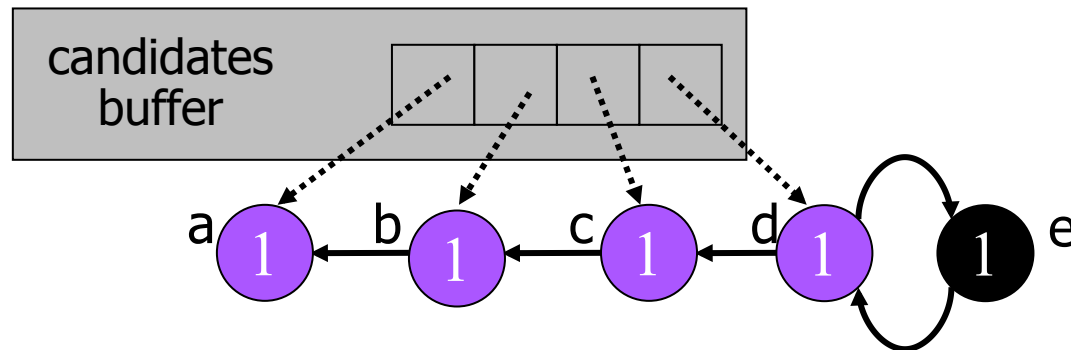
---

- Use purple to avoid candidate duplications.
- If  $rc(O)$  is decremented:
  - If  $rc(O) == 0$  then  $O$  is colored black and recycled (even if buffered).
  - Else, if  $O$  is not purple, then color  $O$  purple & buffer  $O$ .
- If  $rc(O)$  is incremented,  $O$  is colored black.
- Cycle collection is run only on purple candidate in the buffer.

The Lazy algorithm substantially decreases the number of traversed candidates.

# Tracing Complexity

- At worst case, the algorithm is quadratic in the size of the graph.



- ⇒ Solution: each phase is run on all candidates simultaneously (and not on each candidate separately).
- Algorithm is linear in the graph's size.



# Acyclic Objects

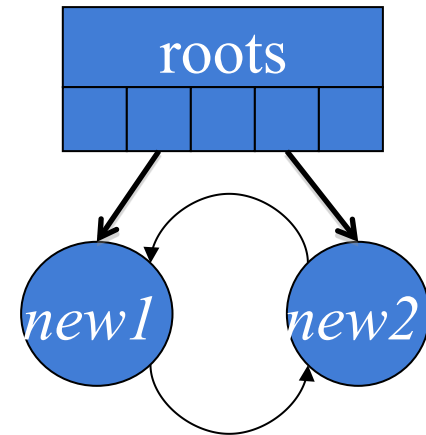
---

- Some objects are inherently acyclic (scalars, strings, arrays of the previous...).
- ⇒ The cycle collection ignores acyclic objects.
- Significantly reduces the overhead of cycle collection.
  - Usually reduces the objects considered as roots of cycles by an order of magnitude.

# New Objects Problem

A garbage cycle can be formed by a reference deletion from the roots.

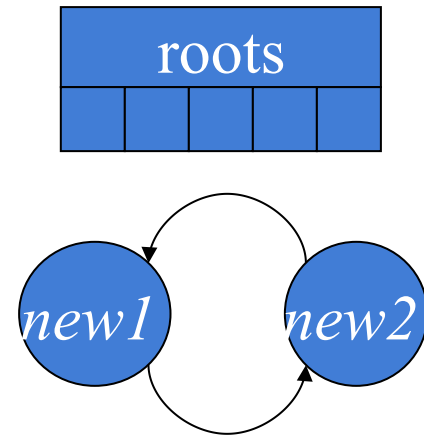
- Two objects are created,
- Their pointers form a cycle,
- Their local pointers are erased.



# New Objects Problem

A garbage cycle can be formed by a reference deletion from the roots.

- Two objects are created,
- Their pointers form a cycle,
- Their local pointers are erased.
- Solution 1: Write-barrier on roots (costly).
- Solution 2: The following objects are candidates:
  - All objects created since the last collection become candidates.
  - Root referents of previous collection.

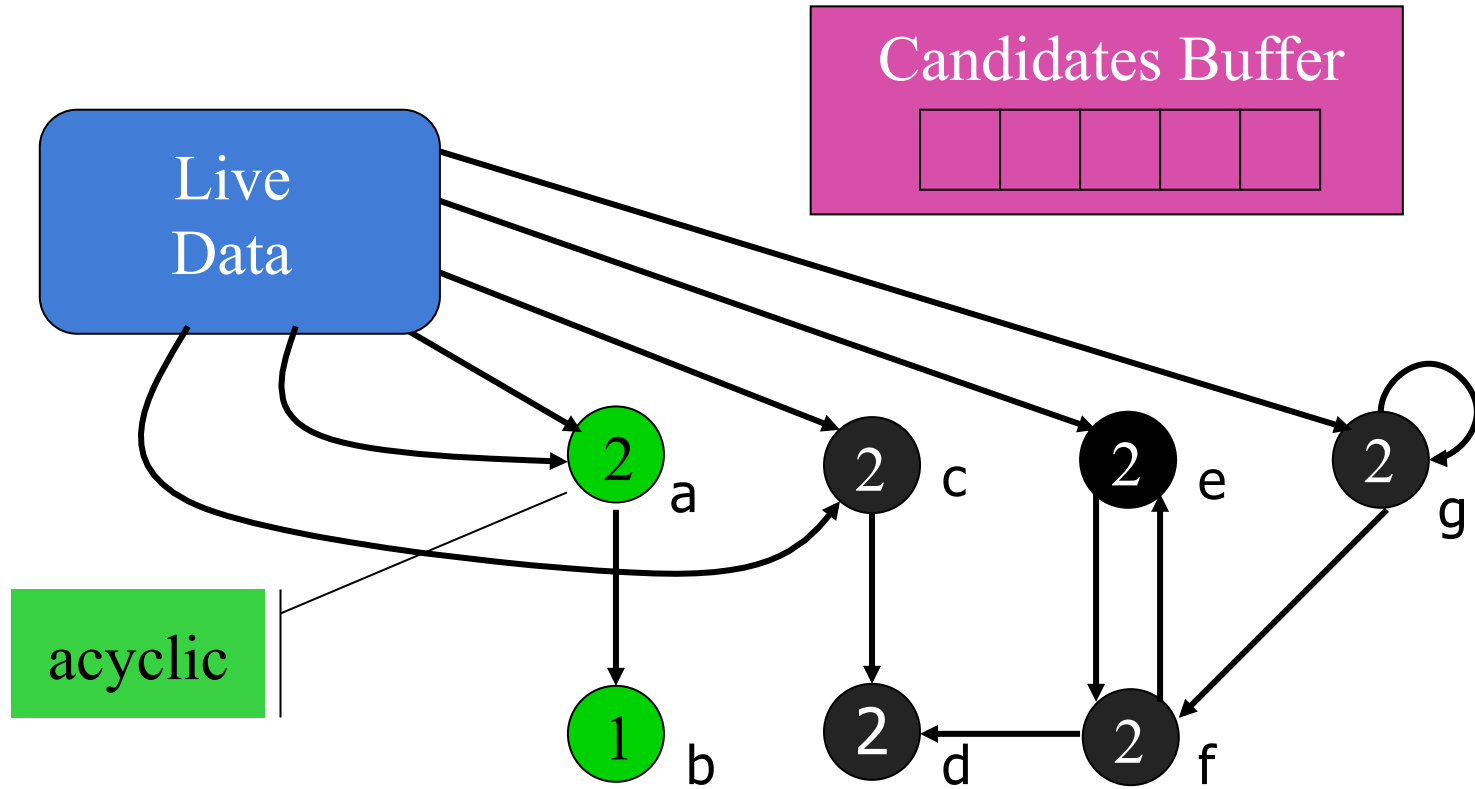




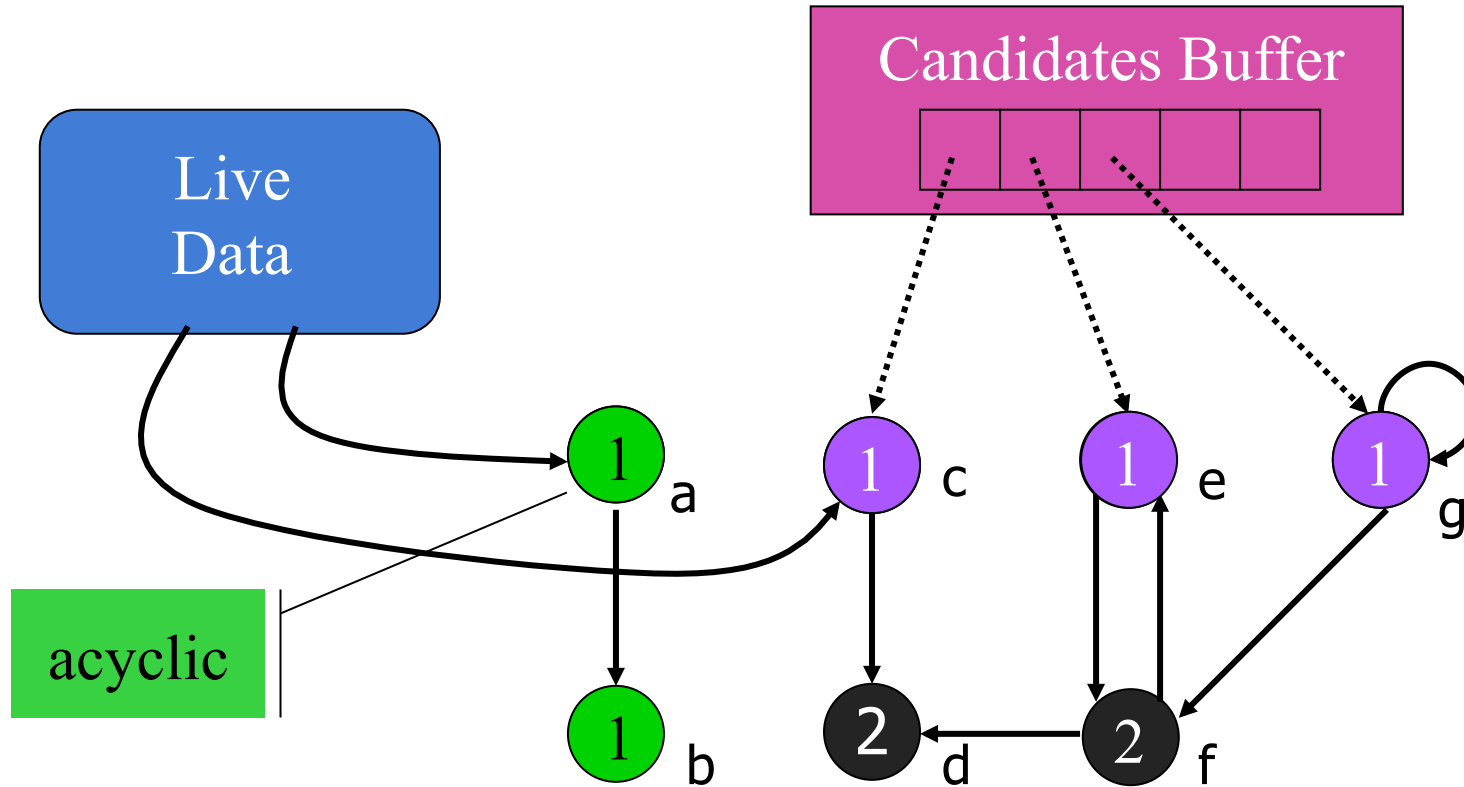
# Correctness of Solution 2

---

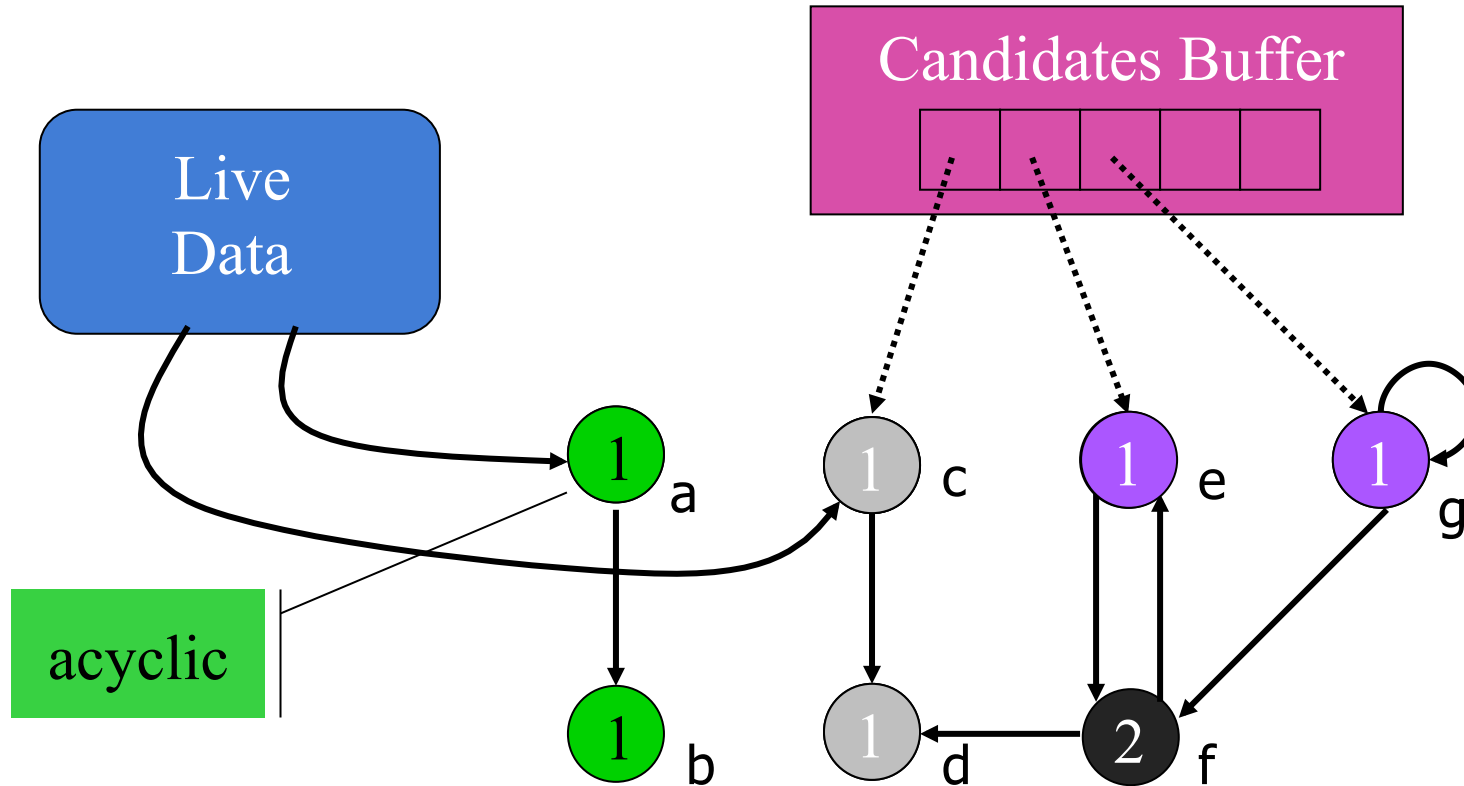
- The following objects are candidates:
  - All objects created since the last collection.
  - Root referents of previous collection.
- Case 1: at least one **new** object in cycle.
  - We have it as a candidate.
- Case 2: all objects are old in cycle.
- Case 2a: at least one of the objects was **referenced by roots** in previous collection.
  - We have it as a candidate.
- Case 2b: no root reference in previous collection, thus at least one of the objects was **referenced from heap**.
  - Added to candidates when heap reference deleted.



