Algorithms for Dynamic Memory Management (236780)

Lecture 7

Lecturer: Erez Petranks
Topics Last Week

- DLG
- Mostly Concurrent Collection: IBM’s view.
Today:
concurrent (not mostly) collection

• Snapshot concurrent collection:
  • [Demers-Weiser-Hayes-Boehm-Bobrow-Shenker 1990]
  • [Furusou-Matsuoka-Yonezawa 1991]

• Sliding Views:
  • [Azatchi-Levanoni-Paz-Petrank 2003] (following [Levanoni-Petrank 2001]).
Recall Terminology
(stop the world, parallel, concurrent, ...)
Snapshot Copy-On-Write
A snapshot (concurrent) collection

- A naïve collector:
  - Stop program threads
  - Create a snapshot (replica) of the heap
  - Program threads resume
  - Trace replica concurrently with program
  - Objects identified as unreachable in the replica may be collected in the real heap.

Main idea: no one touches unreachable objects except the collector
A snapshot (concurrent) collection

- A naïve collector:
  - Stop program threads
  - Create a snapshot (replica) of the heap
  - Program threads resume
  - Trace replica concurrently with program
  - Objects identified as unreachable in the replica may be collected in the real heap.

Problem: taking a replica of the heap is not realistic
Using Locality

- A property of “typical” benchmarks:
  - Small parts of the heap are being modified at short time interval. (Locality.)
- Idea: it is enough to copy only the parts that are being modified.
  - The rest can be read from the heap.
- Problem: cannot tell in advance which areas will be modified by the program.
- Solution: copy each area just before it is modified.
Using Page Protection

- To begin the collection, all mutators are stopped and all memory pages are write-protected.
- When mutators update an object, the collector takes the trap on the protected page and creates a copy of the page for its use in the trace. It then clears the page protection.
- Copies may be “released” before sweep.
- Correctness: an unreachable object remains unreachable!
Advantages:

- A simple concurrent collector.
- No need for compiler support (write barrier).
Some inefficiencies

- Inefficiency: manual protection is usually inefficient.
  - Also, adding to page trap code…
- Copying applies to all fields although only pointers are relevant.
- Scalability: copying a page requires synchronization.
- Predictability: efficiency depends on the system.
- Time for writing is unstable.
- Programs work at object level, this mechanism at page level
  - a waste to copy a full page.
A Property of Some Operating Systems (e.g., UNIX Fork):

- Processes may request sharing memory pages.
- After the share, each process gets its own copy of the pages and any modifications done by one process are not visible to the other.
- A similar problem for operating system: should the memory be copied? If it is not modified, then it is not necessary.
- Some systems implement a “copy-on-write” strategy. Pages are shared until one of the processes modifies a page and then two copies are created.
GC with Copy-On-Write:

- To start GC: fork and share the heap.
- One process lets the program execute.
- Second process traverses the heap to mark live objects.
- When marking done, second process hands the mark table to the first process, which executes the sweep.
Snapshot copy-on-write

Properties:

- No need for compiler support (write barrier).
- System implemented copy-on-write is efficient.
  - No need to add page trap code…
- System dependent.
- Copy page is also triggered by non-pointer modifications.
- Time for writing is unstable.
- Overhead is not proportional: one integer modification causes a page copy.
Using Sliding Views

[Levanoni-Petrunk 2001]
[Azatchi-Levanoni-Paz-Petrunk 2003]

1. Object level logging
2. No synchronization in write barrier
3. On-the-fly
Object logging via a Write Barrier

Update( ptr, obj ){
    if (object O containing ptr is not dirty) {
        log( O’s pointers )
        SetDirty(O)
    }
    ptr = obj
}

A dirty bit per-object signifies if object has been copied already
Problem with Concurrency

Suppose T1 and T2 run this code concurrently. Both find object not dirty. T1 is quicker: logs old value, sets dirty bit, modifies O. T2 now does the same.

