Algorithms for Dynamic Memory Management (236780)

Lecture 9

Lecturer: Erez Petrank

Last Week

- Snapshot concurrent collection
- On-the-fly mark-sweep via sliding views.

Topic Today

- Reference counting
  - Basic reference counting
  - Improvements: Lazy freeing, Limited field, Deferred RC
  - Problems with concurrent RC
  - The Levanoni-Petrank collector (using sliding views and more).

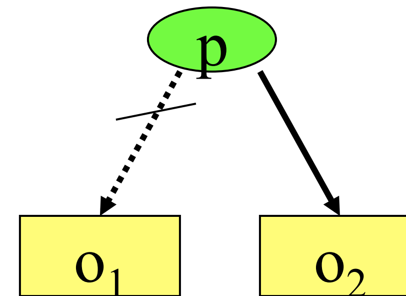
Reference counting [Collins 1960]

- Associate a reference count field with each object: how many pointers reference this object.
- When nothing points to an object, it can be deleted.

© Erez Petrank

Basic Reference Counting

- Each object has an rc field, new objects get o.rc:=1.
- When *p* that points to o_1 is modified to point to o_2 we do: o_1.rc--, o_2.rc++.



- if then o_1.rc==0:
  - Delete o_1.
  - Decrement o.rc for all descendants of o_1.
  - Recursively delete objects whose rc is decremented to 0.

3 years later…

- [Harold-McBeth 1963] Reference counting algorithm does not reclaim cycles!
- But, it turns out that “typical” programs do not use too many cyclic structures.
  - This is correct also for most modern benchmarks (including SPECjbb2000).
- So, other methods are used “seldom” to collect the cycles.

Motivation for RC

- Initially (naïve algorithms):
  - Simple implementation
  - Immediate reuse of unreachable objects
  - Good locality of reference
  - Overheads distributed throughout computation
- Today (complex algorithms):
  - Reference Counting work is proportional to work on creations and modifications.
    - Can tracing deal with tomorrow's huge heaps?
  - And still, good locality.

- Initially (naïve algorithms):
  - Cost of removing last ptr to an object unbounded
  - Overall overhead is greater then that of tracing G.C.
  - Substantial space overhead (counter per object)
  - Inability to reclaim cyclic data structures
- Today (complex algorithms):
  - Overall overhead still greater than tracing
  - Inability to reclaim cyclic data structures

- We will next go through some ideas for improving naïve RC.
- These ideas were mostly proposed in the 60's-70's.
- Not much engineering evolvement until 2000 because tracing algorithms dominated commercial products.
- We will explain why and how this is changing lately.

RC Improvements

- Lazy Freeing
- Limited count field
  - 1-bit rc
  - Software cache
- Deferred RC

Later, we'll move into concurrent RC

Lazy Freeing [Weizenbaum, 1963]

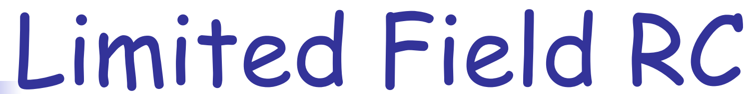
- Problem: uneven processing overheads.
  - Cost of deleting last pointer to an object O depends on size of sub-graph rooted at O.
- Solution – lazy freeing:
  - Free lazily with a stack.
  - When last pointer to O deleted: push O to stack.
  - During allocation: pop O from stack, free O, decrement rc for children of O.
    If any got down to 0 – push it to the stack.

© Erez Petrank

Lazy Freeing Properties

- Advantage:
  - Splitting the costs of deletion evenly throughout the computation.
    - Efficiency (almost) unchanged: deletion is just spread into many allocation operations.
- Disadvantages:
  - Complication of the allocation process
    - It may take time to realize that a large consecutive space has been freed.

Limited Field RC



- Problem: Space overhead for ref counters.
  - Theoretically, large enough to hold number of pointers in the heap - as large as a pointer.
- Idea: use small rc fields (may overflow).
- When rc overflows, think of it as stuck.
- Stuck rc's are not decremented/incremented.
- To reclaim objects with stuck rc, their rc values must be restored.
- This is done with a tracing collector (the one that reclaims cyclic data structures).

Restoring Reference Counters

```
Mark_sweep()=
    for Obj in heap
        RC(Obj) = 0
    for p in Roots
        mark( *p )
    sweep()
    if free-list is empty
        abort "Out-of-Memory"
```

```
mark( Obj )
    incrementRC( Obj )
    if ( Obj.rc ) == 1 //first visit
        for C in Children ( Obj )
            mark( C )
```

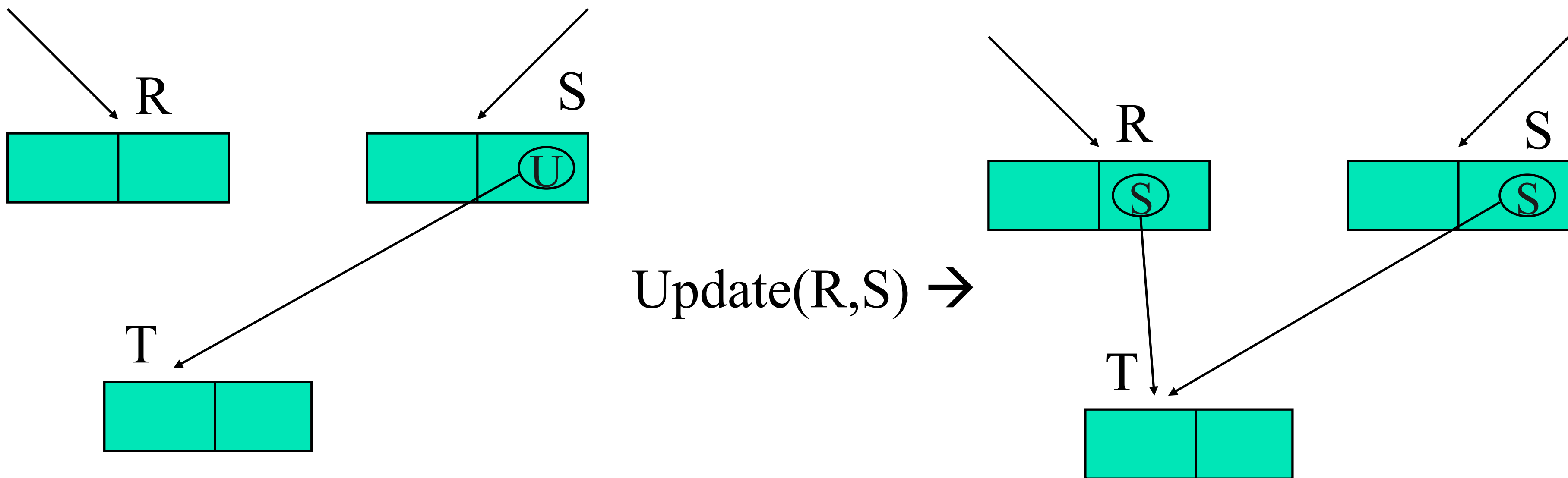
Using Short RC Field

- Statistics: Most objects have unique references.
- [Wise and Friedman, 1977] Use one bit for the reference count.
- Simple implementation: use one bit for each object (unique/shared)
- An improved implementation: the bit is in the pointer [Stoe, 1984].
  When rc is incremented, we can change the rc without fetching the object itself.