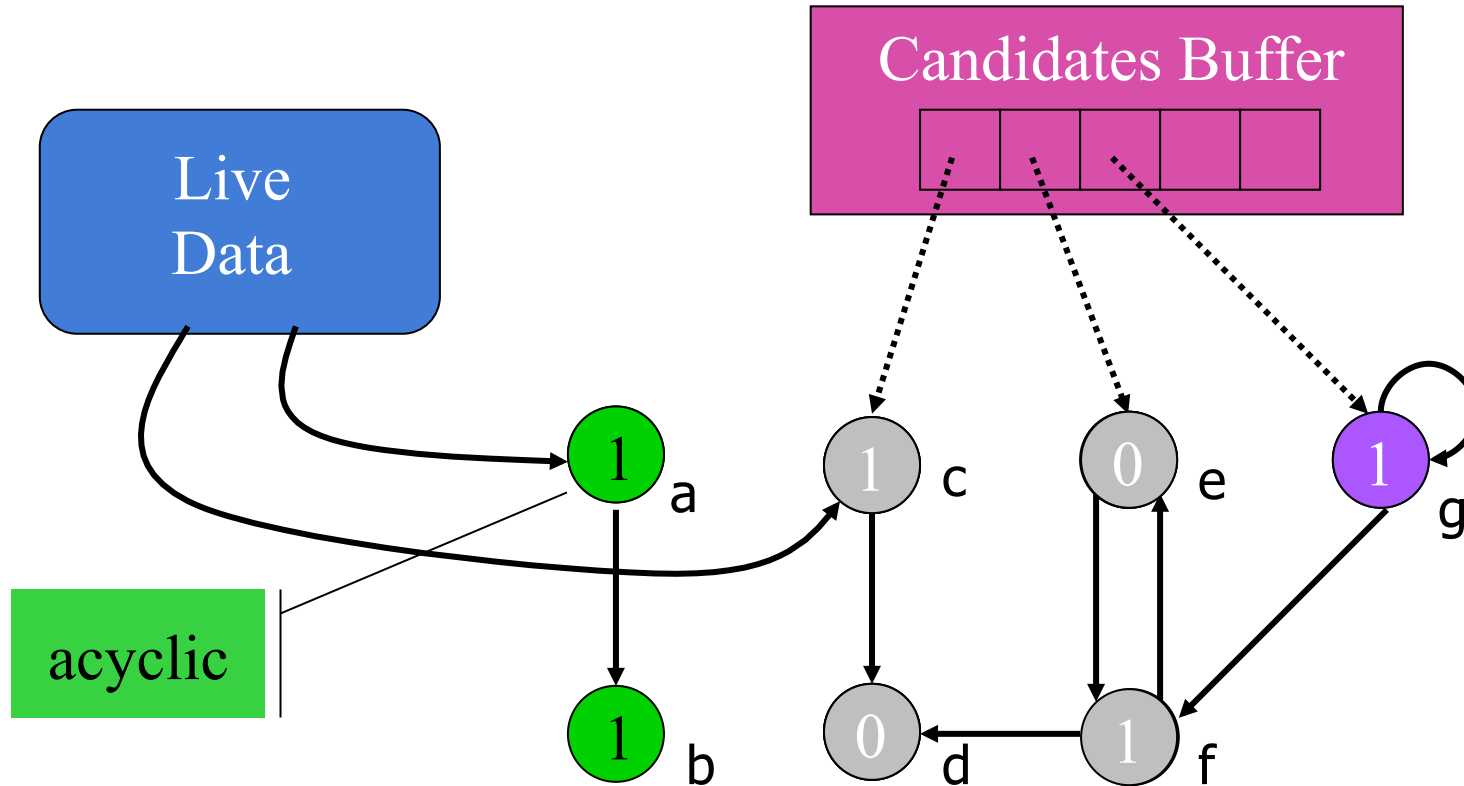
Taken from a presentation of David  
F. Bacon



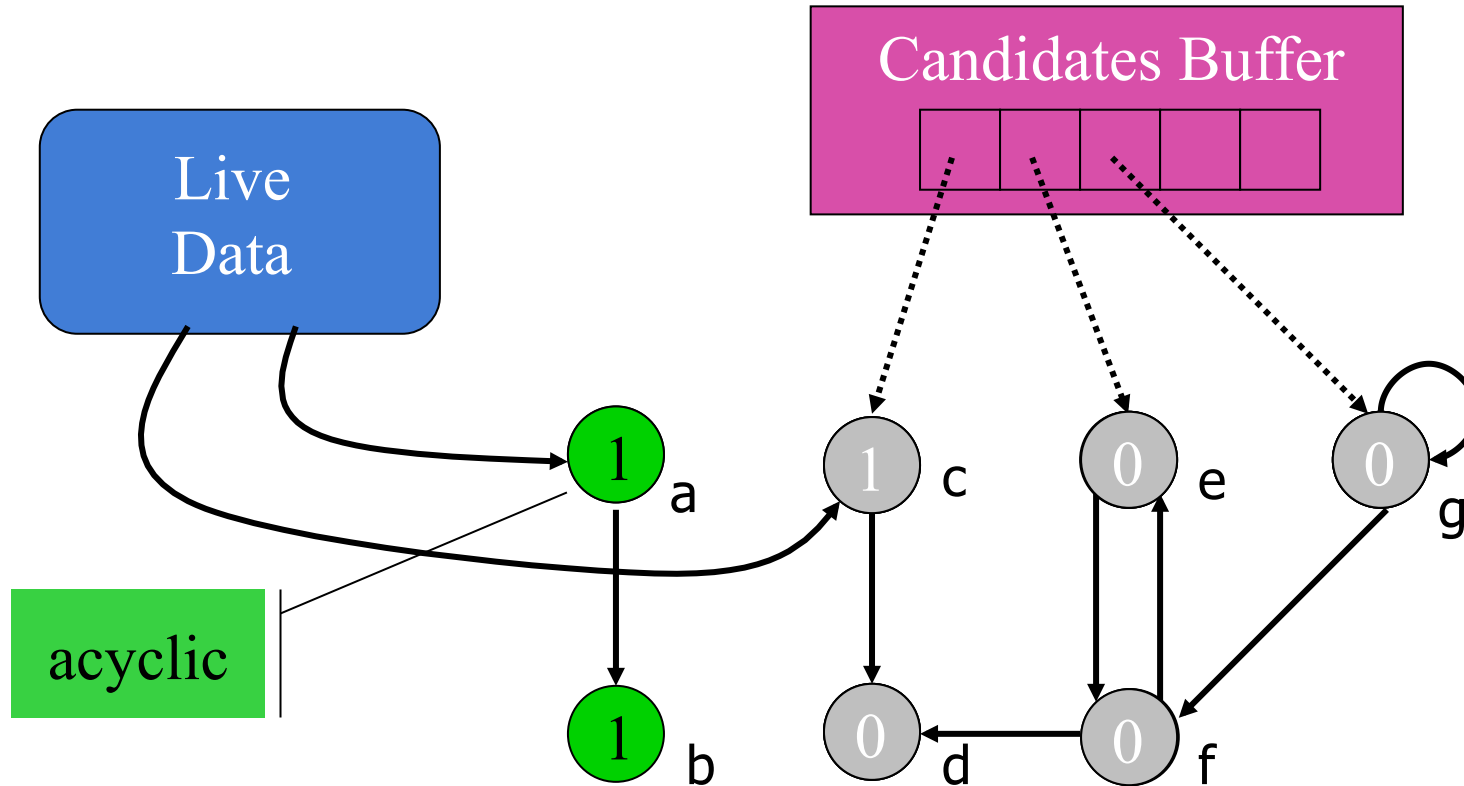
1. Process Decrements and Accumulate Candidates.



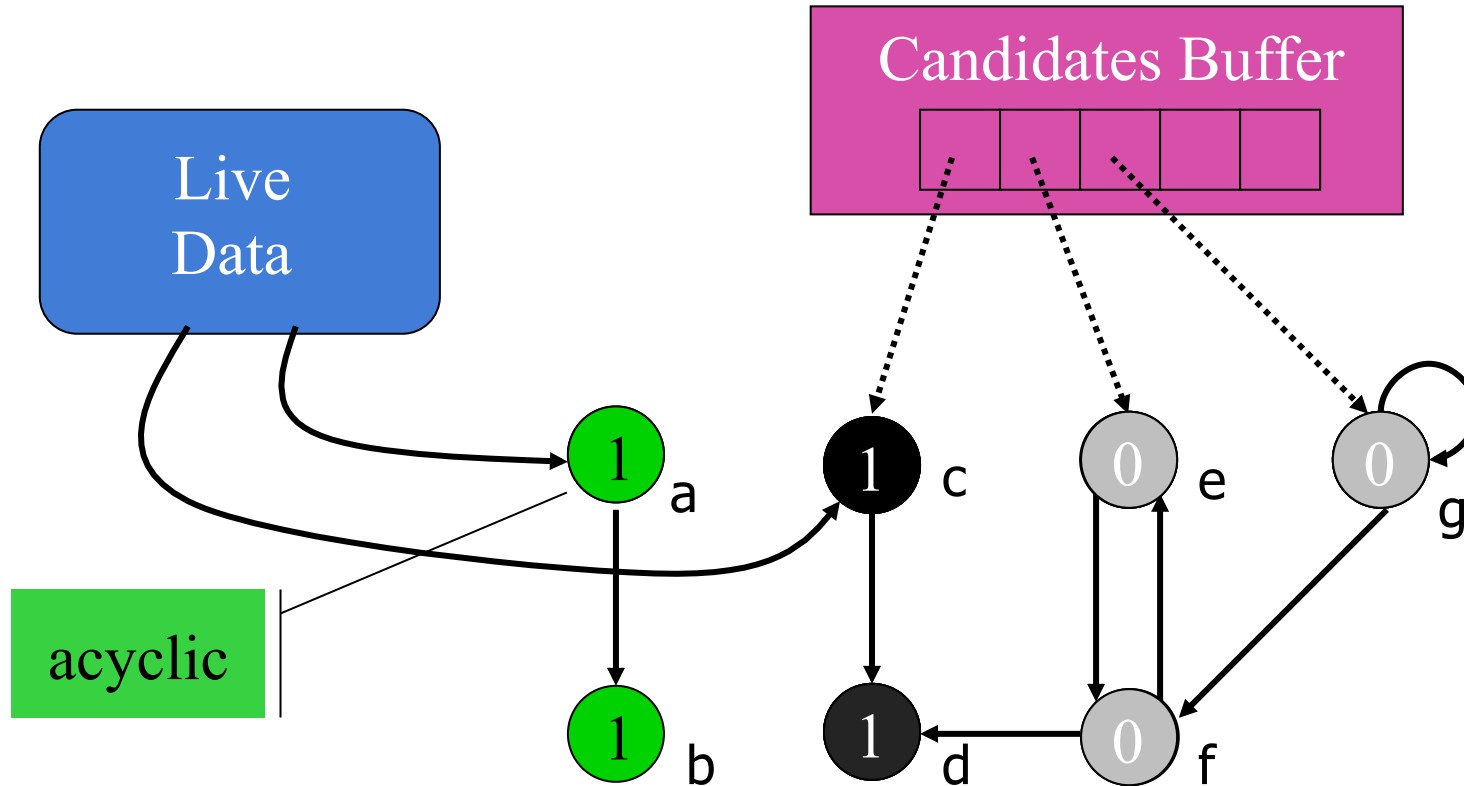
1. Process Decrements and Accumulate Candidates.
2. Mark Gray: Subtract Internal Reference Counts.