T1 & T2 log different values. Which of the logs is "right"?
Modifying the Write Barrier

Update( ptr, obj ) {  // Let O be object containing ptr
  if (O is not dirty) {
    log( O’s pointers )
    SetDirty(O)
  }
  ptr = obj
}
Nice Parallel Behavior

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.
Thread 1:

Update( ptr, obj ){
    saved = copy(O)
    if (O is not dirty) {
        /* if T1 got here, Thread 2 has */
        /* not set the dirty bit, thus, has */
        /* not yet modified the slot. */
        log( saved )
        SetDirty(O)
    }
    ptr = obj
}

Thread 2:

Update( ptr, obj ){
    saved = copy(O)
    if (O is not dirty) {
        /* if T2 got here, Thread 1 has */
        /* not set the dirty bit, thus, has */
        /* not yet modified the slot. */
        log( saved )
        SetDirty(O)
    }
    ptr = obj
}
Fact: most updates are to dirty objects. Thus, the “real” write barrier runs an optimistic initial “if”
Optimized Write Barrier

```
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”
**Logging Operation**

- Logging to a *global* buffer requires synch.
- Thus, each thread holds a *local* buffer.
  - Obtain a buffer from “free” global pool.
  - Return a full buffer to a “full” global pool.
  - Entry contains: (obj address, pointers contents)
- Collector’s access to buffers:
  - *look* at any local buffer, or
  - “*steal*” local buffer (get it and give the mutator a new one).
The Concurrent Collector

- Stop all program threads
- Scan roots (locals)
- Initiate write barrier usage
- Resume threads
- Trace from roots.
  - To trace a dirty object use local buffers to obtain its pointers.
- Stop write barrier usage
- Sweep to reclaim unmarked objects.
- Clear all buffers and dirty bits.

Note: no write barrier on locals, they are traced during the halt!
The Concurrent Collector

- Stop all program threads
- Scan roots (locals)
- Initiate write barrier usage
- Resume threads
- Trace from roots.
  - To trace a dirty object use local buffers to obtain its pointers.
- Stop write barrier usage
- Sweep to reclaim unmarked objects.
- Clear all buffers and dirty bits.

Snapshot time is here
Locating Entry of a Dirty Object

- **Option 1**: the dirty “flag” is actually a pointer. For each object it points to the buffer entry (null means non-dirty).

- **Option 2**: run in phases:
  1. Record dirty objects as gray (& don’t trace).
  2. When out of non-dirty objects, go over local buffers and trace logged gray objects.
  3. If new gray objects created during step 2, run step 2 again. (Optimizations possible.)

- Current implementation uses Option 1.
Reading the Snapshot value (simplified)

Trace(O) {
  if (O.color == white) then // if not marked
    shade(O)
  // Make a copy of object state at snapshot time.
  if (O.dirty != NULL) then // if object dirty
    copy := get-copy-from-buffer (O.dirty)
  else // object not dirty
    copy := get-copy-from-heap (O)
  if (O.dirty != NULL) then // object got dirty concurrently
    copy := get-copy-from-buffer (O.dirty)
  // Trace copy
  for each slot s of copy do
    Trace(s)
  O.color := black
}
Created Objects’ Color and Dirt

- **Color:**
  - black during the trace (they are alive and have no children during creation),
  - gray/white during sweep (as in DLG),
  - white when collector idle.
  - (Color toggle method possible and advisable.)

- **Dirt:** new objects are created dirty during the trace, non-dirty otherwise.
  - They will never be traced since they are black.
  - Dirty bit on lightens the write barrier (significantly).
Correctness

- **Liveness**: if an object is unreachable when mutators are halted, it will not be traced.
- **Safety**: follows from the snapshot paradigm
  - **Lemma**: all objects reachable during the snapshot are marked, and all objects created after the snapshot are marked.
  - **Fact**: Unreachable objects remain unreachable until collector reclaims them.
  - **Corollary**: safety.
From Concurrent to On-the-Fly

- Our next goal is to avoid simultaneous halting of all mutators.
- During the halt we scan roots and initiate write barrier.
- With no single synchronization point, we will not have a snapshot view of the heap.
- Instead, we will get a “sliding view”.
Plan

- What is a sliding view of the heap?
- How can a sliding view be used to collect unreachable objects?
- Why does the above algorithm become a sliding-views collector when mutators are not halted simultaneously?
What is a Sliding View?

- Recall: 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 \([t_1,t_2]\) is a copy of the content of each object in the heap, each object \( O \)'s content is copied at time \( t(O) \), satisfying \( t_1 \leq t(O) \leq t_2 \).