Assigning a Pointer

```
Copy(R,S) {
        if RC(*S) == unique
            markShared(*S)
        delete(*R)
        *R = *S
}
```

Note that we do not touch T during this assignment.



Using a Software Cache

- Often, adjustments to rc's are temporary.
- E.g.: `for( p = head ; p ; p = p→next ) { .. }`
- Problem: for all objects in linked list, rc is raised to 2 and gets stuck.
- Solution: use a software cache.
- The most recent objects whose rc was raised to overflow the rc field are cached.
- If their rc is decremented before leaving the cache, they are removed from cache, otherwise, they may get stuck.

Deferred Reference Counting

- Problem: overhead on updating program variables (locals) is too high.
- Solution [Deutch & Bobrow] :
  - Don't update rc for local pointers.
  - rc's reflect only references from heap objects. Not from stack.
  - We can't delete objects whose rc drops to zero.
  - Instead objects with rc = 0 are pushed to ZCT (Zero Count Table).
  - "Once in a while": collect all objects with o.rc=0 that are not referenced from local roots.

Deferred Reference Counting

```
delete(N)=
   decrementRC(N)
   If (N.rc) == 0
      add N to ZCT
```

```
update(R,S)=    //only for heap pointers
   incrementRC(S)
   delete(*R)
   remove S from ZCT  // (is this useful?)
   *R = S
```

```
collect()=
   For N in roots              // note roots
      incrementRC(N)
   for N in ZCT                // reclaim garbage
      If (N.rc) == 0
         for M in children(N)
            delete(*M)
         reclaim(N)
   for N in stack              // restore roots
      decrementRC(N)
```

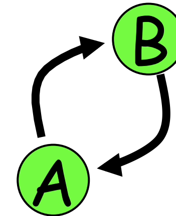
Deferred RC Properties

- Advantages:
  - Deferred RC reduces overhead by 80%.
- Disadvantages:
  - Immediacy of collection lost !
  - Space overhead for ZCT.

- Used in most modern RC systems.

Why wasn't RC Used ?

- Does not reclaim cycles



- A heavy overhead on pointer modifications (even with deferred RC).
- Traditional belief: "Cannot be used efficiently with parallel processing"
  - Lock or "compare & swap" for each pointer update.

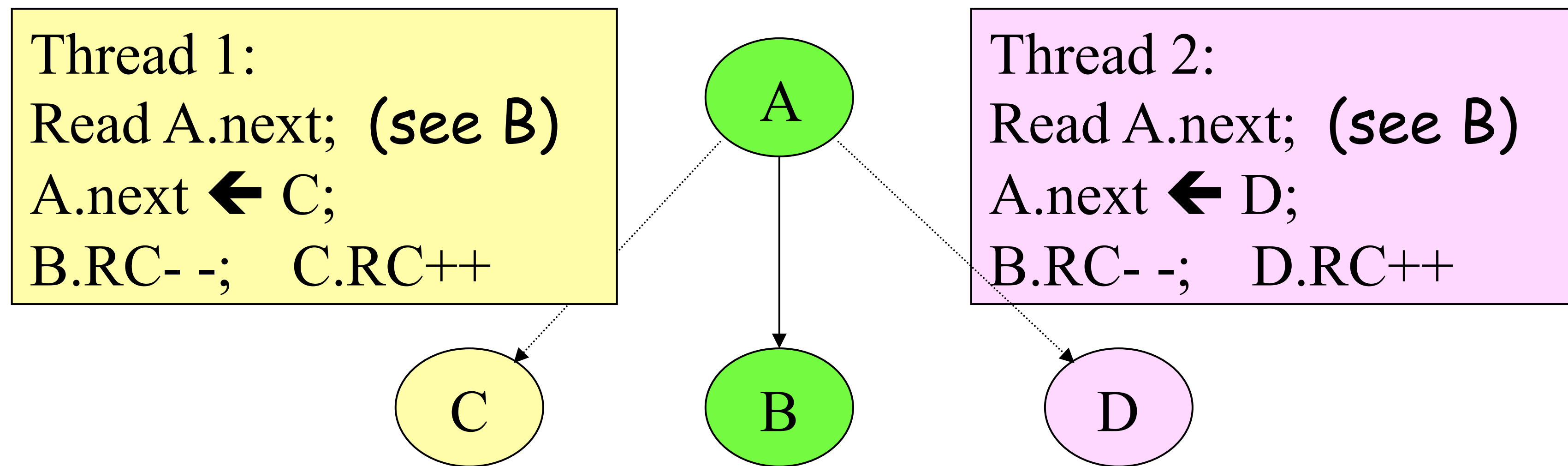
Multithreaded RC?

- Problem 1: ref-counts updates must be atomic

- Fortunately, this can be easily solved :
  Each thread logs required updates in a local buffer and the collector applies all the updates during GC (as a single thread).

Multithreaded RC?

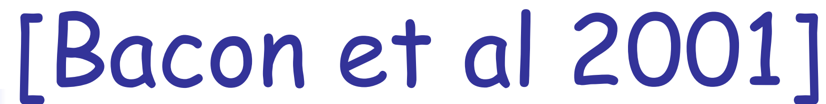
- Problem 1: ref-counts updates must be atomic.
- Problem 2: parallel updates confuse counters:



Multithreaded RC: First Attempt [DeTreville 1990]:

- Lock heap for each (heap) pointer modification.
  - Mutator knows which counter to update.
  - Mutator logs information to a global buffer.
- Mutators do not touch the counters
  - Each mutator records its updates in the global buffer.
  - Reference counts are updated by the collector.
  - This is meant to make it easy for the mutators while the collector works concurrently.

- The Collection Operation (snapshot alike):
  - GC thread reads the global buffer and roots to update ref counts up to a specific point.
  - Reclaims objects with rc 0 and not local at that point.



[Bacon et al 2001]

- Defer the decrements by one cycle. This allows:
  - On-the-fly collection, as objects are prudently reclaimed.
  - Getting rid of the Zero-Count-Table (at the cost of keeping a decrement list).
- A single compare-and-swap allows determining which objects' counts need updates.
  - Logging to local buffers.
  - Reference counts updated by collector.
- Advantages: short pauses, novel cycle collector.
- Disadvantages: sync in write barrier (throughput), floating garbage (heap consumption).

Efficient On-the-Fly Reference Counting Garbage Collector

Levanoni and Petrank

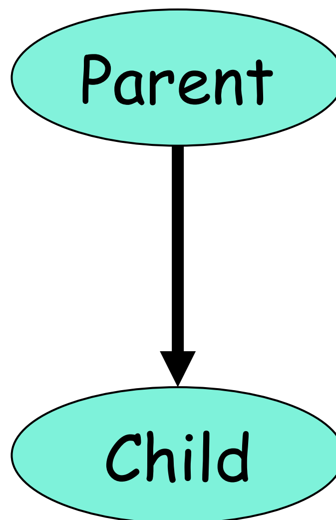
Goal

- Ameliorate two major problems of reference counting:
  - High write barrier overhead, and
  - Required sync in write barrier to make RC concurrent.
- Obtain: a new RC collector
  - Light write barrier.
    - In particular, no sync. operation required.
  - On-the-fly, good for a multiprocessor.

© Erez Petrank

Reducing RC Overhead:

- We start by looking at the "parent's point of view".
- We are counting rc for the child, but rc changes when a parent's pointer is modified.