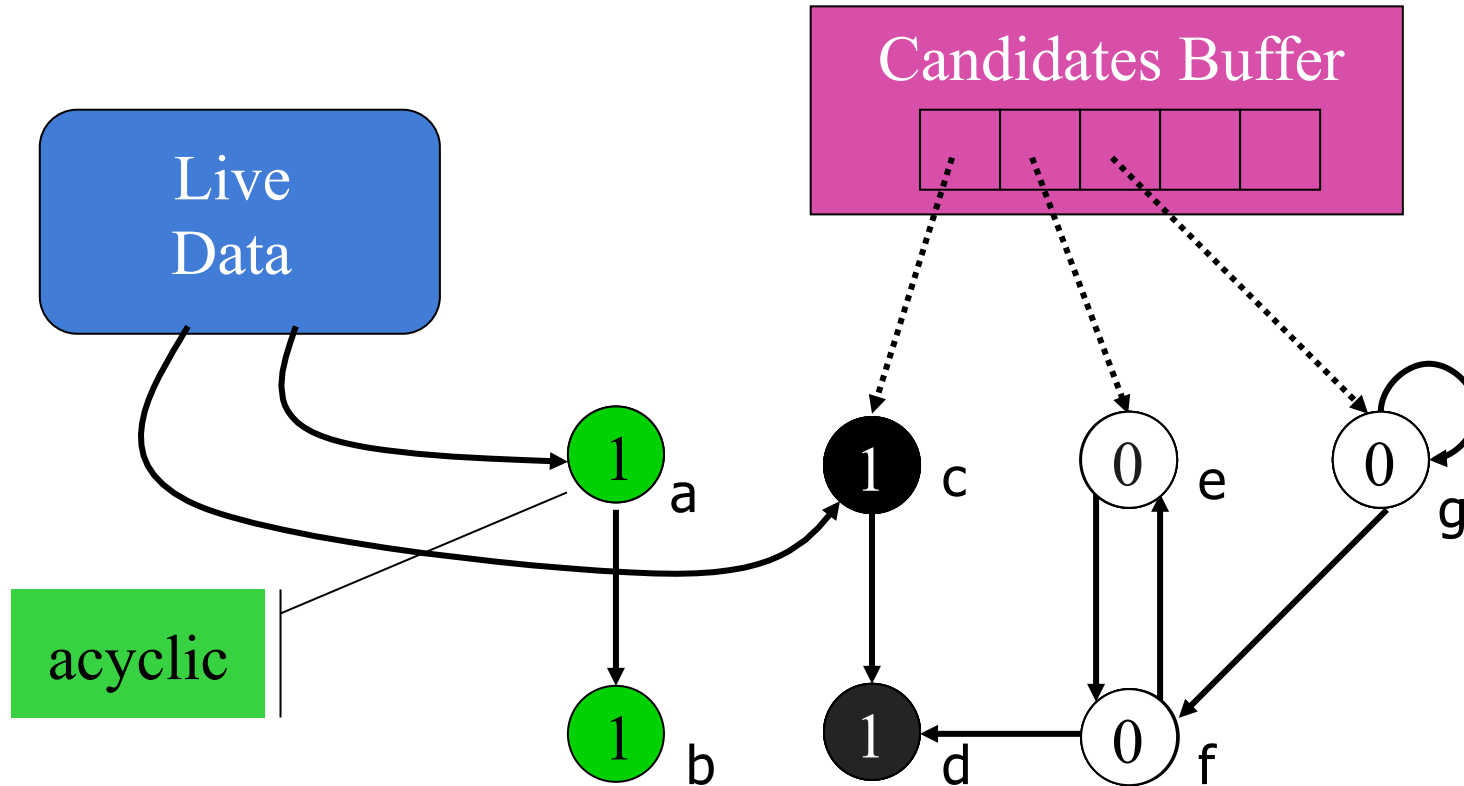
1. Process Decrements and Accumulate Candidates.
2. Mark Gray: Subtract Internal Reference Counts.



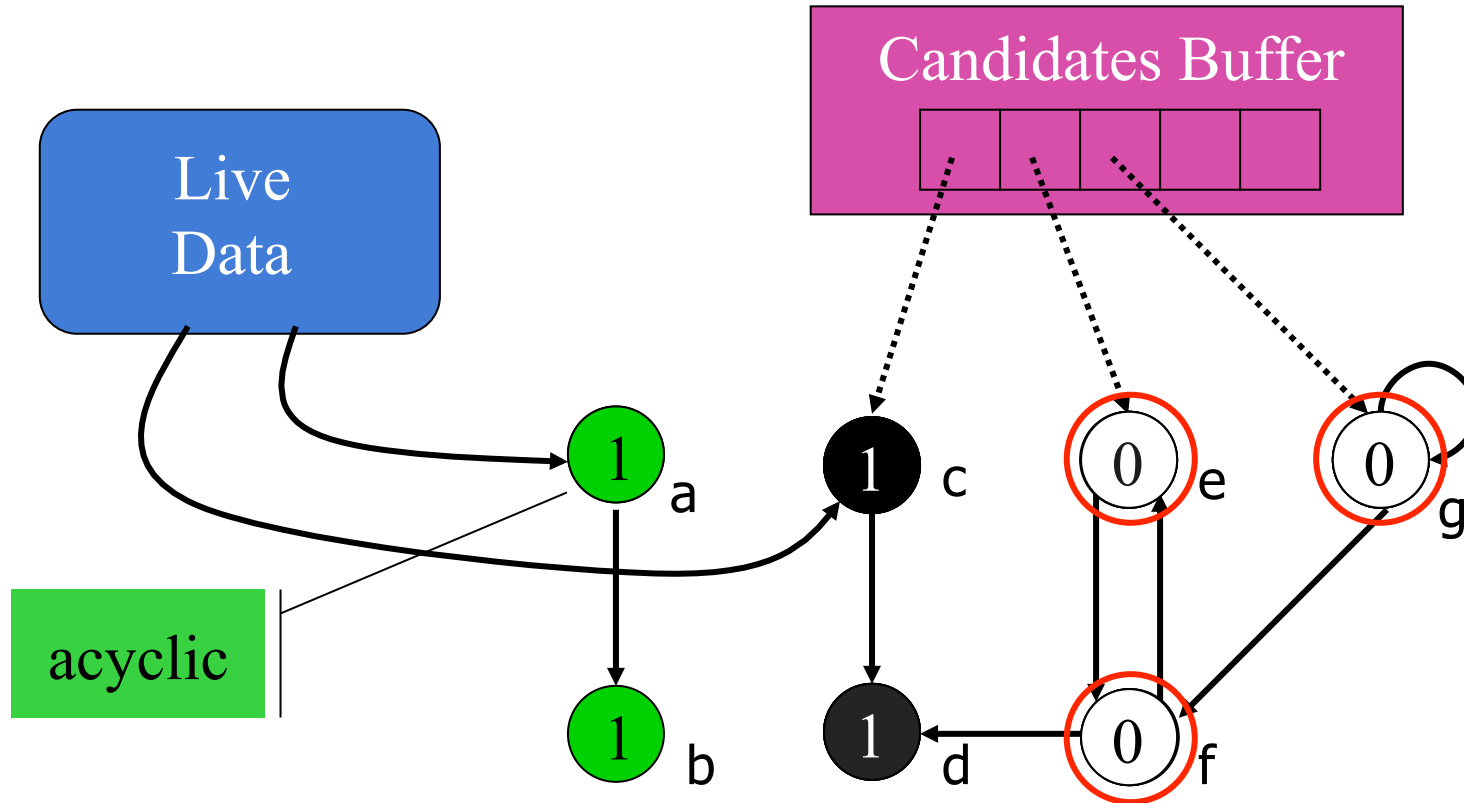
1. Process Decrements and Accumulate Candidates.
2. Mark Gray: Subtract Internal Reference Counts.



1. Process Decrements and Accumulate Candidates.
2. Mark Gray: Subtract Internal Reference Counts.
3. Scan: Restore Live, Mark Dead White.



1. Process Decrements and Accumulate Candidates.
2. Mark Gray: Subtract Internal Reference Counts.
3. Scan: Restore Live, Mark Dead White.



1. Process Decrements and Accumulate Candidates.
2. Mark Gray: Subtract Internal Reference Counts.
3. Scan: Restore Live, Mark Dead White.
4. Collect White.

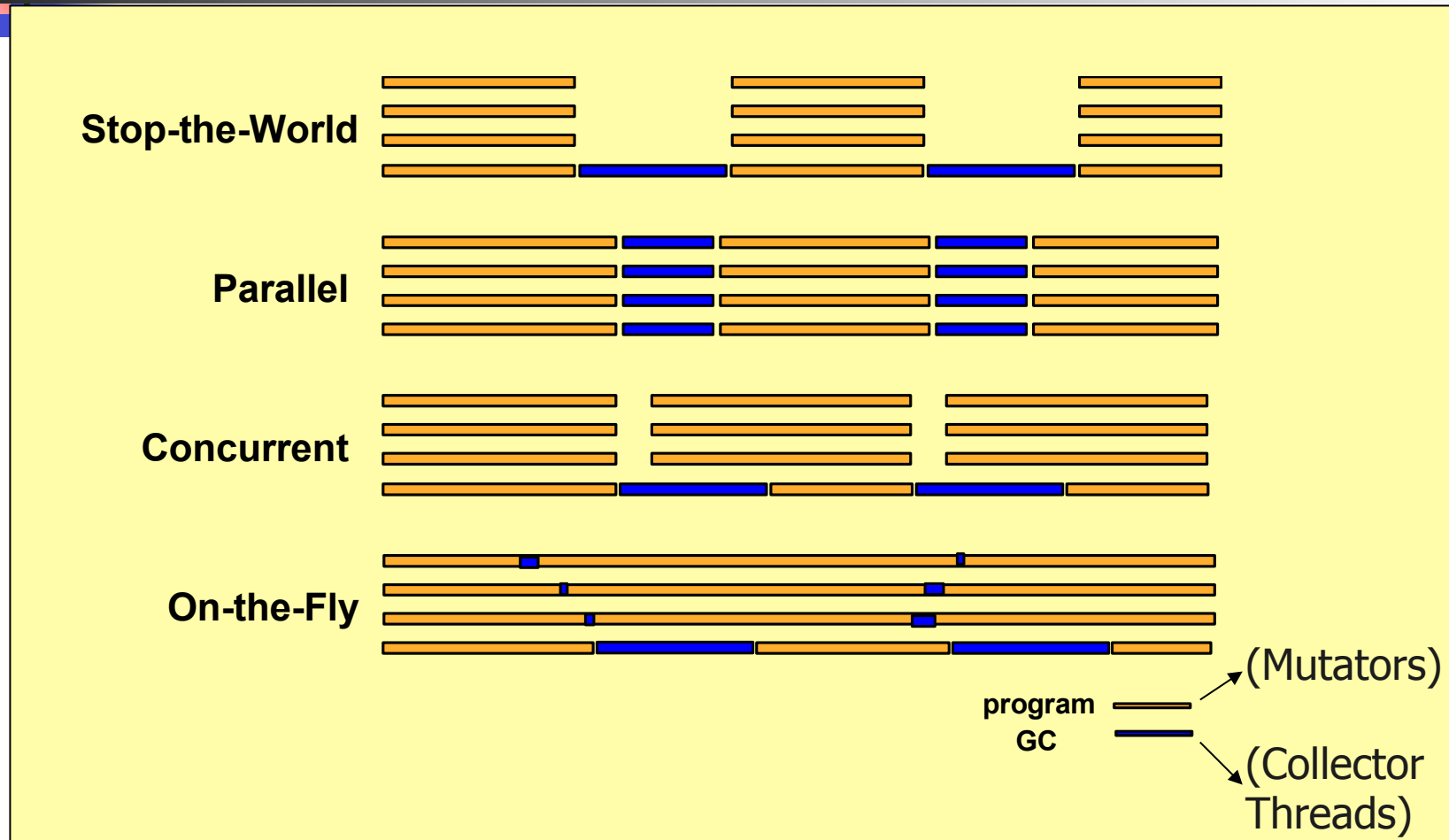
Taken from a presentation of David  
F. Bacon



# Concurrent Cycle Collection

---

# Terminology





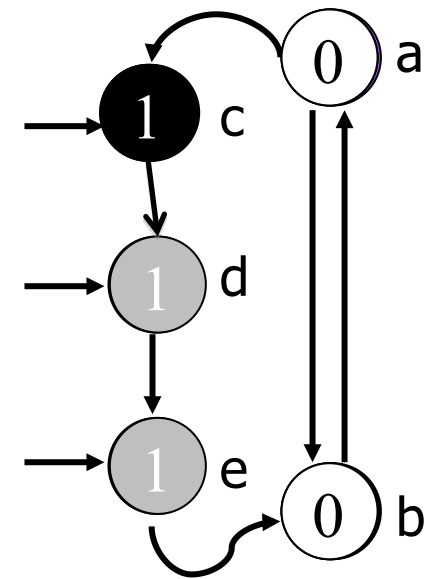
# Moving to concurrency

---

- If we just take the above cycle collector and use concurrently, we may foil correctness.
- Examples of problems:
- Object graph is modified while collector scans it.
  - Cannot rely on repeated traversals to read the same set of nodes and edges.
- Counts may be out of date due to mutator activity.
- Safety problem: the cycle collection algorithm may reclaim live nodes as a garbage cycle.

# Safety Problem - Example

- The mutator deletes an edge, causing the *Scan* phase to incorrectly infer a live object to be garbage.
- The edge *c*->*d* is cut between the *MarkGray* and *Scan* procedures.
- If no precaution is taken:
  - *a* & *b* determined to be garbage.
  - RC of all objects is not correct.
  - *d* & *e* left gray.





# First On-the-Fly Cycle Collector

---

- [Bacon & Rajan 2001]
- A relatively complicated solution, attempting to solve the inconsistent object graph view.
- Main ideas:
  - Repeat scanning and use more colors to ensure objects are not modified during the traversals.
  - Identified cycles are not reclaimed, but only recorded.
  - The next collection runs further tests to ensure that recorded cycle is indeed garbage, and then reclaims it.
- No guarantee on eventual reclamation, due to a (rare) race.



# The Drawbacks and the Solution

---

- Main problems in Bacon & Rajan's concurrent algorithm:
  - Practical: reduced efficiency due to repeated traversals.
  - Theoretical: progress is not guaranteed.
- Problem cause: repeated traversals of a graph do not read the same set of nodes and edges.
- [Paz-Bacon-Kolodner-Petrank-Rajan 05]: use a fixed-view of the heap.
  - Repeated traces are bound to trace the same graph.
- A snapshot of the heap (concurrent collection) or a sliding-view of the heap (on-the-fly collection).



# Cycle Collection on a Snapshot

---

- We have seen concurrent collectors that obtain a snapshot
  - By copying data before it is modified.
- Given a snapshot-obtaining mechanism, we can use the simple non-concurrent cycle collector on the snapshot.
  - It is not disturbed by program activity.
- **All** garbage cycles are collected.
  - A garbage cycle created, must exist in next snapshot.
- **Only** garbage cycles are collected.
  - An unreachable cycle in the snapshot is indeed a garbage cycle.