Graphically

Snapshot

Heap Addr.

Heap Addr.

Sliding Views

Heap Addr.

© 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!
Understanding Sliding Views

- What’s wrong with the sliding view?
  - The view of the heap is not a snapshot. Intuitively: “non-consistent” view of the heap.
- How different is a sliding view at \([t1,t2]\) from a snapshot at time \(t2\)?
  - Objects that are not modified in the interval \([t1,t2]\) (most objects) look the same.
  - Objects that are modified can appear in any of their states during the interval.
- Let’s concentrate on what we need…
Can we miss a pointer to O?

- Consider the set of objects pointing to O at t2: \((A_1, A_2, \ldots, A_n)\).
- 4 possibilities:
  - **Case 1**: Pointers to O are deleted and added during \([t_1,t_2]\).
  - **Case 2**: Pointers to O are only added (but not deleted) during \([t_1,t_2]\).
  - **Case 3**: Pointers to O are only deleted (but not added) during \([t_1,t_2]\).
  - **Case 4**: Pointers to O are not modified during \([t_1,t_2]\).
Can we miss a pointer to O?

- 4 possibilities:
  - Case 1: Pointers to O deleted & added during [t1,t2].
  - Case 2: Pointers to O added during [t1,t2].
  - Case 3: Pointers to O deleted during [t1,t2].
  - Case 4: Pointers to O not modified during [t1,t2].

- Case 4 is easy: S.V. at [t1,t2] sees the same pointers that a snapshot at t2 sees.

- Case 3 is not a problem. S.V. at [t1,t2] will see more pointers to O than snapshot at t2.
  - There were more pointers to O at t1; some were deleted.
Can we miss a pointer to O?

- 4 possibilities:
  - Case 1: Pointers to O deleted & added during [t1,t2].
  - Case 2: Pointers to O added during [t1,t2].
  - Case 3: Pointers to O deleted during [t1,t2].
  - Case 4: Pointers to O not modified during [t1,t2].

- Cases 1 and 2 pose a problem. S.V. at [t1,t2] sees only a subset of the pointers to O that a snapshot at t2 sees.
  - In particular, it may see no pointers to O
Solution

- The two problematic possibilities:
  - Case 1: pointers to O deleted and added during [t1,t2].
  - Case 2: pointers to O added during [t1,t2].
- In both cases a pointer to O is added.
- To make sure no reachable object is reclaimed we run a snooping mechanism.
- During [t1,t2] we record the new 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)
}

Snooping Properties

- **Gain:**
  - We may use a sliding view at \([t1,t2]\) to safely reclaim garbage.

- **Pay:**
  - Added code to the write barrier.
  - Buffers take space, especially if large.
  - Added floating garbage: objects unreachable at \(t2\) may be marked and not reclaimed.
  - However, each snooped object is alive at \(t1\). Also, if \([t1,t2]\) is short, a small number of objects is snooped.

- **Correctness:**
  - We trace a superset of the pointers of snapshot \(t2\).
First Step: Initiation of Write Barrier

- For each thread:
  - Stop thread and initiate snooping.
- For each thread:
  - Stop thread and initiate write barrier usage.
- Stop all threads
- Scan roots (locals)
- Resume threads
- Stop snooping
- Trace from roots and snooped objects.
  - Whenever a dirty object is discovered use buffers to obtain its pointers.
- Stop write barrier usage
- Sweep to reclaim unmarked objects.
- Clear all buffers and dirty bits.

Originally:
Stop all program threads
Scan roots (locals)
Initiate write barrier usage
Resume threads
First Step: Initiation of Write Barrier

- For each thread:
  - Stop thread and initiate snooping.
- For each thread:
  - Stop thread and initiate write barrier usage.
- Stop all threads
- Scan roots (locals)
- Resume threads
- Stop snooping
- Trace from roots and snooped objects.
  - Whenever a dirty object is discovered, use buffers to obtain its pointers.
- Stop write barrier usage
- Sweep to reclaim unmarked objects.
- Clear all buffers and dirty bits.

This implicitly creates a sliding view!

Each object is traced according to its state between t1 and t2.

$t1$ is here

$t2$ is here
A Sliding View is Implicitly Used

- Three cases for each traced object. (Say, object O is traced at time t3.)
  - Case 1: not modified between t1 and t3. We think of it as being read by the s.v. at time t2.
  - Case 2: modified and logged between t1 and t3. We think of it as being read by the s.v. at: time = minimum { t2, time of logging} .
  - Case 3: modified but not logged between t1 and t3. (This means that it was not modified after t2.) We think of it as being read by the s.v. at t2.
First Step: Initiation of Write Barrier

- For each thread:
  - Stop thread and initiate snooping.