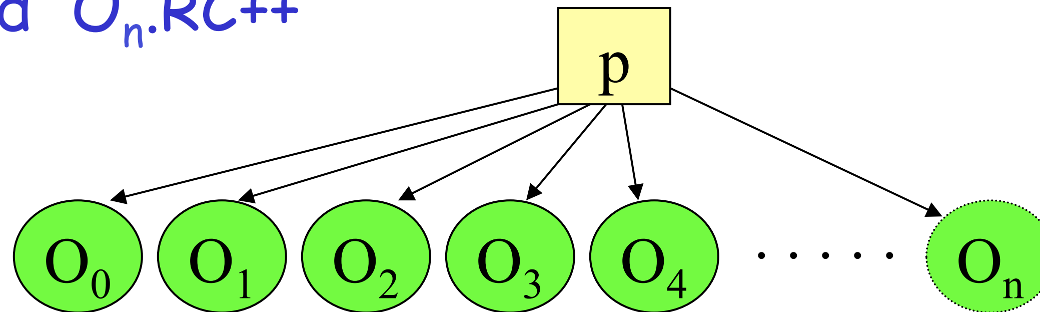


Improving RC Overhead:
An Observation

- Consider a pointer *p* that takes the following values between GC's: $O_0, O_1, O_2, \ldots, O_n$.
- All RC algorithms perform 2n operations:

  $O_0.RC--; O_1.RC++; O_1.RC--; O_2.RC++; O_2.RC--; \ldots ; O_n.RC++;$
- But only two operations are needed:

  O_0.RC-- and O_n.RC++



Updating rc’s Between Two Snapshots

- Suppose rc’s are updated for previous snapshot.
- To get all rc’s updated for the current snapshot, we need to worry only about modified pointers !
- For each modified pointer p, we need to decrement its previous descendant and increment its current descendant.

Use of Observation

Time

Garbage Collection

$P \leftarrow O_1$; (record p's previous value O_0)

$P \leftarrow O_2$; (do nothing)

…

$P \leftarrow O_n$; (do nothing)

Garbage Collection: For each modified slot p:

- Read p to get O_n, read records to get O_0.
- O_0.rc-- , O_n.rc++

Only the first modification of each pointer is logged.

Some Technical Remarks

- When a pointer is first modified, it is marked dirty and its previous value is logged.
- We actually log each *object* that gets modified (and not just a single pointer).
  - Reason 1: we don't want a dirty bit per pointer.
  - Reason 2: object's pointers tend to be modified together.
- Only non-null pointer fields are logged.
- New objects are "born dirty".

Effects of Optimization

- RC work significantly reduced:
  - The number of logging & counter updates is reduced by a factor of 100-1000 for typical Java benchmarks !

Elimination of RC Updates

Benchmark	No of stores	No of “first” stores	Ratio of “first” stores
jbb	71,011,357	264,115	1/269
Compress	64,905	51	1/1273
Db	33,124,780	30,696	1/1079
Jack	135,174,775	1,546	1/87435
Javac	22,042,028	535,296	1/41
Jess	26,258,107	27,333	1/961
Mpegaudio	5,517,795	51	1/108192

Effects of Optimization

- RC work significantly reduced:
  - The number of logging & counter updates is reduced by a factor of 100-1000 for typical Java benchmarks !
- Write barrier overhead dramatically reduced.
  - The vast majority of the write barriers run a single "if".
- Last but not least: the task has changed ! We need to record the first update.

Reducing Synch. Overhead

The second issue:

- Using the write barrier of previous lecture, we do not need any sync. operation.

- Recall that previous solutions required sync overhead (lock or compare-and-swap).

The Write Barrier

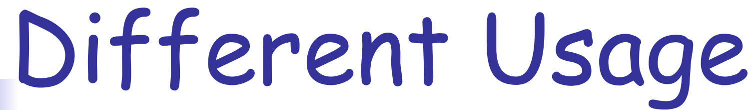
```
Update( ptr, obj ){
   saved = copy(O's pointers)
   if (O is not dirty) {
      log( saved )
      SetDirty(O)
   }
   ptr = obj
}
```

<u>Observation:</u>
If two threads:
1. invoke the write barrier in parallel, and
2. both log an old value,

then both record the same old value.
(Discussed in previous lecture.)

Different Goal, Same Technique

- For tracing, we needed to record previous values of all modified pointers, so we can obtained their values during the snapshot.
- For RC, we need to record all modified pointers, so that we can update rc's:
  - decrement rc for value at previous snapshot (O_0),
  - increment rc for value at current snapshot (O_n).



Different Usage

- For tracing, we needed to know if something changes during the trace.
- For RC, we need to keep write barrier running all the time.
  - We must know all first modifications from previous snapshot to this snapshot.
  - At the current snapshot, we steal buffers from mutators and give them new ones to record changes for the next collection.

Timeline

Snapshot collection i:
Take buffers
Clear dirty bits
Update rc's
collect

Between collections:
Record all first modifications in local buffers

Snapshot collection i+1:
Take buffers
Clear dirty bits
Update rc's
collect

© Erez Petrank

Timeline: Zoom on One Collection

Stop threads.

Resume threads.

Scan roots; Get buffers; erase dirty bits;

Decrement values in read buffers;

Increment "current" values;

Collect dead objects

© Erez Petrank

Timeline

Unmodified current values are in the heap. Modified are in new buffers.

Stop threads.

Resume threads.

Scan roots; Get buffers; erase dirty bits;

Decrement values in read buffers;

Increment "current" values;

Collect dead objects

```
Update( ptr, obj ){
   if (O is not dirty) {
      saved = copy(O)
      if (O is not dirty) {
         log( saved )
         SetDirty(O)
      }
   }
   ptr = obj
}
```

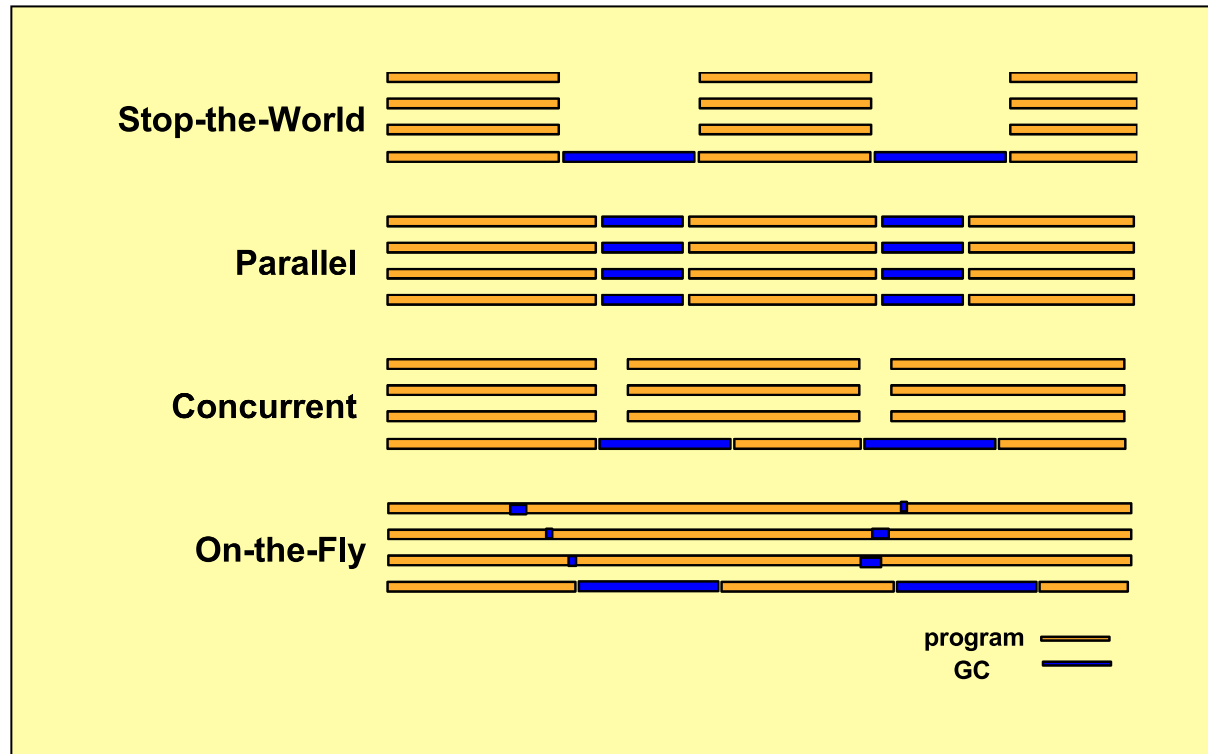
<u>Fact:</u> most updates are to dirty objects.