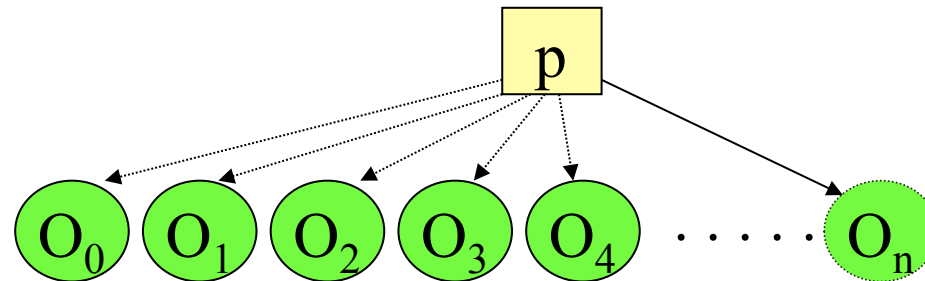
# Using the Levanoni-Petrack Mechanism

---

- Write-barrier records non-null pointer values for each modified object.
- If the object traced is not dirty, its current pointers are relevant.
- Otherwise, its snapshot pointer values are recorded in the threads' local buffers.
- Can we miss candidates? Recall the mechanism...

# Levanoni-Petrank's Reference-Counting

- Consider a pointer  $p$  that takes the following values between GC's:  $O_0, O_1, O_2, \dots, O_n$ .



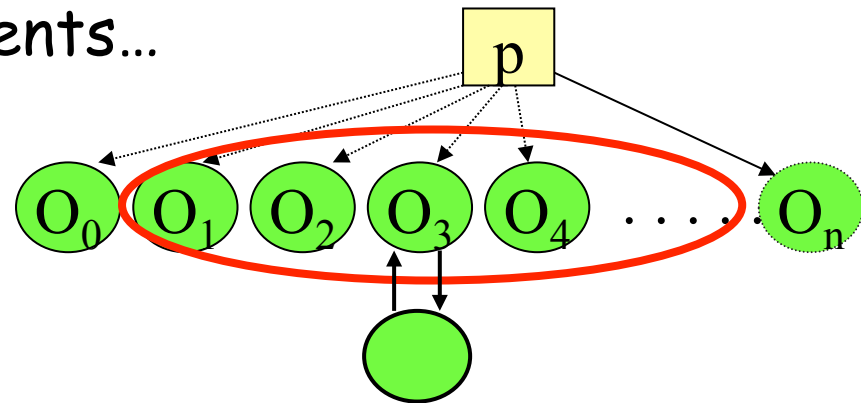
- All RC algorithms perform  $2n$  operations:

~~$O_0.RC--$~~ ;  ~~$O_1.RC++$~~ ;  ~~$O_1.RC--$~~ ;  ~~$O_2.RC++$~~ ;  ~~$O_2.RC--$~~ ; ...,  ~~$O_n.RC++$~~ ;

But only 2 operations are needed:  $O_0.RC--$ ,  $O_n.RC++$

# Less Cycle Candidates

- Until now, we have assumed all objects whose rc is decremented to a non-zero value plus all new objects are candidates.
- But the Levanoni-Petrank write barrier does not log all (or even most) decrements...
- Are we going to miss garbage cycles because of that?
- No! We reclaim all garbage cycles (at a much lower cost). We just need a better analysis.





# Why do we *not* miss a cycle?

---

- **Case I:** the cycle has a young node (created since last collection).
  - Since all young objects are candidates: we'll find it.
- **Case II:** the cycle has no young node (old nodes).
- Consider this cycle in the previous collection.
  - All objects must have existed.
- **Case II-a:** an object in the cycle was directly reachable from the roots in previous collection.
  - Since all roots in previous collection are candidates, we'll get it.



## Why do we *not* miss a cycle?

---

- **Case II-b:** the cycle was reachable from some external pointer (and not from a root).
- That external pointer was modified since the previous view, or its object died.
- That pointer modification must have been logged, which implies a candidate.
- Or the collector reclaimed the object, decremented the child's rc, which implies a candidate.



# Gain

---

- Effectiveness of cycle collection depends on the number of objects traced.
- The Levanoni-Petrank technique simplifies the algorithm by using a snapshot view efficiently.
- But it also reduces the number of candidates, because we do not record many of the (irrelevant) pointer modifications.
- Additional ways to reduce tracing work:
  - Lazy tracing.
  - Trace all candidates simultaneously.
  - Do not trace acyclic objects.



# Further Reducing Traced Objects

---

- Most candidates do not belong to a garbage cycle. How can we further reduce the cycle tracing work?
- **Idea 1:** let candidates mature more before tracing them.
- Check only candidates recorded  $k$  collections ago, which have not “blinked” since.
  - Have not died,
  - Their rc has not changed,



## Idea 2: Known Live Objects

---

- Some objects are known to be alive and the cycle collector may ignore them. In particular:
- Recently dirtied objects,
- Objects referenced by the roots,
- Objects appearing in later buffers (whose rc has changed since buffered).



# Idea 3: More Saving for Known Live Objects

---

- The scan phase may sometimes color a sub-graph white, and only later discover it is actually alive.
- Colors the same sub-graph twice!
- Idea 3: try to save such double colorings by starting to color black the externally reachable sub-graphs, before coloring objects in white.
- Use the above list of known live objects.
- If they exist in the gray graph, start coloring their sub-graph black, before proceeding with the scan phase.



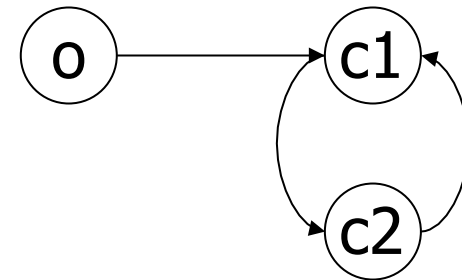
# Cycle Collection on Heap's Sliding-View

---

- The collector can be made on-the-fly using the Levanoni-Petrank techniques as in previous lecture.
  - No simultaneous halt of all program threads.
- But, now the snapshot becomes a sliding-view.
- The cycle collector traces the sliding-view graph instead of the snapshot graph.
- A garbage cycle created before a collection begins, must exist in the sliding-view, and is thus reclaimed.
- Can the sliding-view include an un-reachable cycle, which never existed in reality?

# False Cycles in the Heap's Sliding-View

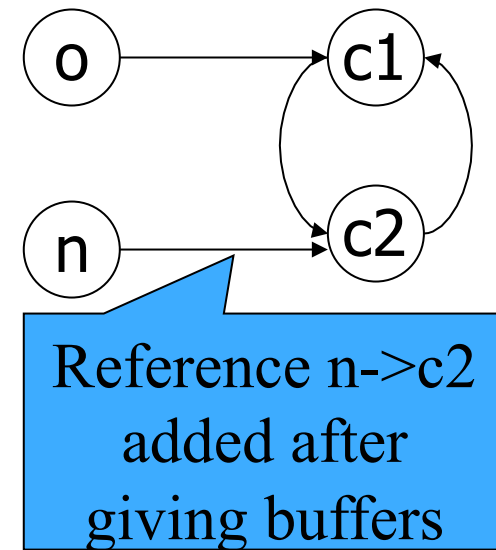
Example:



# False Cycles in the Heap's Sliding-View

## Example:

1. Thread 1 cooperates with the collector, passing its local buffers to the collector.
2. Thread 1 creates a new object  $n$ , that references  $c2$ .



# False Cycles in the Heap's Sliding-View

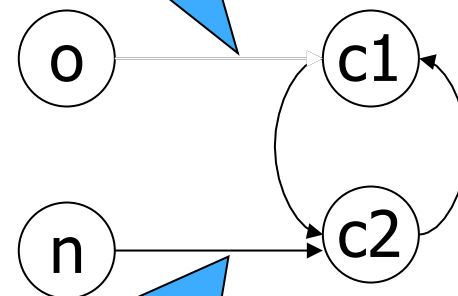
## Example:

1. Thread 1 cooperates with the collector, passing its local buffers to the collector.
2. Thread 1 creates a new object  $n$ , that references  $c2$ .
3. Thread 2 deletes the pointer  $o \rightarrow c1$ .
4. Thread 2 cooperates with the collector, passing its local buffers to the collector.

⇒ The collector might “think” that  $c1$  and  $c2$  compose a garbage cycle.

- Solution- **the snooping mechanism**:
  - Snooped objects ( $c2$ ) are ignored by the cycle collector.

Reference  $o \rightarrow c1$   
removed before  
giving buffers



Reference  $n \rightarrow c2$   
added after  
giving buffers



# Remarks on Snoopied Objects

---

- Like roots, snoopied objects are considered candidates for the next collection.
- Snoopied objects are considered part of the set of “known live objects”, and help reduce the tracing work.



# Measurements Setting

---

- Implemented in Jikes with two collectors:
  - The Levanoni-Petrank collector.
  - The age-oriented collector.
- Some words on the age-oriented collector:
  - Uses mark and sweep for the young generation and reference counting for the old generation.
  - A very reasonable choice for generations...
  - Not collecting the old is a huge saving: no automatic candidacy of newly created objects...

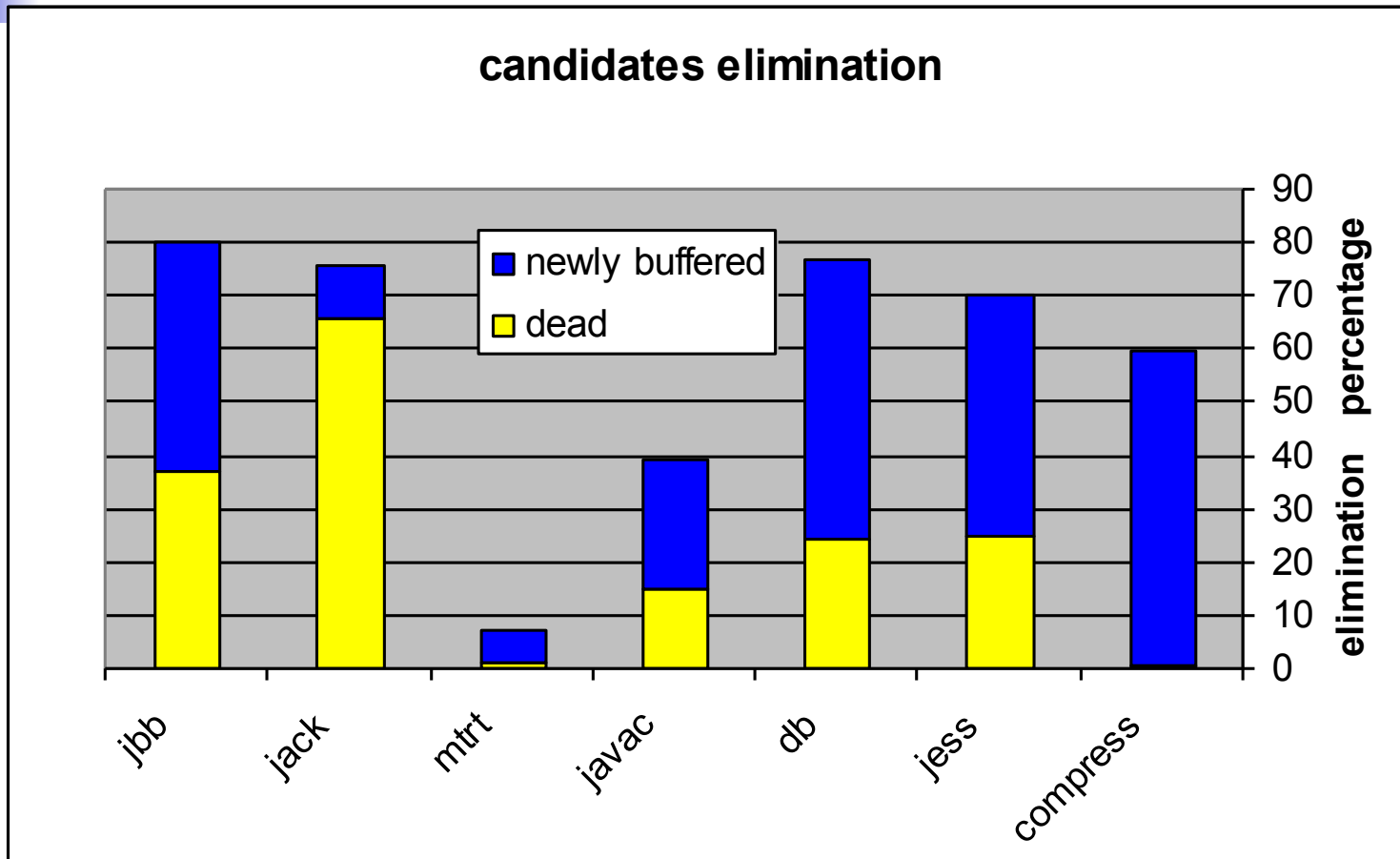


# Measurements

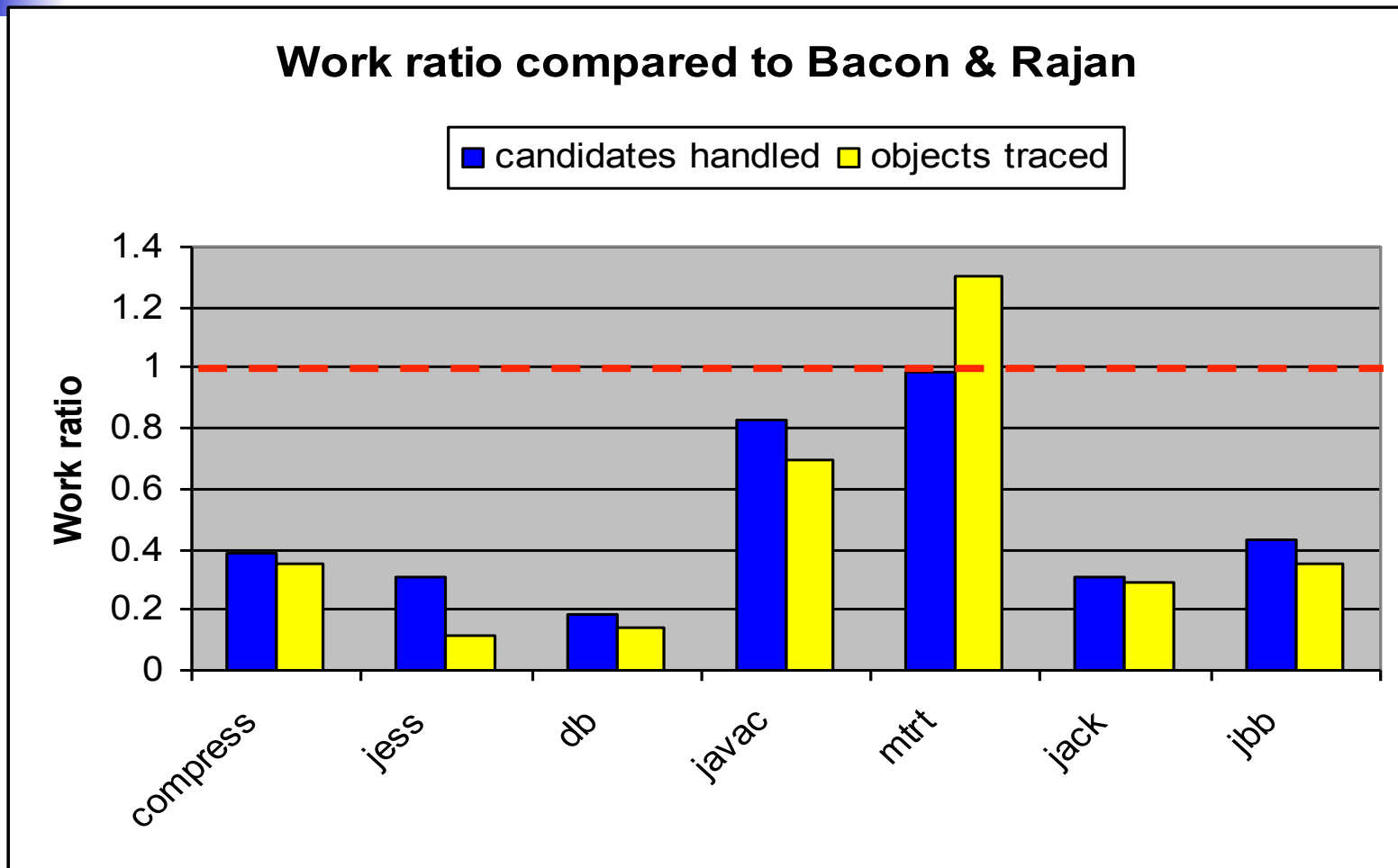
---

- Measurements:
  - Throughput comparison between cycle collection and a backup tracing collector.
  - Characteristic comparison to the previous on-the-fly cycle collector.