- For each thread:
  - Stop thread and initiate write barrier usage.
- Stop all threads
- Scan roots (locals)
- Resume threads
- Stop snooping
- Trace from roots and snooped objects.
  - Whenever a dirty objects is discovered use buffers to obtain its pointers.
- Stop write barrier usage
- Sweep to reclaim unmarked objects.
- Clear all buffers and dirty bits.
First Step: Initiation of Write Barrier

- For each thread:
  - Stop thread and initiate snooping.
- For each thread:
  - Stop thread and initiate write barrier usage.
- Stop all threads
- Scan roots (locals)
- Resume threads
- Stop snooping
- Trace from roots and snooped objects.
  - Whenever a dirty objects is discovered use buffers to obtain its pointers.
- Stop write barrier usage
- Sweep to reclaim unmarked objects.
- Clear all buffers and dirty bits.

Crucial that all start snooping before t1. Hence, the handshake.

No handshake required to stop snooping after t2.
Scanning the roots separately

- **Next goal**: scan roots of each thread separately.
- **Problem**: no write-barrier on roots (threads’ stacks)
- Therefore, pointers added to the roots are not snooped.
- Can we make sure all root pointers are traced?
- **Observation 1**: no safety for on-the-fly collectors if pointers may move directly from one thread’s stack to another.
- **Reason**: after scanning the roots of T1, move a pointer to O from T2 to T1. Object O may be missed by the collector.
- This is not possible in Java. Possible in C, C++, but considered “inappropriate”.
Scanning the roots separately

- **Next goal**: scan roots of each thread separately.
- **Assumption**: pointers cannot be moved directly from one thread’s stack to another.
- They have to be written into the heap (or to global variables) before moving from one stack to another.
- Will we trace all root pointers at time t2?
The On-The-Fly Collector

- For each thread:
  - Stop thread and initiate snooping.
- For each thread:
  - Stop thread and initiate write barrier usage.
- For each thread:
  - Stop thread and scan its roots.
- Stop snooping
- Trace from roots and snooped objects.
  - Whenever a dirty objects is discovered use buffers to obtain its pointers.
- Stop write barrier usage
- Sweep to reclaim unmarked objects.
- Clear all buffers and dirty bits.

Originally:
- Stop all program threads
- Scan roots (locals)
- Initiate write barrier usage
- Resume threads
Scanning the roots separately

- t1: write barrier starts (snooping already running).
- t2’: all threads executing write barrier
- t2: all roots have been scanned
- (Now snooping turns off)

Assumption: pointers cannot be moved directly from one thread’s stack to another.
- Heap pointers scanned a *superset* of snapshot at t2’ (or t2’)
- If we miss a root pointer p of t2, then it was added to the roots.
- From where? From the heap…
  - On the heap at t2’ (, or
  - written to the heap between [t2’,t2] and then copied to roots.
Optimizing the write barrier

- We only need to store:
  1. the object once.
  2. non-null pointer values of object.
  3. while tracing is on.
  4. objects that have not been traced.

- Implication of 4: new objects are never stored.

Slow path of the write barrier (object needs logging) is seldom taken (~ 1/300)
Write Barrier Statistics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Long path frac.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjbb2000</td>
<td>1 / 299</td>
</tr>
<tr>
<td>Compress</td>
<td>1 / 894</td>
</tr>
<tr>
<td>Jess</td>
<td>1 / 13,210</td>
</tr>
<tr>
<td>Db</td>
<td>1 / 305</td>
</tr>
<tr>
<td>Javac</td>
<td>1 / 160</td>
</tr>
<tr>
<td>Mpegaudio</td>
<td>1 / 64,099</td>
</tr>
<tr>
<td>jack</td>
<td>1 / 16,572</td>
</tr>
<tr>
<td>mtrt2</td>
<td>1 / 4116</td>
</tr>
</tbody>
</table>
Performance Measurements

- Implementation for Java on the Jikes Research JVM
- Compared collectors:
  - Jikes parallel collector (Parallel)
  - Jikes concurrent RC (Jikes concurrent)
- Benchmarks:
  - Server benchmark: SPECjbb2000 --- business-like transactions in a large firm
  - Client benchmarks: SPECjvm98 --- mostly single-threaded client benchmarks
Pause Times vs. Parallel

Pause Times

- Jikes parallel
- tracing
- Jikes STW

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

Pause Times - Concurrent

- tracing
- Jikes Concurrent
SPECjbb2000 Throughput

SPECjbb200 - Tracing vs. Jikes STW

heap sizes

Jikes parallel

© Erez Petrank
Conclusions

- A non-intrusive, efficient mark & sweep garbage collector suitable for multiprocessors.
- An implementation on Jikes and measurements on a multiprocessor.
- Low pause times (1ms) small throughput penalty (10%).