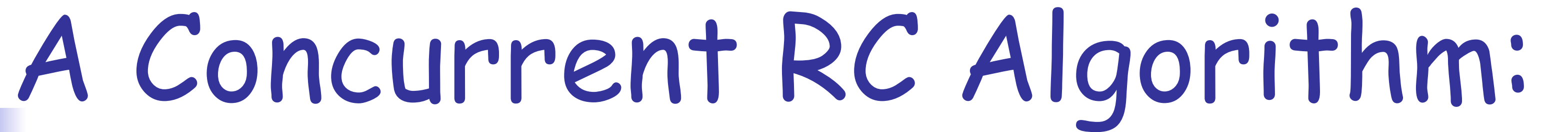
<u>Thus</u>, the “real” write barrier runs an optimistic initial “if”

Technical Issues

- Technical issues discussed in previous talk are still relevant.
- Each thread uses a local buffer to log in.
- Collector's may "look" at buffers or "steal" them.
- To find a buffer entry of a dirty object: the dirty "flag" is actually a pointer. For each object it points to the buffer entry (null means non-dirty).

Recall Terminology





A Concurrent RC Algorithm:

- Use write barrier with program threads.

- Create a snapshot:
  - Stop all threads
  - Scan roots (locals)
  - get the buffers with modified slots
  - Clear all dirty bits.
  - Resume threads

- Then run collector:
  - For each modified slot:
    - decrease rc for previous snapshot value (read buffer),
    - increase rc for current snapshot value ("read heap"),
  - Reclaim non-local objects with rc 0.

Correctness

- Liveness: if an object is unreachable when mutators are halted, and is not part of a cycle, then it has rc 0.
- Safety: follows from the snapshot paradigm
  - **Lemma**: all reachable objects at the snapshot must have rc>0 or be directly referenced by the roots.
  - **Fact**: Unreachable objects remain unreachable until collector reclaims them.
  - **Corollary**: safety.

Concurrent RC Algorithm: Properties

- Snapshot oriented, concurrent, (not so bad…)

Pause time:

- Stop all threads
- clean all dirty bits.
- mark roots of all threads.

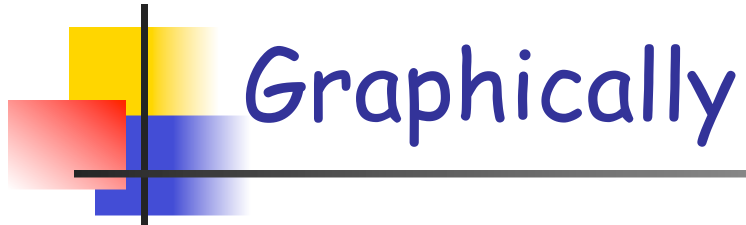
Desired pause time:

- Stop one thread to mark its own local roots !

- To achieve short pause times use sliding views.

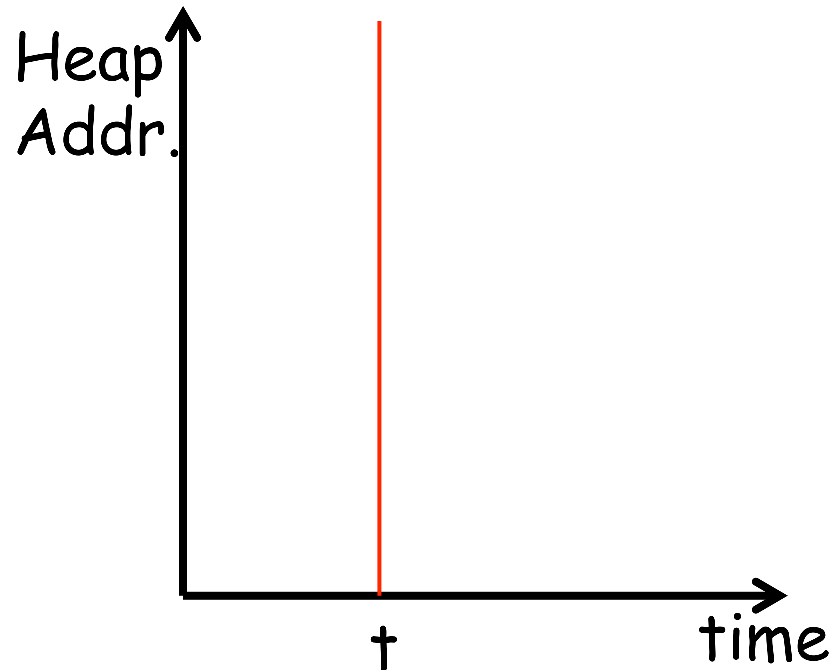
Reminder: What is a Sliding View?

- A snapshot of the heap at time t is a copy of the content of each object in the heap at time t.
- A sliding view of the heap at time interval [t1,t2] is a copy of the content of each object in the heap, each object O's content is copied at time t(O), satisfying
  $t1 \leq t(O) \leq t2$.

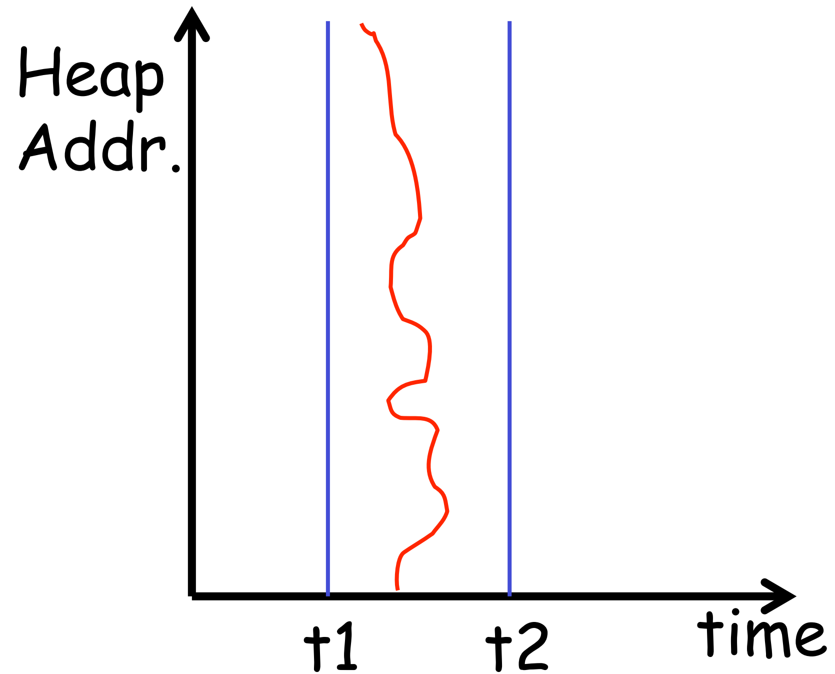


Graphically

Snapshot



Sliding Views



© Erez Petrank

Using a Sliding View to Collect

- The problem: no "consistent" (snapshot) view of the heap.
- The danger: reachability of objects may be missed!