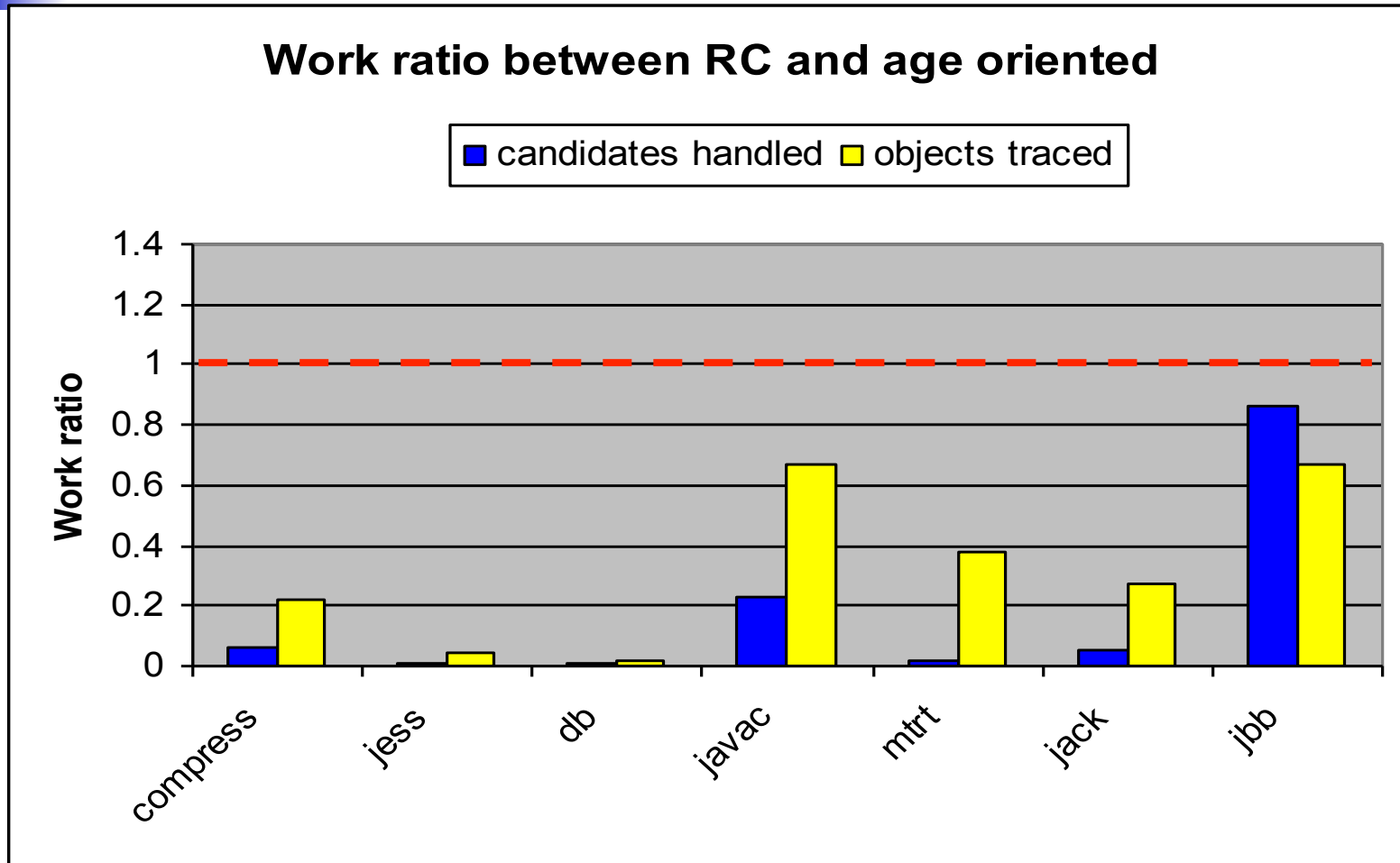
# Reduction in Candidates



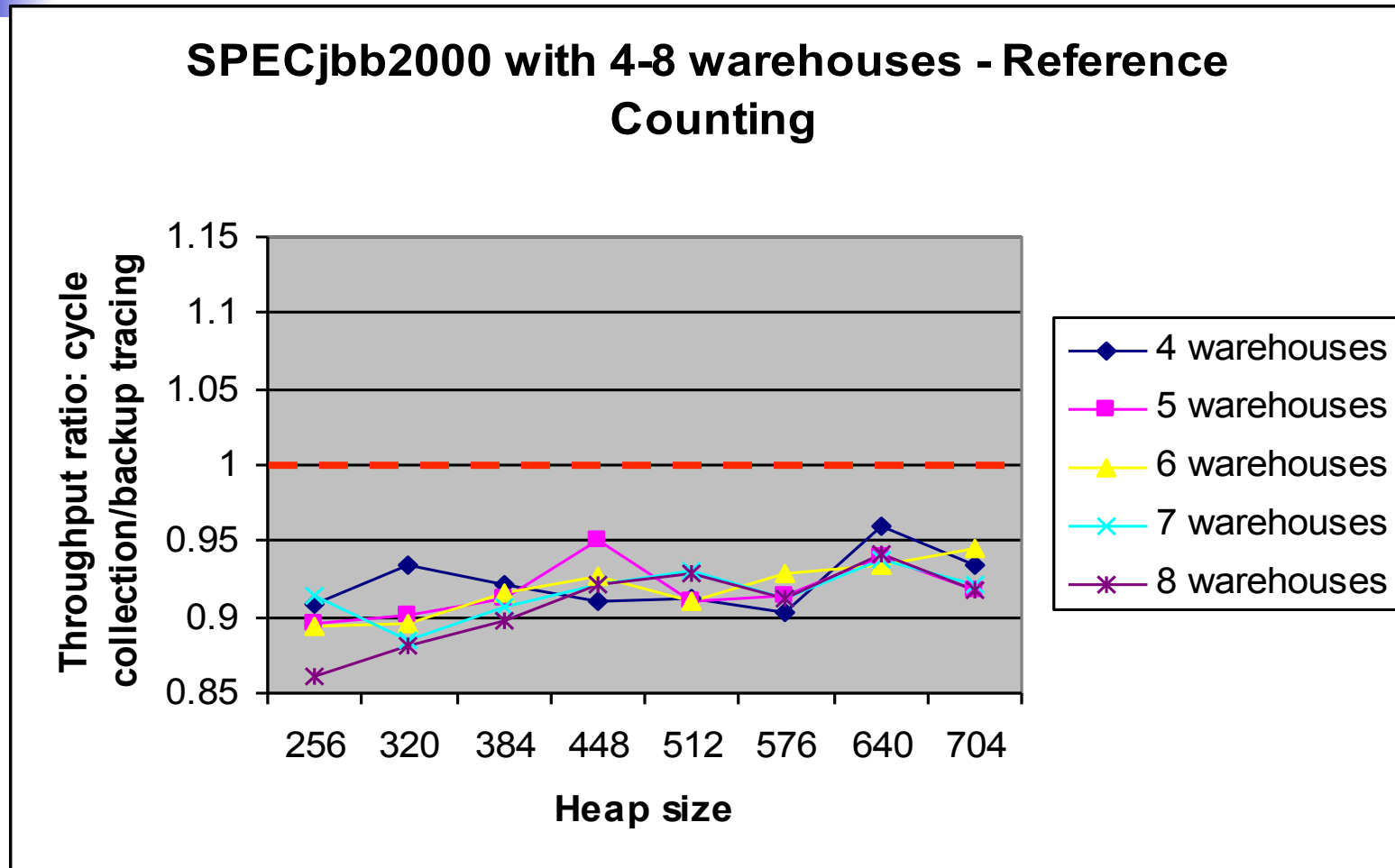
# Work Reduction



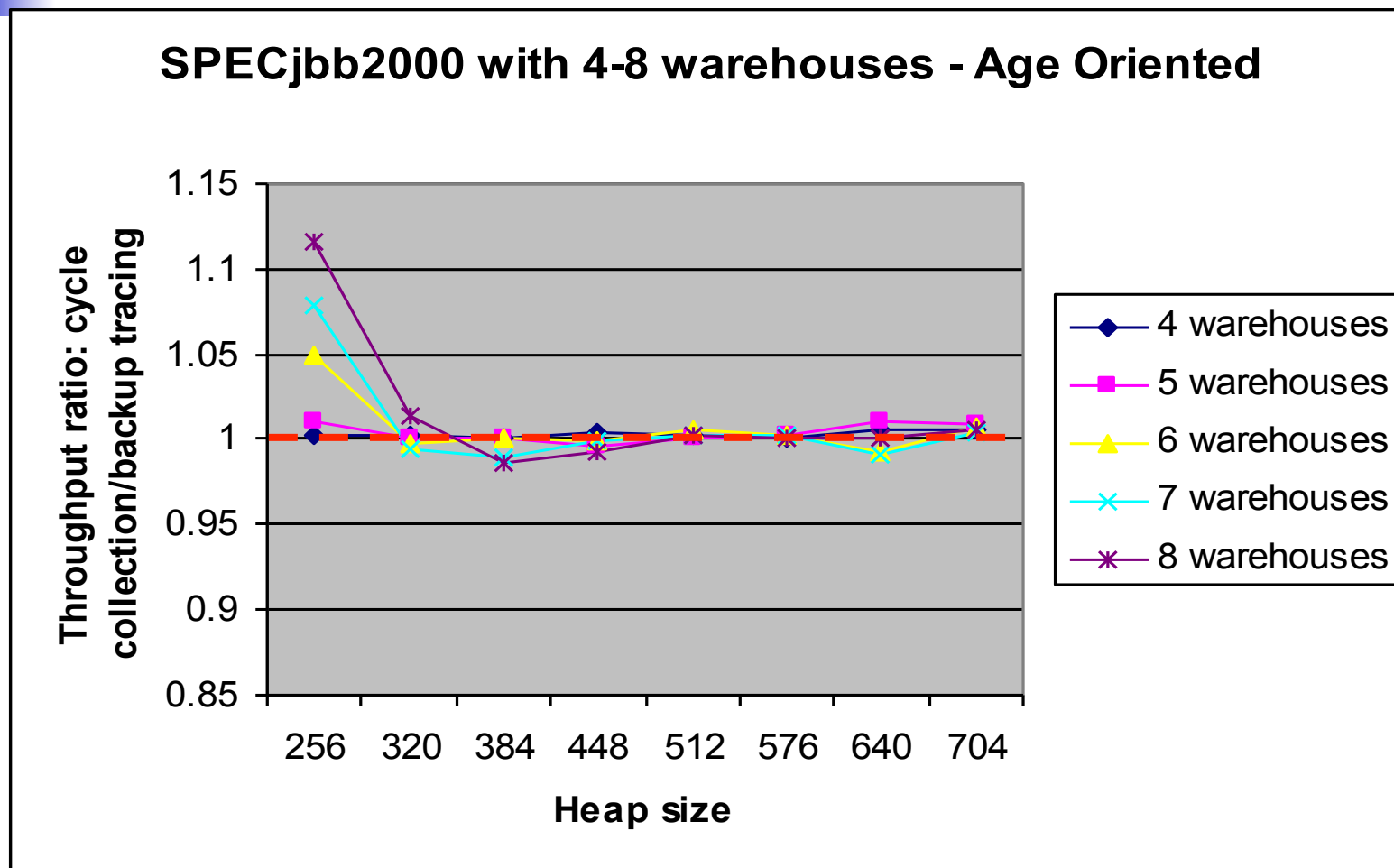
# Work Reduction with the Age-Oriented Collector



# Throughput Comparison of Cycle-Collection Vs. Backup Tracing



# Throughput Comparison: Cycle-Collection vs. Backup Tracing





# Conclusion

---

- We've seen:
  - Cycle collection basic algorithm
  - Concurrent cycle collection using Levanoni-Petrank
- Measurements with today's benchmarks:
  - Reference counting for full heap: prefer a backup tracing.
  - Reference counting for old generation: slight preference to cycle collection.
- Eyes for the future: with large heaps cycle collection may outperform backup tracing.

