Algorithms for Dynamic Memory Management (236780)

Lecture 9

Lecturer: Erez Petrank
Topic last week

• Allocation techniques
• Hoard (by Emery)
Topics this week (extended)

• The Compressor:
  • Parallel version
  • Concurrent version
• Reference counting
No Class Next Week!

We meet again on June 24
The Compressor
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.
The Compressor

• [Kermany-Petrunk 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 table providing a bit for each beginning of heap object, and each end.

• **Mark-bit vector allows computing a target address for each object, single pass.**
Map Old to New Address

- 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.

<table>
<thead>
<tr>
<th>0</th>
<th>50</th>
<th>100</th>
<th>125</th>
<th>200</th>
<th>275</th>
<th>325</th>
<th>350</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1100</td>
<td>1200</td>
<td>1300</td>
<td>1400</td>
<td>1500</td>
<td>1600</td>
<td>1700</td>
</tr>
</tbody>
</table>
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: $\text{Fix-up}(1375) = 1000 + 125 + 50 = 1175$.
- (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.
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.
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.
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.
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.
A Major Issue with Concurrency

• If we move objects while application is running, two copies appear.
  – Difficult to synchronize all threads to properly switch between using one copy to 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: $\text{Fix-up}(1525) = 1000+275+0 = 1275 \,(=1525-250)$.
• Object 8: $\text{Fix-up}(1550) = 1000+275+25 = 1300 \,(=1550-250)$.
• 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

<table>
<thead>
<tr>
<th>Number of WareHouses</th>
<th>Throughput (ops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7,000</td>
</tr>
<tr>
<td>2.75</td>
<td>8,833.333</td>
</tr>
<tr>
<td>4.5</td>
<td>10,666.667</td>
</tr>
<tr>
<td>6.25</td>
<td>12,500</td>
</tr>
<tr>
<td>8</td>
<td>14,333.333</td>
</tr>
<tr>
<td>10</td>
<td>16,166.667</td>
</tr>
<tr>
<td>12</td>
<td>18,000</td>
</tr>
</tbody>
</table>

![Graph showing throughput vs number of warehouses for different systems](image-url)
### Measurements - pause time

<table>
<thead>
<tr>
<th>generational Mark and Sweep (full collections)</th>
<th>Mark and Sweep</th>
<th>Parallel Compaction (Stop-The-World)</th>
</tr>
</thead>
<tbody>
<tr>
<td>279.73</td>
<td>229.37</td>
<td>319.55</td>
</tr>
<tr>
<td>323.64</td>
<td>287.32</td>
<td>516.89</td>
</tr>
<tr>
<td>347.42</td>
<td>315.71</td>
<td>641.53</td>
</tr>
<tr>
<td>374.41</td>
<td>372.46</td>
<td>770.14</td>
</tr>
</tbody>
</table>


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.
Reference Counting
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.
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 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.
Deficiencies of RC

- Initially (naïve algorithms):
  - Cost of removing last ptr to an object unbounded
  - Overall overhead is greater than that of tracing G.C.
  - 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
RC Engineering

- 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.
- But nice advances in 2000 and on.
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.
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 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 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 can determine counters 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.
Multithreaded RC: First Attempt [DeTreville 1990]:

- 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]

- Pointer modification using a single compare-and-swap.
  - Logging to local buffers.
  - Reference counts updated by collector.
- 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).
- **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.
Reducing RC Overhead:

- We start by looking at the “parent’s point of view”.
- RC maintains rc counting for the child, but rc changes when a parent’s pointer is modified.
Update Coalescing:

- 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:
  \( O_0.\text{RC}--; \ O_1.\text{RC}++; \ O_1.\text{RC}--; \ O_2.\text{RC}++; \ O_2.\text{RC}--; \ldots; \ O_n.\text{RC}++; \)
- But only two operations are needed:
  \( O_0.\text{RC}--; \text{ and } O_n.\text{RC}++; \)

![Diagram of pointer p and nodes O0, O1, O2, O3, O4, ..., On]
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

Garbage Collection

\[ P \leftarrow O_1; \text{ (record p's previous value } O_0) \]
\[ P \leftarrow O_2; \text{ (do nothing)} \]
\[ \ldots \]
\[ P \leftarrow O_n; \text{ (do nothing)} \]

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 Update Coalescing

- 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

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

• 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
}

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

Timeline

Stop threads.

Scan roots; Get buffers; erase dirty bits;

Decrement values in read buffers;
Increment "current" values;

Resume threads.

Collect dead objects

May 28, 2014
© Erez Petrank
Improving Efficiency

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

Fact: most updates are to dirty objects.
Thus, 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

Stop-the-World

Parallel

Concurrent

On-the-Fly

program GC
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 \( rc \leq 0 \).

- **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.
Pause Times vs. STW

Pause Times

<table>
<thead>
<tr>
<th>LevPet</th>
<th>Jikes STW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- jess
- db
- javac
- mpeg
- jack
- mtrt2
- jbb-1
- jbb-2
- jbb-3
Pause Times vs. Jikes Concurrent

Pause Times - Concurrent

- jess
- db
- javac
- mpeg
- jack
- mtrt2
- jbb-1
- jbb-2
- jbb-3
SPECjbb2000 Throughput

![SPECjbb2000 - LevPet vs. Jikes Concurrent](image)

**SPECjbb2000 - LevPet vs. Jikes Concurrent**

- jbb1
- jbb2
- jbb3
- jbb4
- jbb5
- jbb6
- jbb7
- jbb8

heap sizes

- 256
- 320
- 384
- 448
- 512
- 576
- 640
- 704
SPECjvm98 Throughput

SPECjvm98 - Jikes concurrent / LevPet

- jess
- db
- javac
- mpeg
- jack
- mtrt
SPECjbb2000 Throughput

LevPet vs. STW

Heap Size

jbb1  jbb2  jbb3  jbb4  jbb5  jbb6  jbb7  jbb8
SPECjvm98 Throughput

LevPet vs. STW

Heap Size

jess/db/javac/mpeg/jack/mtrt2
SPECjbb2000 Throughput

SPECjbb2000 - Tracing vs. Lev-Pet

heap sizes

256 320 384 448 512 576 640 704

jbb1 jbb2 jbb3 jbb4 jbb5 jbb6 jbb7 jbb8
SPECjvm98 Throughput

![Graph showing SPECjvm98 Tracing vs. LevPet with different heap sizes and application benchmarks like jess, db, javac, mpeg, jack, and mtrt2.]
Levanonie-Petranks RC Properties

- Light write barrier (like tracing).
- Adequate for parallel processing.
- On-the-fly implies Low pauses.
  - only for scanning the thread’s local roots.

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