Virtual Machine Introspection for Program Understanding and Debugging

Samuel Z. Guyer
Tufts University

October 6, 2009

The Good Ol’ Days

In the year 2000

Motivation

- Modern managed languages
- Large, complex runtime system
- Many decisions made dynamically
- Programmer further removed

Problem:
What to do when system doesn’t behave as expected?

Solution
- Virtual machine introspection
- VM has lots of useful information available
- Ask specific questions about what’s going on

Today’s talk: GC assertions

Work with Eddie Aftandilian at Tufts

- Garbage collection is great!
  - Program allocates objects, VM automatically reclaim them
  - No dangling pointers, no double frees
  - Programming utopia

- Problem:
  - No explicit free()
  - How do I know when objects get reclaimed?
  - How do I know they ever get reclaimed?
  - Answer: you don’t!

Is this a real problem?

- Example from SPEC JBB2000:
  - Emulates a 3-tier client/server system
  - Warehouses fulfill orders entered by clients

```java
/* Remove everything from the orderTable and then delete it */
Enumeration orderIter = orderTable.elements();
Order order;
while (orderIter.hasMoreElements()) {
    order = (Order)orderIter.nextElement();
    orderTable.removeEntry(order);
    Factory.deleteEntity(order);
}
Factory.deleteEntity(orderTable);
orderTable = null;
```
What am I expecting?

- **Goal**: free the data structure
  - Clear table, null out the references
  - GC will reclaim them at next collection
  - But does it?

- Turns out, no
  - Customer object holds a reference to the order
  - Customer object sticks around
  - We have a memory leak!

GC Assertions

- **Our solution**: A way to express expected behavior
  - *This object should be reclaimed at the next GC*
  - **Note**: programmer-driven technique
  - The programmer has to add new information
  - Observation: Both the programmer and the VM have valuable knowledge, but no way to communicate

- **Styed after assertions**: Add calls to `assert-property(obj)`
  - Focused on large-scale heap properties

GC Assertions

- **Key**: use garbage collector to check assertions
  - *Piggyback on existing GC heap traversal*
  - GC is already traversing the whole heap
  - Can check many useful properties
  - Without adding significant cost
  - Many cannot be checked any other way

- **Benefits**
  - More *precise* than static analysis
  - More *efficient* then run-time invariant checking
  - More *accurate* than heuristics

Kinds of properties we can check

- Lifetime properties
- Volume properties
- Shape properties

Lifetime properties

- **Problem**: Object that appears to be unreachable is not reclaimed
  - No direct way to find out if a specific object is reclaimed
  - Difficult to determine through program inspection

- **Solution**
  - `assert-dead(p)`
  - *p* should be reclaimed at next collection
  - Variation: `assert-reachdead(p)`
  - All objects reachable from *p* should be dead
Lifetime properties

- **Problem:**
  
  Server app – all objects allocated while servicing a connection should die when the connection terminates
  
  - How do we find out which objects were allocated for that connection?
  - Multiple connections – which one has a problem?

- **Solution:**
  
  `start-region();
  ... service connection ...
  assert-alldead();`
  
  Any object allocated between these two calls should be reclaimed at the next collection

Volume properties

- **Problem:**
  
  Only one instance of a class should be live at any time (singleton pattern)
  
  - Very difficult to enforce
  - Multiple class loaders, subclassing, serialization, etc.

- **Solution:**
  
  `assert-instances(T, I)`
  
  Number of live objects of type $T$ should not exceed $I$

Shape properties

- **Problem:**
  
  Nodes in a tree should never have more than one incoming pointer

- **Solution:**
  
  `assert-unshared(p)`
  
  - At most one object points to $p$
  - Could be extending to general in/out degree

Shape properties

- **Problem:**
  
  Objects in a collection are pointed to by a separate index.
  Index should never point to an object that is no longer in the collection.
  