# Concurrent and Parallel Compaction

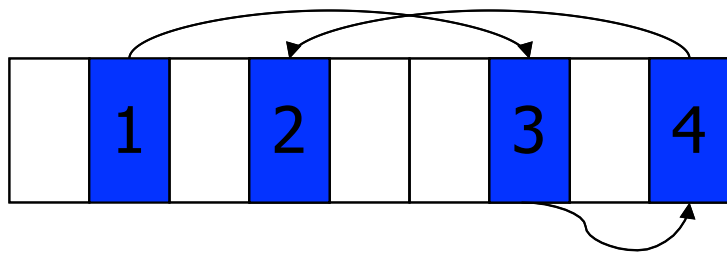
# The Generic Task

- Assume live objects are marked (i.e., the mark phase is done).
- The algorithm **moves** objects to one (or a small number of) areas in the heap
- Pointers are **modified** to reference the new locations.

# Comparison Criteria

- Complexity:
  - Number of heap passes,
  - Number of table passes,
  - Cache performance.
- Extra space required.
- Restrictions on objects (e.g., equal size).
- Compaction quality:
  - Order of objects in output.
  - Number of packed areas (best: 1 area).

# Object Ordering



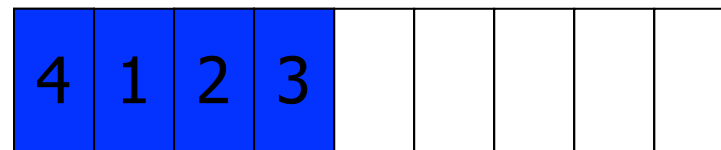
**Arbitrary** - no guaranteed order.

**Linearizing** - objects pointing to one another are moved into adjacent positions.

**Sliding** - objects are slid to one end of the heap, maintaining the original order of allocation.

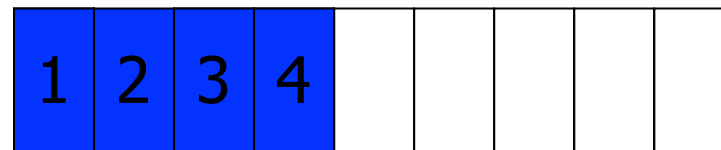
Arbitrary

Worst



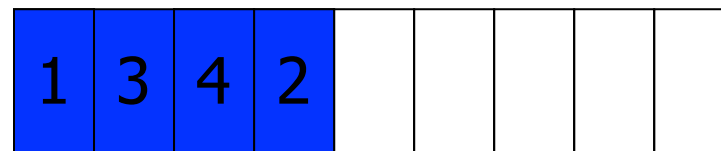
Sliding

Best



Linearizing

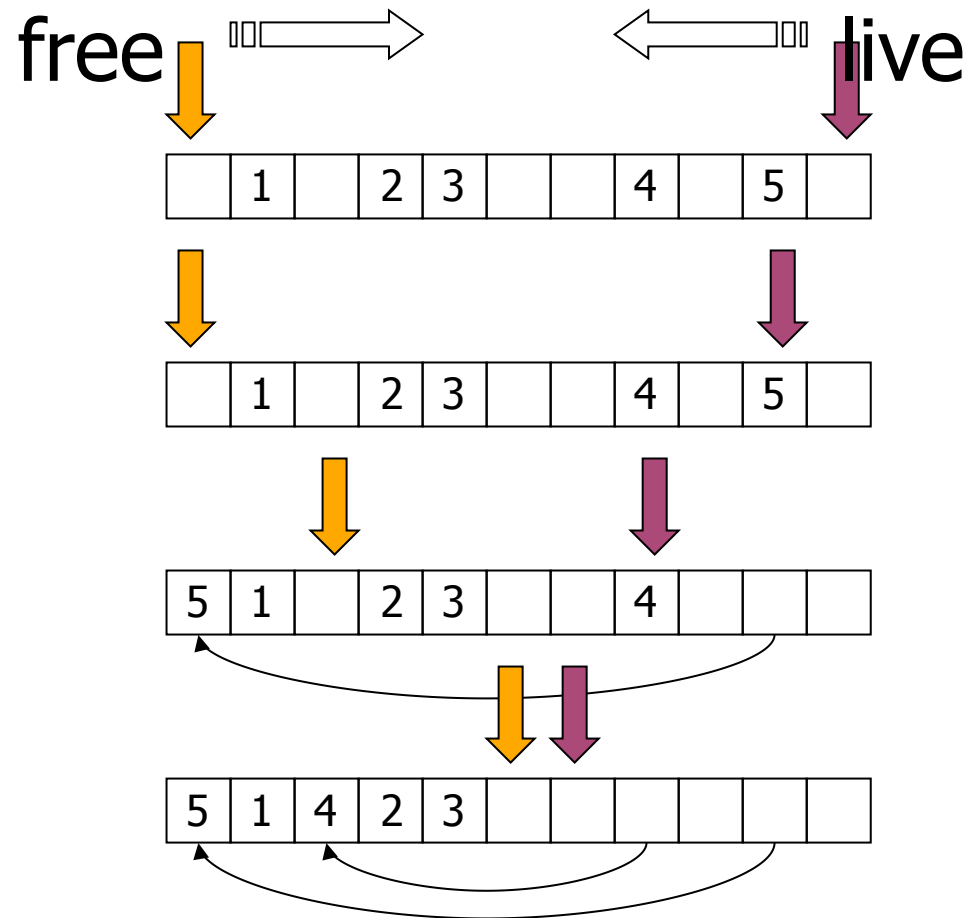
Good



# Concurrent, Parallel, and Incremental Compaction

- Now that we understand concurrence and parallelism, we go back to compaction (from lecture 2).
- Let's shortly recall previous algorithms, and then introduce the Compressor.

# Two finger, pass I - Example

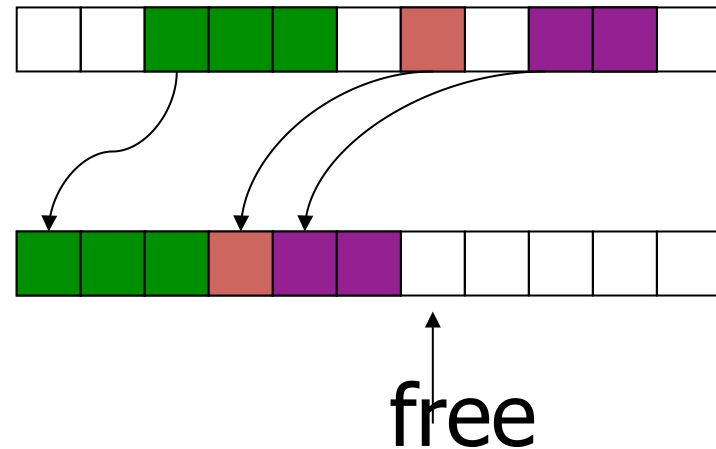


# Two finger - Properties

- 😊 Simple!
- 😊 Relatively fast: One pass + pass on live objects (and their previous location).
- 😊 No extra space required.
- 😞 Objects restricted to equal size.
- 😞 Order of objects in output is arbitrary.
  - 😞 This significantly deteriorates program efficiency! Thus - not used today.

# The Lisp2 Algorithm

- **Pass 1:** Address computation. Keep new address in an additional object field.
- **Pass 2:** pointer modification.
- **Pass 3:** two pointers (free & live) run from the bottom. Live objects are moved to free space keeping their original order.



# Lisp 2 - Properties

- 😊 Simple enough.
- 😊 No constraints on object sizes.
- 😊 Order of objects preserved.
- 😞 Slower: 3 heap passes.
- 😞 Extra space required - a pointer per object.

# Jonker's Algorithm [1979]: Eliminate Extra Space

- Basic idea (threading method):  
for each object *A* keep list of all pointers that reference it. (The pointers are “threaded”.)  
When object *A* moves - update all pointers in the list.
- Actually, Deal first with pointers pointing forwards, then with backwards pointers.

# Threaded Compaction- Analysis

- No extra space required
- Variable size objects
- Preserves order
- Two passes
- But -
  - each iteration may touch several other objects.
  - requires a header distinguishable from pointer.

# Parallel Compaction: SUN's Version

- [Flood Detlefs Shavit Zhang 2001]
- First parallel compaction (for SMP)
- 3 phases (similar to the LISP 2 algorithm):
  - Forwarding pointers installation
  - Fix up pointers phase
  - Compaction phase
- Each phase done in parallel

# Properties

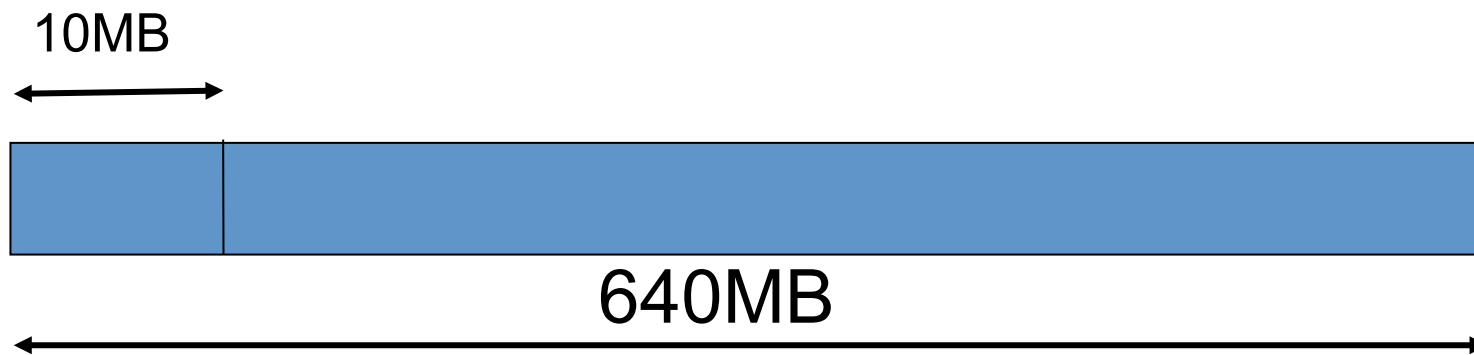
- 😊 Runs in parallel - good scalability!
- 😊 Keeps order of objects
- 😞 Objects are not packed
- 😞 Requires extra word per object (or a smart use of the reclaimable space)
- 😞 Coarse-grained load balancing
- 😞 3 heap passes

# IBM's Parallel Compaction

- [Abuaiadh-Ossia-Petrank-Silbershtein 2004]
- A more involved parallelization of the LISP-II compaction algorithm.
- Unlike SUN: Objects are packed to the bottom.
- Better than LISP-II: A smaller (and controllable) space overhead: keep some small table instead of a forwarding pointer in each object.
- More efficient --- 2 heap passes (each executed in parallel):
  - **Move** and keep some info
  - Use info to **fix up** pointers

# Parallelism versus Compactin

- We want to compact all objects together and not make several piles of objects.
- Heap is divided to  $n$  **areas**
- For example:  $n = 64$  was used for 640MB heap and 8 processors.



# Properties

- Almost all objects are condensed to the bottom of the heap.
- Order of objects is essentially preserved.
- Good parallelism with almost no contention.
- Space overhead low compared to forwarding pointers.

# Conclusion --- IBM's Parallel Compaction Algorithm

- More efficient than the previously used threaded algorithm even on a uniprocessor.
- Good speedup
- Good compaction quality.

# The Compressor

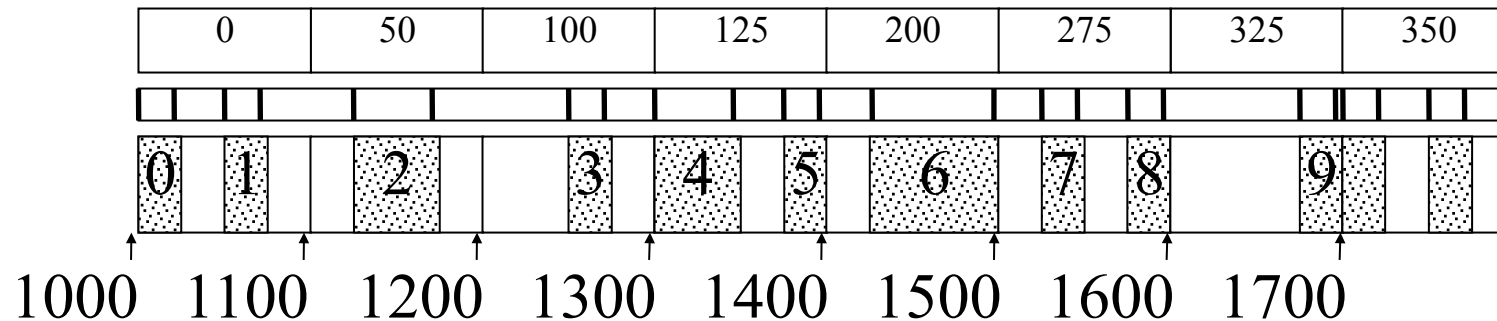
- [Kermany-Petrank 2006]
- The goal: concurrent and parallel compaction with low overhead.
- Overhead reduction via a **single heap pass**.
- Extensions include parallelism and concurrency:
  - Use virtual machine primitives:
  - Assign a new virtual space and copy the compacted areas into it.
  - This allows moving and fixing pointers at the same time.
- Objects are packed to the bottom, maintaining address order.

# Compressor - Overview

- Compute locations of new objects (markbits pass)
- Move objects + fix their pointers (heap pass)
- Assume a mark-bit exists providing a bit for each beginning of heap object, and each end.
- Typically, such a bit-map is produced by a mark-sweep collector.
- A pass over this mark-bit vector, allows computing a target address for each object.

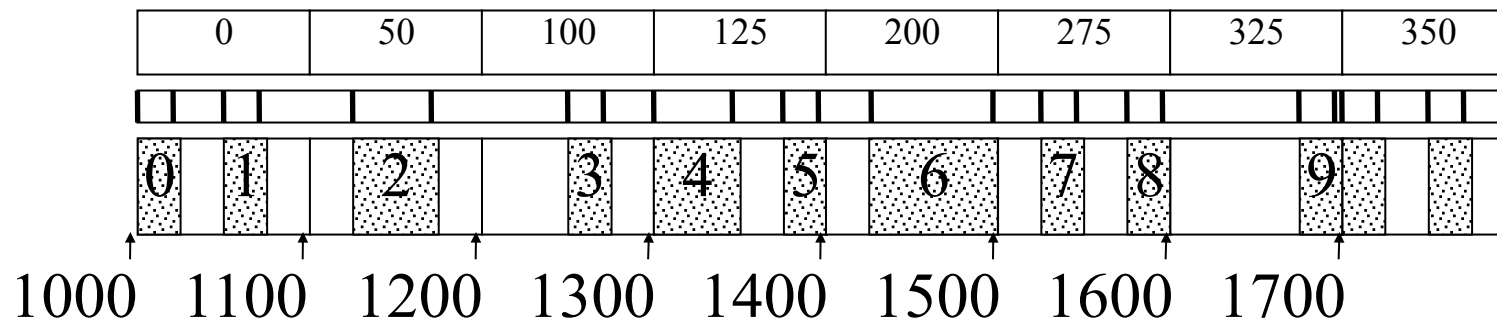
# Compressor - Overview

- Use a table to store relocation info succinctly.
- Heap partitioned to blocks (typically, 512 bytes).
- Start by computing for each block the total size of objects preceding that block (**the offset vector**).
- Requires a single pass over the mark-bit vector.
  - Assume: a **markbit vector** with a bit set for beginning and ending word of each object.