Danger in Sliding Views

Program does:
O1.p ← O
O2.p ← O
O1.p ← NULL

Here sliding view reads O2 (p NULL)

Here sliding view reads O1 (p NULL)

Problem: reachability of O not noticed!

Can we miss a pointer to O?

- As explained in previous lecture, we can ensure not missing a live object if we use the snooping mechanism.
- During [t1,t2] we record the target of each modified pointer.
- These snooped objects (and their descendants) are not reclaimed. (Technically, they become roots.)
- This way, a sliding view can be used to reclaim objects.

Write Barrier with Snooping

```
Update( ptr, obj ){
   saved = copy(O)
   if (O is not dirty) {
      log( saved )
      SetDirty(O)
   }
   ptr = obj
   if (snooping)
      snoop(obj)
}
```

A Sliding-Views RC Algorithm:

- Use write barrier with program threads.

- <u>Create a sliding view</u>:
  - Stop each thread to initiate snooping.
  - Stop each thread to steal buffers with modified slots
  - Clear all dirty bits.
  - Stop each thread to scan roots (locals)
  - Stop snooping, and add snooped to roots.

- <u>Then run collector:</u>
  - For each modified slot:
    - decrease rc for previous snapshot value (read buffer),
    - increase rc for current snapshot value ("read heap"),
  - Reclaim non-local objects with rc 0.

© Erez Petrank

Updating the Reference Counts

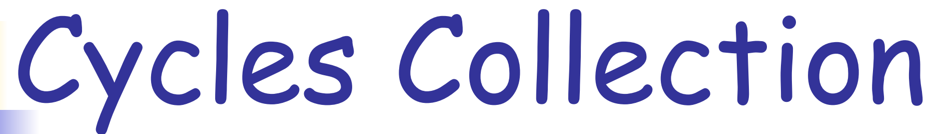
- Do the reference counts reflect a consistent view of the heap pointers at any point in time?
- **No!!!**
  rc's correspond to the heap pointers as reflected by a sliding view.

- This is counter-intuitive
  - Seems like we need to update reference counts "properly"
- But what matters is that we may use these reference counts to reclaim unreachable objects.

Issues skipped

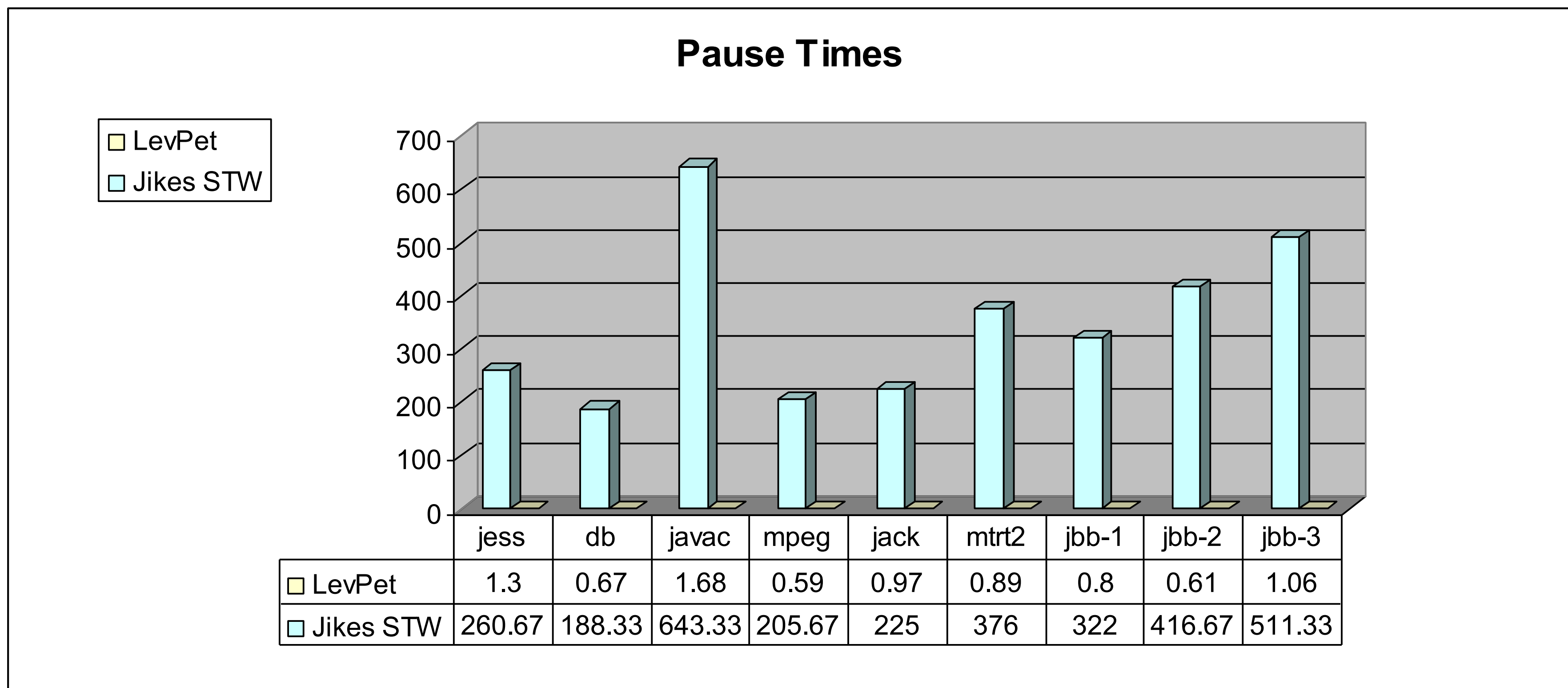
- Clearing of the dirty bits is executed while write-barrier operates. That requires some care.
- Double logging should be consolidated.
  - Here, if there are two entries, then rc's may be updated twice. We must use only one log for each modified object (it is not enough that the logs are equal, they must be unique).
  - Easily solved if the dirty bit is a "dirty pointer".
- Recording roots with no simultaneous halt.
- Correctness discussion.

Cycles Collection



- Use a tracing algorithm infrequently.
- The sliding-views mark and sweep algorithm has the same write barrier so it may be easily used infrequently.
- Usage of cycle collection algorithms is also possible.
- Cycle collection is the next topic (for next week).