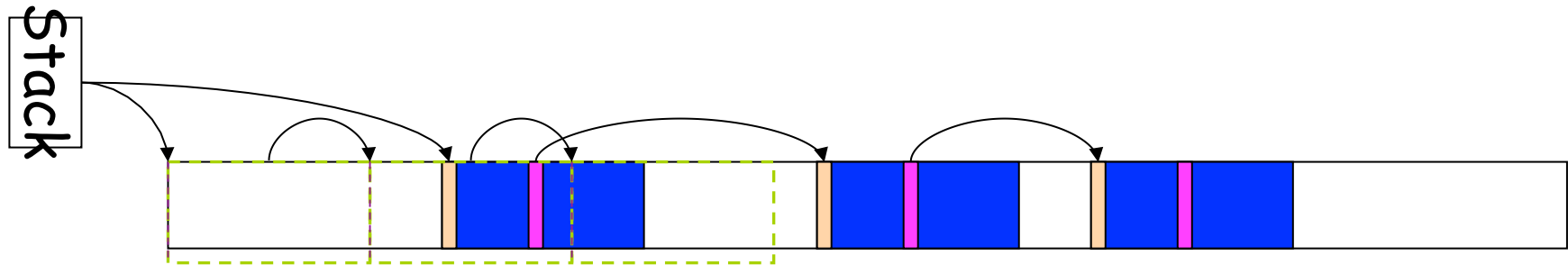
# Using the Offset Vector

- Computing an address from the offset vector:
- The new address of an object  $O$  in block  $B$  is the offset vector entry of  $B$  plus the size of the objects in  $B$  preceding  $O$ .
- For object 5:  $Fix-up(1375) = 1000 + 125 + 50 = 1175$ .
- (Give up the additional IBM structures.)



# The Basic Compressor (Single-Threaded Compaction)

- Stop application threads.
- Calculate Offset vector.
- Fix roots.
- For every object in address order:
  - Move object
  - Fix its pointer.
- Resume application threads.



# Properties

- One table pass, one heap pass = efficient.
- Small space overhead (the offset vector).
- But - Single threaded.
- The main supporting invariant:
  - “Old object version is not stepped on before it moves”.
  - Due to ordered move of objects.
- This invariant cannot be guaranteed when parallel threads move objects in parallel.

# Parallelization

- If we had two spaces ...
- The Compressor could compact the objects from one space (**from-space**) into the other (**to-space**).
- Advantage: each object can be moved independently of the others → Simple parallelization
- Problem: space overhead.

# Virtual Memory Reminder

- A process has a large **virtual** (memory) address space, maybe larger than physical memory.
- This space is divided into virtual pages (typically 4KB)
- Some of the virtual pages are kept on physical memory while the rest reside on the hard disk.
- The system keeps a map from virtual to physical pages.
- Due to locality of computation, **page faults** (pages that are not available in the memory and must be retrieved from the disk) are seldom.

# Virtual Memory Primitives

- A virtual page is *mapped* into a physical page when a space on that page is allocated.
- Virtual pages consume “real” resources (such as space on disk and memory) when mapped.
- A virtual page can be *released* or *unmapped* when the process does not need the data on it anymore.
- A physical page can be mapped from two different virtual pages.
  - Changing one virtual address will affect the other...

# Virtual Memory Primitives

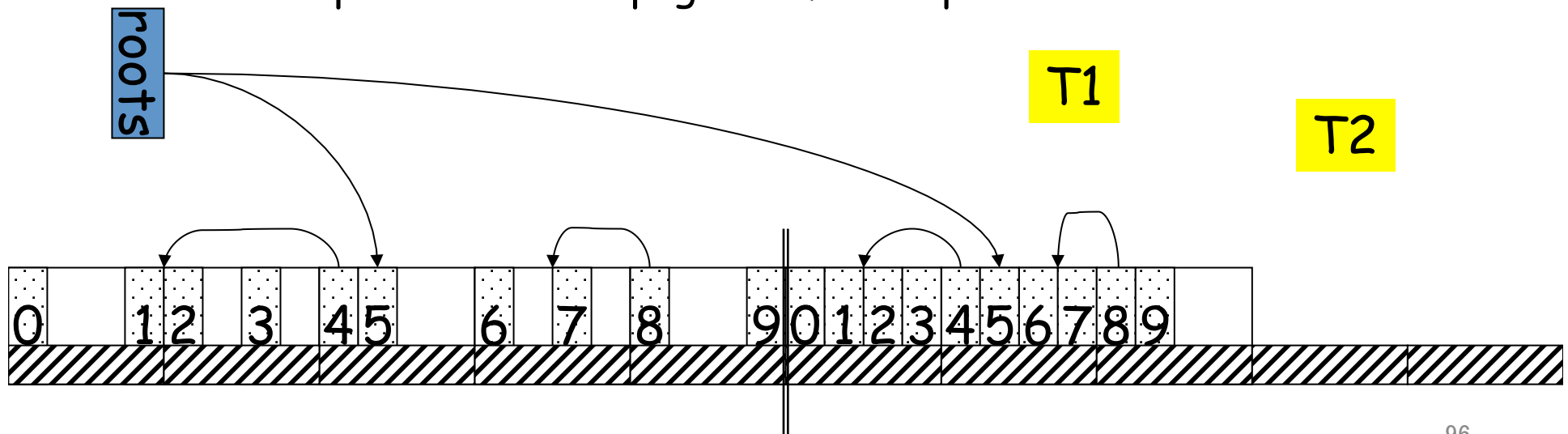
- A virtual page can be *read-protected* or *write-protected*.
- When the process tries to read (or write) from a read- (or write-) protected page, a **trap** springs.
- A **trap handler** code is invoked at that time.

# Parallel Compressor without Double-Space Overhead

- Initially, **to-space** is not mapped to physical pages.
  - It is a virtual address space.
- Execute in parallel: for every (virtual) page in **to-space**:
  - Map the virtual page to physical memory.
  - Move the corresponding **from-space** objects & fix pointers.
  - Unmap the relevant pages in from-space.

# Parallel Compressor without Double-Space Overhead

- Initially, **to-space** is not mapped to physical pages.
  - It is a virtual address space.
- Execute in parallel: for every (virtual) page in **to-space**:
  - Map the virtual page to physical memory.
  - Move the corresponding **from-space** objects & fix pointers.
  - Unmap the relevant pages in from-space.



# Parallel Compressor Algorithm

- Stop application threads.
- Calculate Offset vector.
- Fix roots.
- Run in parallel: while there are pages in to-space to handle:
  - Map the page.
  - Move the objects to the page and fix their pointers.
  - Unmap the unnecessary pages in from-Space.
- Resume application threads.

# Problem with Concurrency

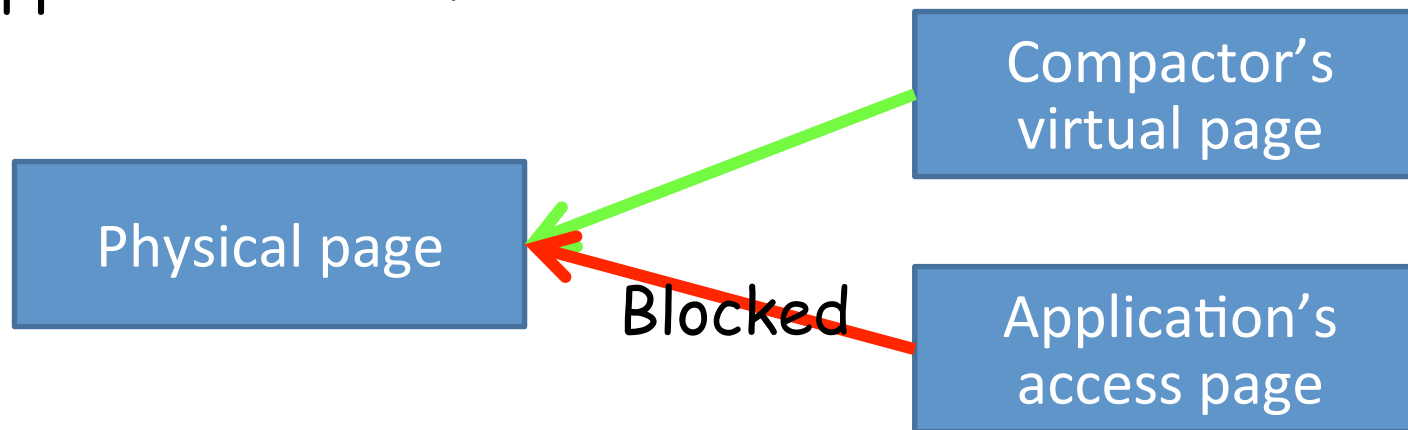
- If we move objects while application is running, two copies appear.
  - It is difficult to synchronize all threads to properly switch between using one copy or the other.

# Concurrent Compressor

- Solution: at a well-defined point in time, all application threads stop using old versions and start using only moved objects (in to-space).
  - Initialization: all threads are stopped, and
    - Roots are modified to point to to-space,
    - All to-space pages are read-protected, and
    - Application resumes.
  - Trying to touch a to-space page springs a trap.
  - The trap handler moves the relevant objects into the to-space page and lifts protection.

# Fixing To-Space

- How can the trap handler fix a protected to-space while it is still protected?
- Lifting the protection is not an option, because other threads may touch the non-ready page.
- Solution: double mapping. The handler reaches the page via a different virtual space, unprotected and mapped to the same location.



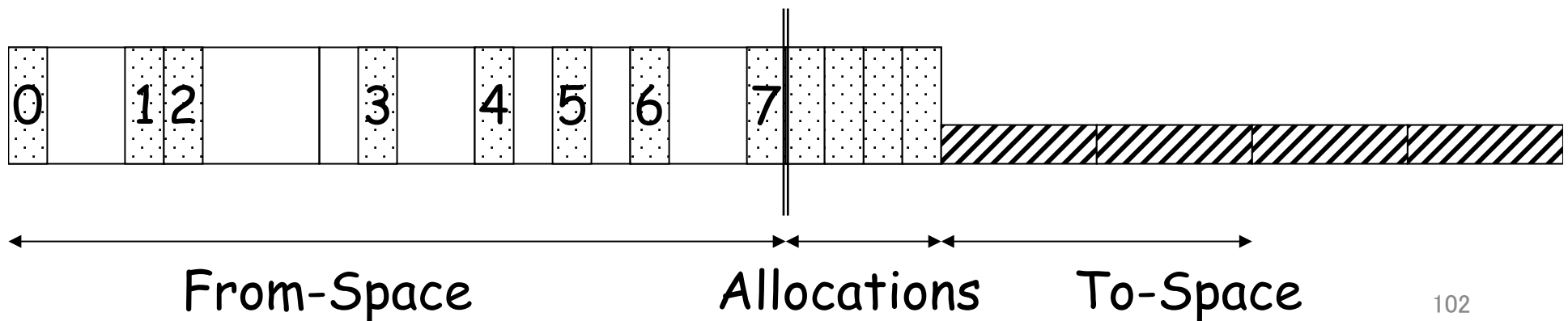
# A Simple Concurrent Solution

- Stop application threads.
- Calculate offset vector.
- Protect to-space.
- Fix roots.
- Resume application threads.
- Move objects via a trap handler.



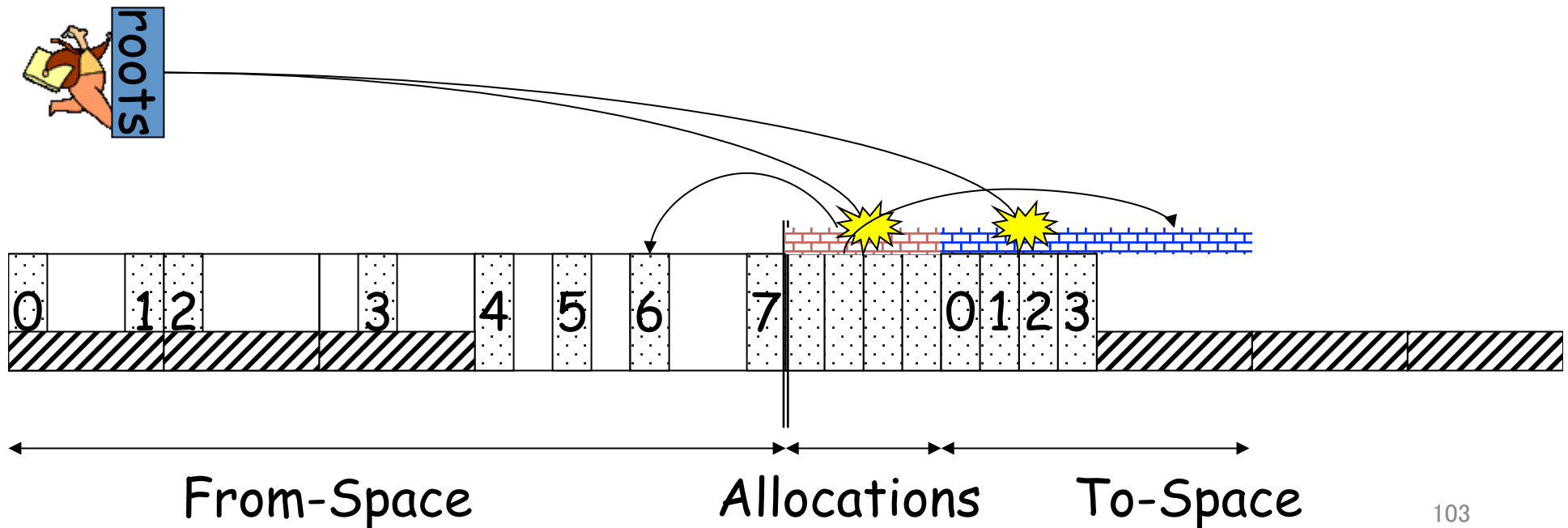
# Concurrent Solution - Offset Vector

- The offset vector can be calculated without stopping the application threads:
  - The calculation is done concurrently by the application threads.
  - Done only for old objects.
  - *New objects* are allocated in to-space.



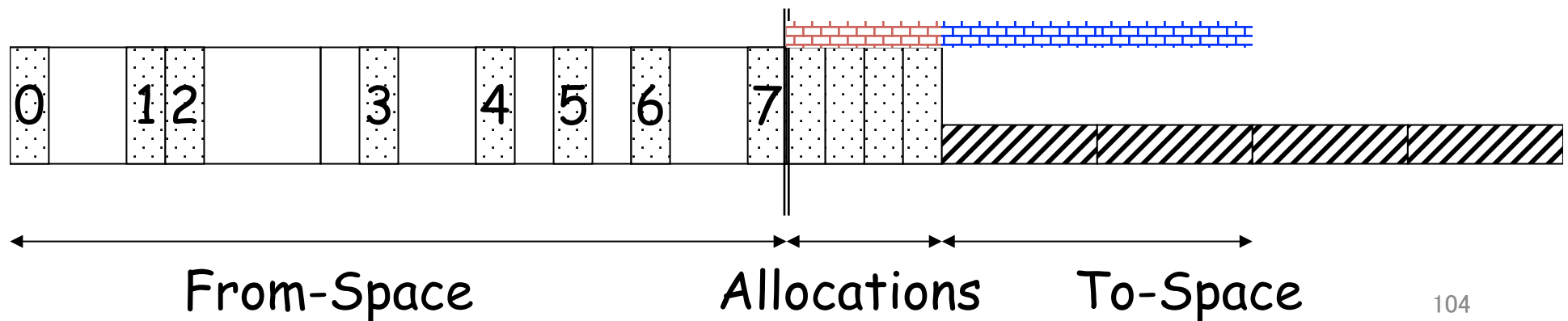
# Concurrent Solution - Offset Vector

- New objects may contain pointers to from-space.
- New object pages are later protected.
- Pointers in new objects are fixed via traps.



# Concurrent Solution - Protection

- Turning on protection of to-space pages can be done while the application runs, because these pages are not in use.
- Protection of pages with new objects must be done while the application is stopped.



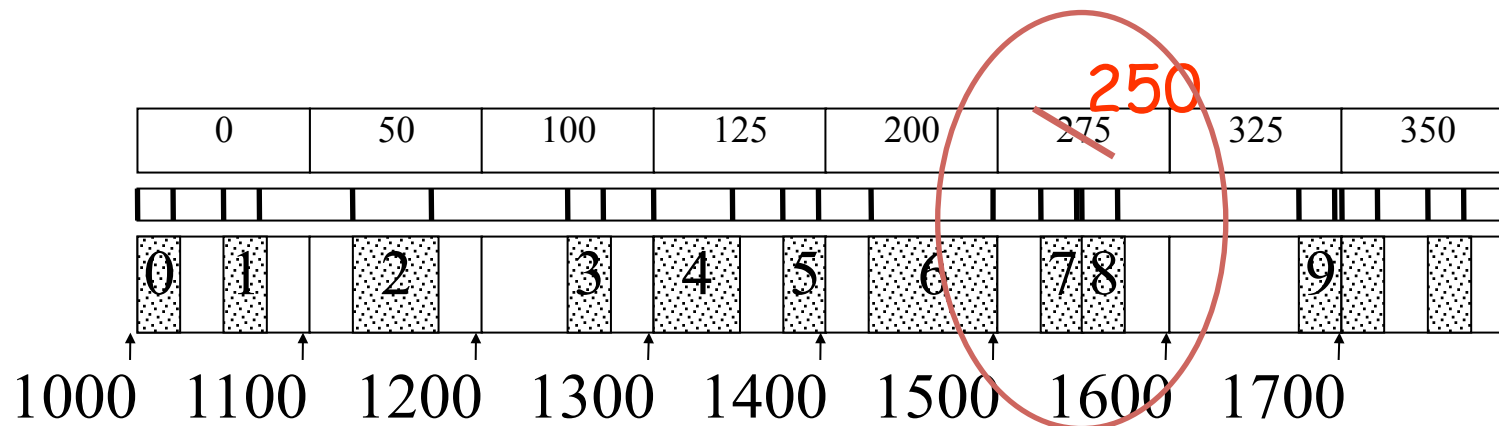
# Better Concurrent Solution scheme

- Stop application threads; “tell them” to start allocating in new object space; resume threads.
- Calculate offset vector.
- Turn on protection on to-space.
- Stop application threads.
- Fix roots.
- Protect the new objects space.
- Resume application threads.
- Move the objects via a trap handler.
- Fix the new objects via a trap handler.

# Fix-up function - an improvement.

In case the block is *dense*:

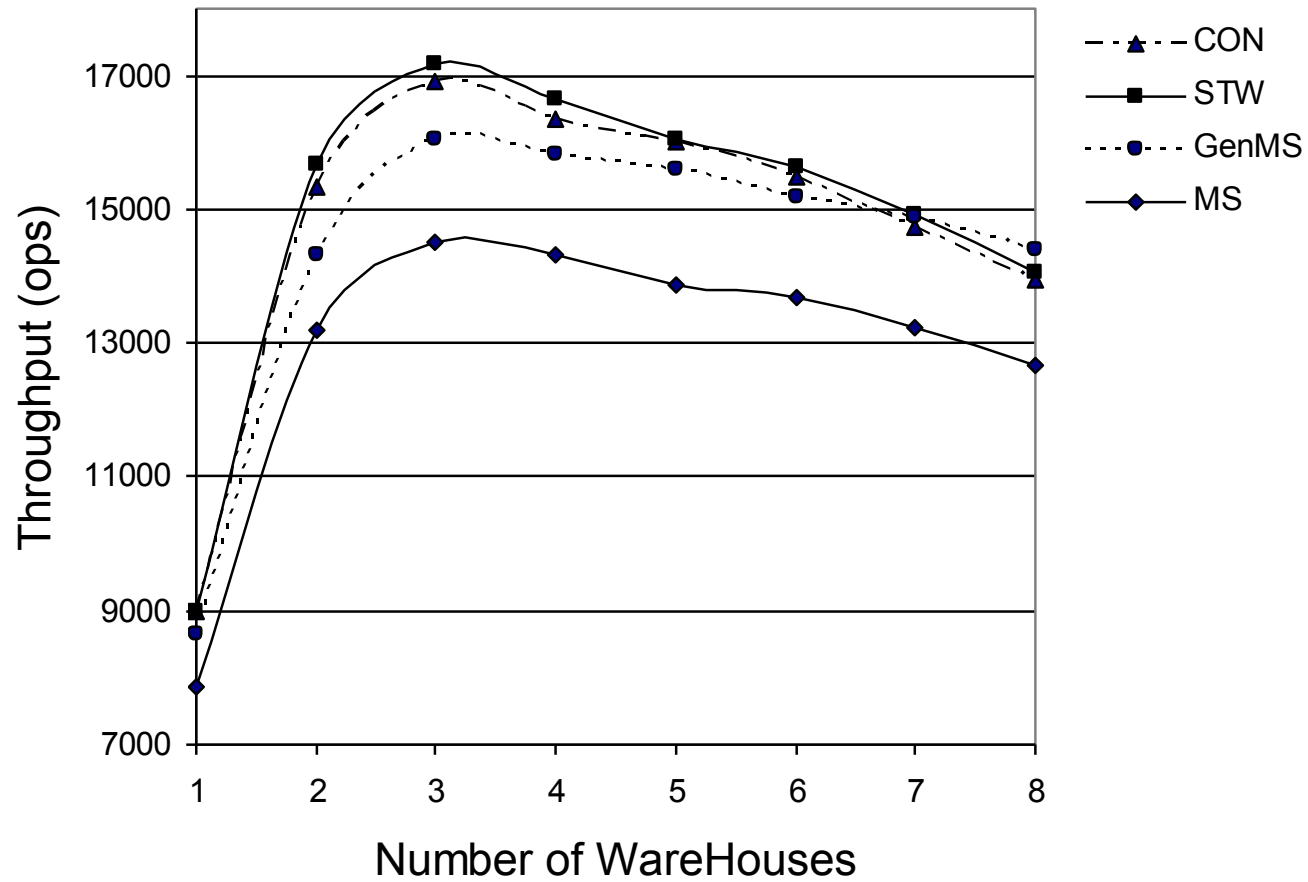
- Object 7:  $Fix-up(1525) = 1000 + 275 + 0 = 1275 (=1525 - 250)$ .
- Object 8:  $Fix-up(1550) = 1000 + 275 + 25 = 1300 (=1550 - 250)$ .
- In this block, new address = old address - 250, because it is dense. It turns out many blocks get dense with time.
- Instead of keeping the 275, keep 250.
- The LSB distinguish between the cases.



# Implementation & Measurements

- The Compressor was implemented on the Jikes RVM - a research Java Virtual Machine.
- The main benchmark is the Specjbb2000, A server application running one to eight threads.
- The Compressor performance was compared to the performance of Mark-Sweep and GenMS.
- Unfortunately, there were no concurrent nor compaction algorithms on the Jikes RVM, that could be compared.

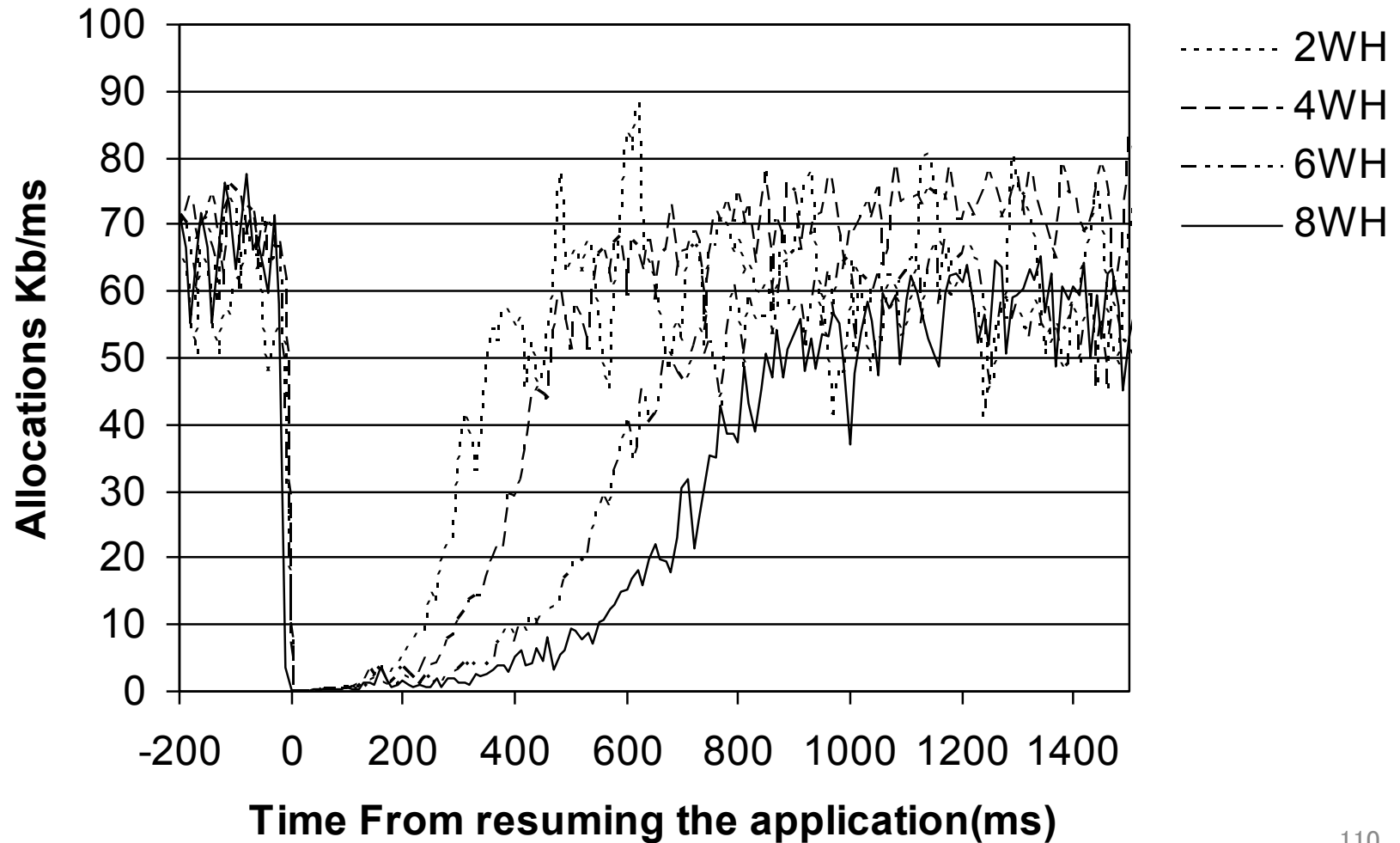
# Measurements - efficiency



# Measurements - pause time

	Parallel Compaction (Stop-The- World)	Mark and Sweep	generational Mark and Sweep (full collections)
jbb 2-WH	319.55	229.37	279.73
jbb 4-WH	516.89	287.32	323.64
jbb 6-WH	641.53	315.71	347.42
jbb 8-WH	770.14	372.46	374.41

# Measurements - Allocations per time



# Conclusion

## The Compressor:

- The first compactor with one heap pass (in addition to a table pass).
- Fully compact all the objects in the heap.
- Preserves the order of the objects.
- Low space overhead.
- Uses memory services to obtain parallelism.
- Uses traps to obtain concurrency.

# Conclusion --- Compaction

- Uniprocessor compaction:
  - Two fingers, Lisp2, Threaded (Yonkers)
- Parallel compaction:
  - Sun's compaction, IBM's compaction.
  - Parallel and Concurrent: the Compressor.
- Issues considered:
  - Efficiency, space overhead, parallelism, compaction quality, locality.