Pause Times vs. STW



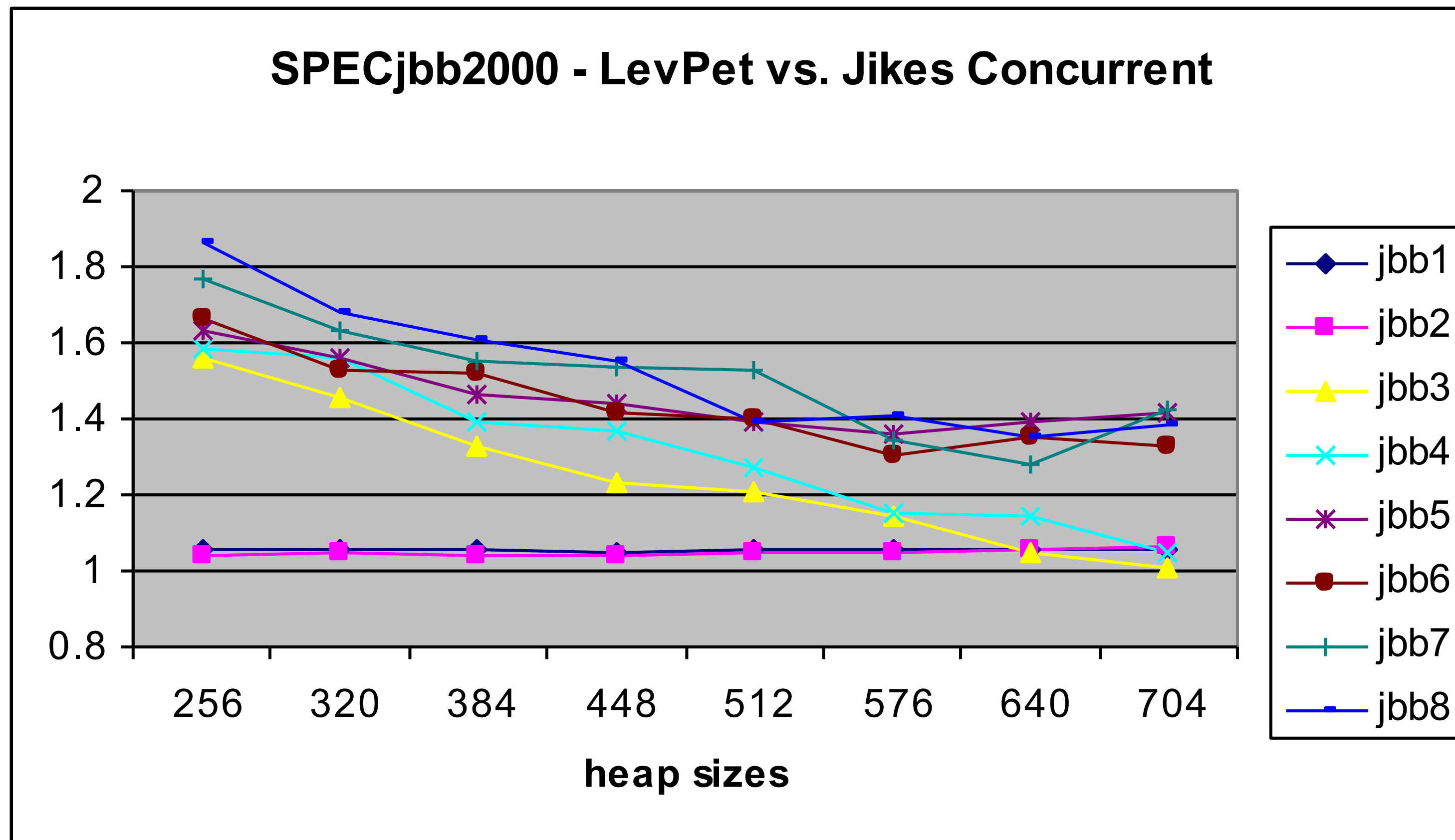
	jess	db	javac	mpeg	jack	mtrt2	jbb-1	jbb-2	jbb-3
LevPet	1.3	0.67	1.68	0.59	0.97	0.89	0.8	0.61	1.06
Jikes STW	260.67	188.33	643.33	205.67	225	376	322	416.67	511.33

Pause Times vs. Jikes Concurrent

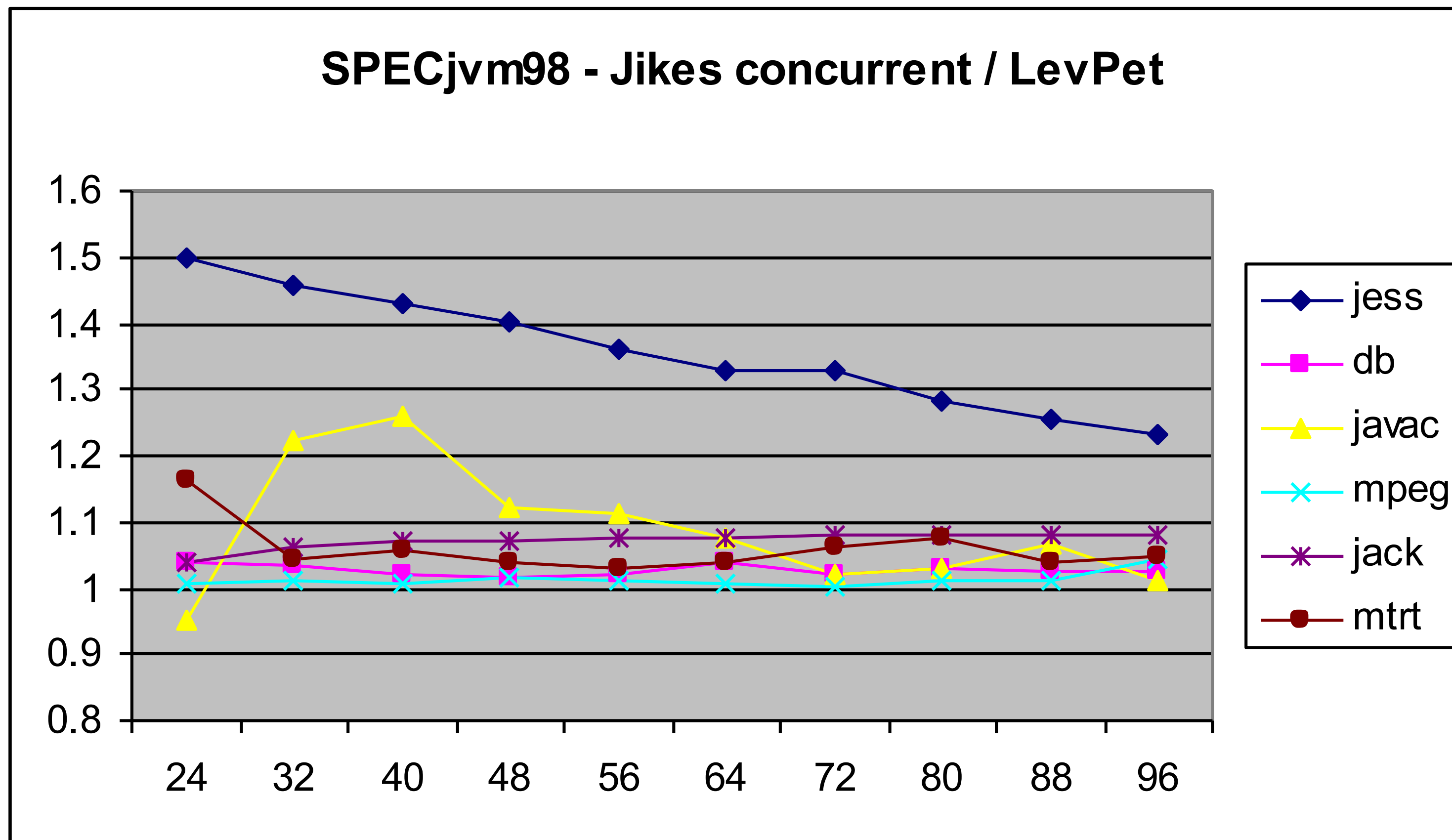
Pause Times - Concurrent

	jess	db	javac	mpeg	jack	mtrt2	jbb-1	jbb-2	jbb-3
Jikes Concurrent	2.77	1.84	2.81	0.8	1.66	1.8	1.79	2.6	3.15
LevPet	1.3	0.67	1.68	0.59	0.97	0.89	0.8	0.61	1.06

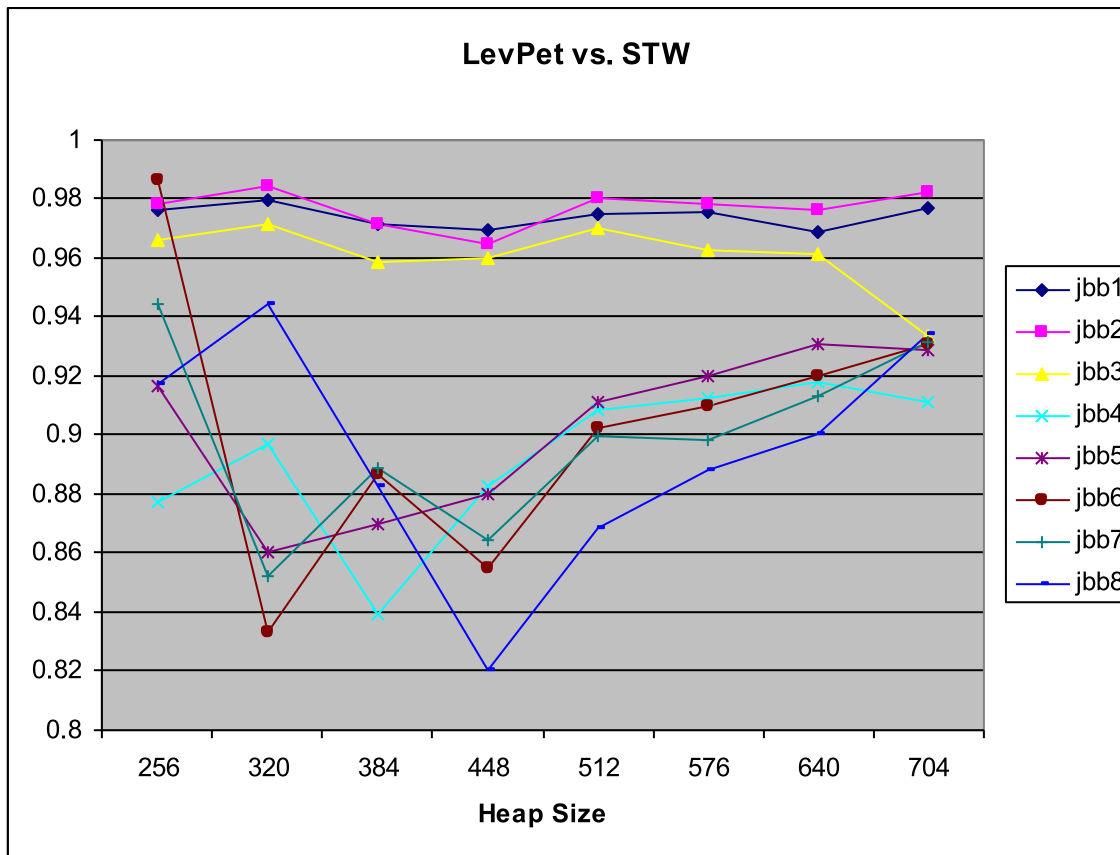
SPECjbb2000 Throughput



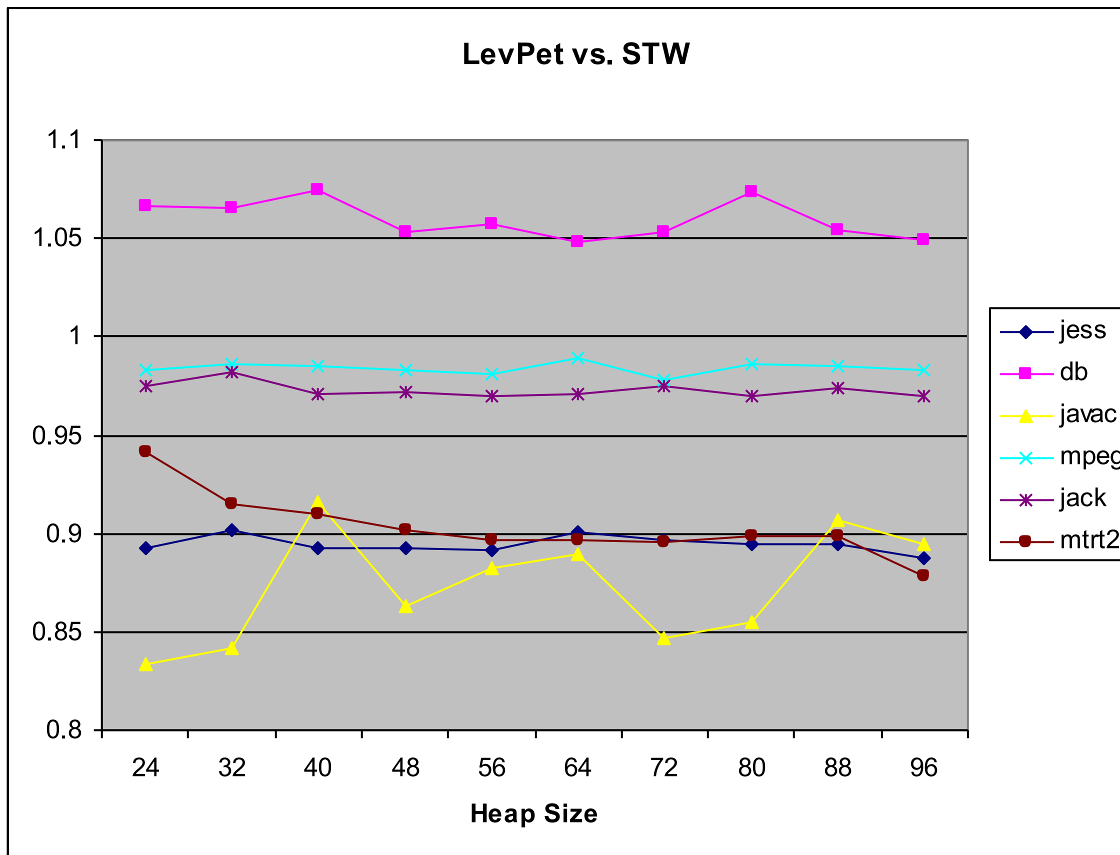
SPECjvm98 Throughput



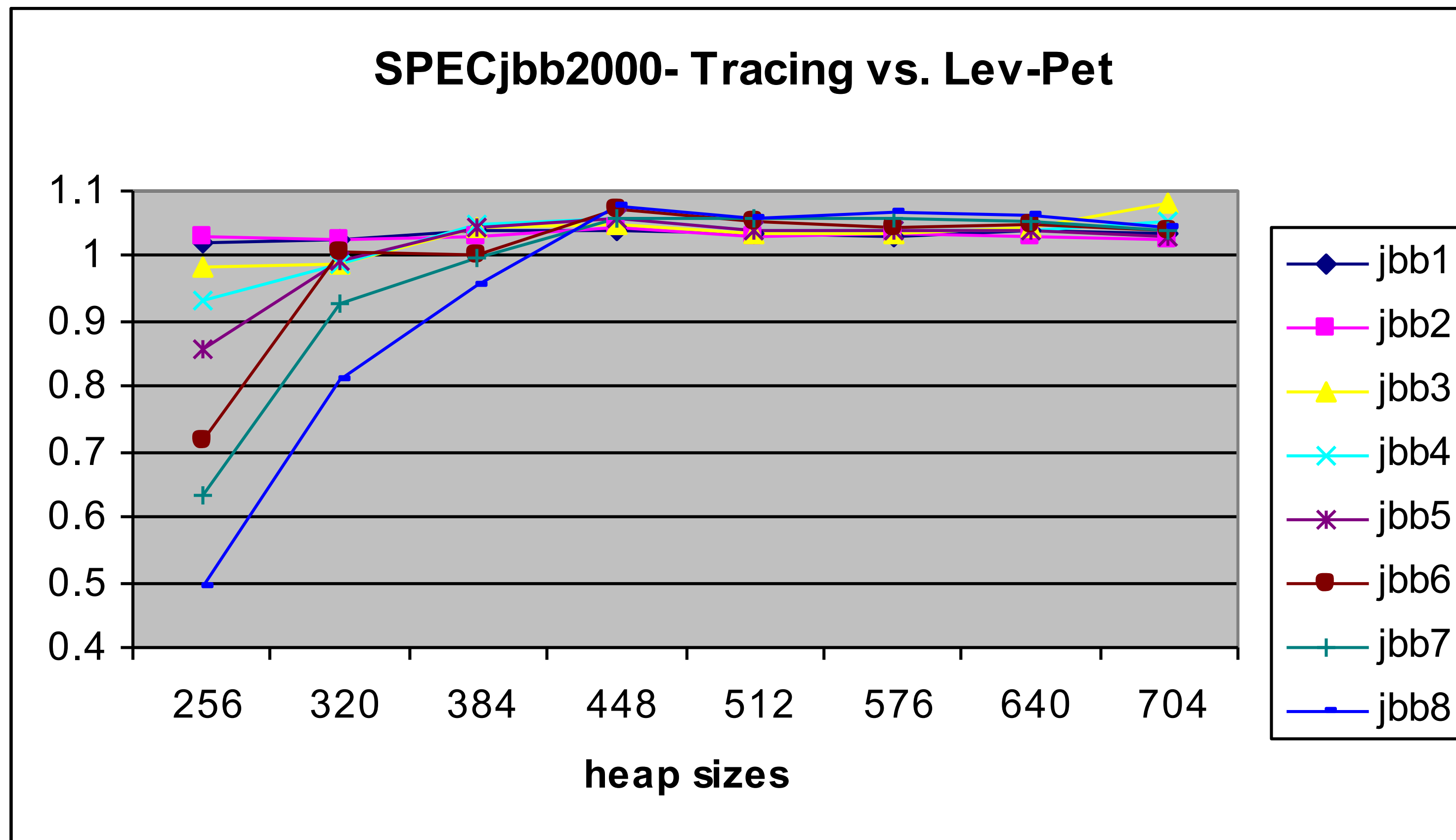
SPECjbb2000 Throughput



SPECjvm98 Throughput

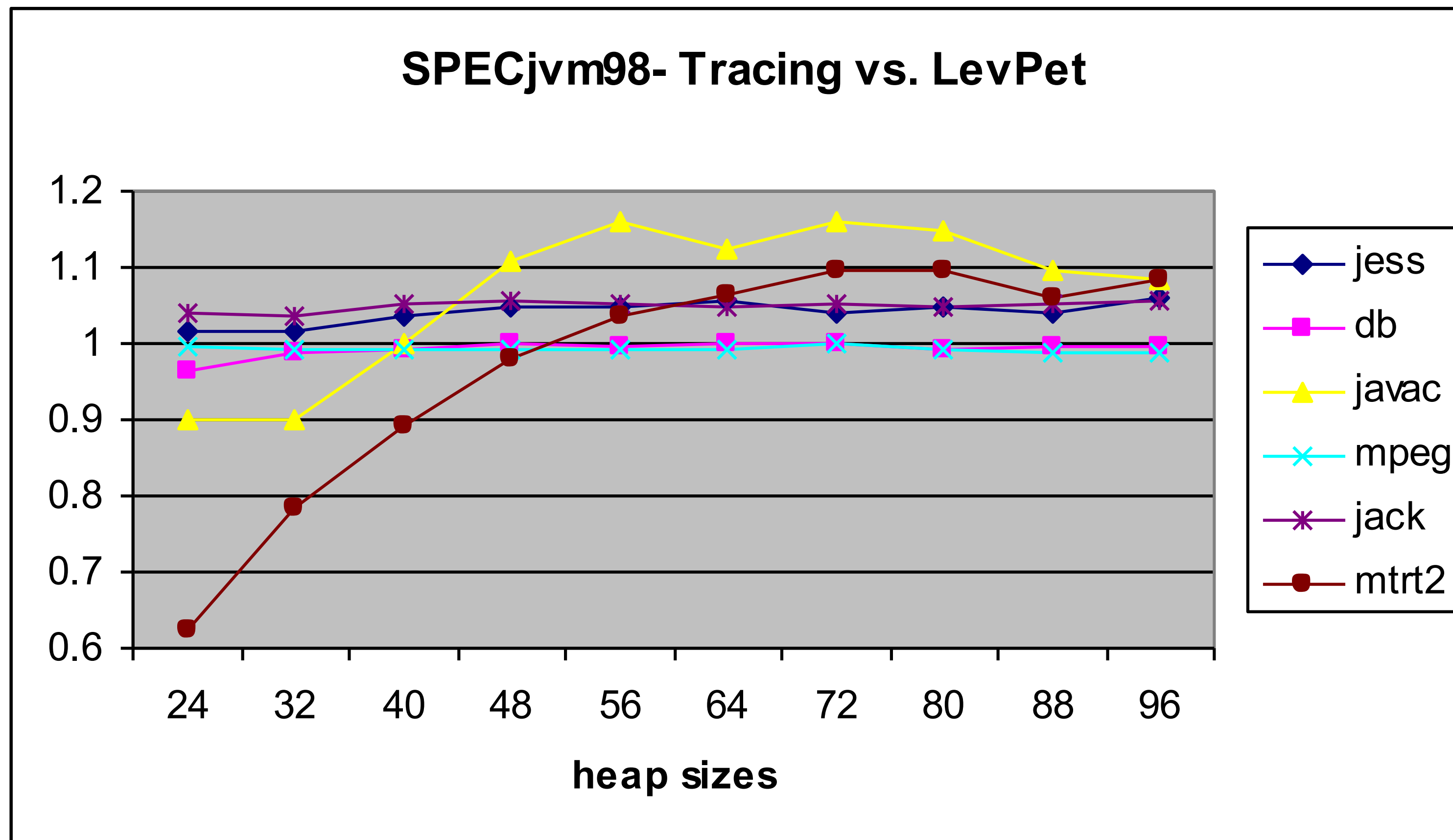


SPECjbb2000 Throughput



© Erez Petrank

SPECjvm98 Throughput



Levanoni-Petrank Properties

- Light write barrier (like tracing).
- Adequate for parallel processing
- Low pauses: only for scanning the thread's local roots.

- The problem: on-the-fly version is complicated.