  - Index should not keep object alive
  - Very common cause of memory leaks

- **Solution:**
  
  `assert-ownedby(p, q)`
  
  - Some path from root to $p$ passes through $q$
  - Note: not traditional notion of “owns”

Implementation

- **Jikes RVM 3.0.0**
- **Goal:** fast during correct program execution
- **Implemented in a pure mark-sweep collector**
  
  Generational collector would perform better but have longer time-to-detection
- **Basic strategies:**
  
  - Steal bits from object header, check during GC tracing
  
  Example:
    
    - On assert-dead(p), set bit in header of $p$
    - If we encounter this bit during GC, trigger assertion
    - Change order of heap traversal

Implementing assert-ownedby

- **Roots:**
- **Owner:**
- **Owness:**

Note: a limitation of this strategy – cannot have overlapping regions of ownership
Finding the error

- When assertion is triggered, how to find bug?
  - What information will be useful?
  - Observation: most assertions related to reachability and paths through the heap

Provide full path through heap to object
  - Sequence of references/objects from root to offending object
  - Helps determine which data structure is keeping an object alive
  - Implementation: small change to the object tracing code (Essence: leave gray objects in worklist)

Example full-path output

Warning: an object that was asserted dead is reachable.
Type: Lspec/jbb/Order
Path to object: Lspec/jbb/Company; -> Lspec/jbb/infra/Collections/StringStaticBTree; -> [Ljava/lang/Object; -> Lspec/jbb/infra/Collections/StringBTreeNode; -> [Ljava/lang/Object; -> Lspec/jbb/infra/Collections/StringBTreeNode; -> [Ljava/lang/Object; -> Lspec/jbb/Order;

Qualitative evaluation

- We use the assert-dead() assertion to find memory leaks in SPEC JBB 2000
  - Java memory leak – object is still reachable even though it won’t be used again
  - Uses Factory pattern with destroy() methods
  - Instrument destroy() methods with assert-dead() assertions
  - Also instrument loops where we know an object should be unreachable at the end of the loop

Qualitative evaluation

- Found several bugs:
  - “Dead” Order objects reachable from Customer objects; same with Address objects
  - In main program loop, a reference is kept to the Company object from the previous iteration
  - Order objects not properly removed from an orderTable when a DeliveryTransaction is completed
  - An object kept alive by an instance of a non-static nested class (hidden pointer)

- Full-path information was very helpful

Performance

- Benchmarks:
  - SPEC JVM98, SPEC JBB 2000, DaCapo

- Three configurations:
  - Base – unmodified RVM, unmodified benchmarks
  - Infrastructure – modified RVM, unmodified benchmarks
  - WithAssertions – modified RVM, modified benchmarks
    - 16k assertions in db, 15k checked at each GC
    - 31k assertions in jbb, 400 checked at each GC

Run-time overhead

Geomean: 2.75% slowdown for Infrastructure
GC time overhead

Run-time overhead

Geomean: 13.4% slowdown for Infrastructure

Geomean: 1.0% slowdown for WithAssertions

Related work

- Runtime invariant checking
- Modeling languages
  - [JML, Spec#]
- Incrementalization
  - [Shankar 07, Gorbovitski 08]
- QVM
  - [Arnold 08]
- Static analysis
- Leak detection
  - [Mitchell 03, Chilimbi 04, Bond 06, Jump 07]
- User-controlled garbage collection
- JVMTI
- Simplifiers
  - [O’Neill 06]

Current work

- Concurrent assertion checking
  - With Eran Yahav and Martin Vechev at IBM
  - Idea:
    - Piggyback on concurrent GC
  - Why do that?
    - Perform checks on a separate core
    - Could be hard: application keeps running
    - We know how to do this!
  - Snapshot-at-the-beginning concurrent collector
    - No need to invent a wacky new concurrent checking alg.
    - Use extra cores to improve reliability!

- GC assertion logic
  - With Yannis Smaragdakis and Neil Immerman
  - Problem:
    - Existing GC assertions are not composable
  - Idea:
    - Develop a language for GC assertions
  - Key: guarantee that all properties expressible in the language can be checked efficiently during heap scan
  - Many unknowns:
    - Can we check arbitrary data structure invariants?
    - Structural properties of lists, trees, etc.
Current work

- Automatic generation of GC assertions
  *With Kathryn S. McKinley and Maria Jump*

  - **Problem:**
    - Adding assertions could be a lot of work
  - **Idea:**
    - Use dynamic shape analysis to propose plausible assertions
    - Reviewed by the programmer
  - Example:
    - Shape analysis finds all tree nodes have one incoming pointer and two (disjoint) outgoing pointers

Future

- What else can we ask the VM?
  - Compilation questions?
    - Ask about code generation or particular optimizations
    - "I thought this method would get inlined, did it?"
  - Concurrency questions?
  - Security questions?

- In the year 2009…
  - I’m using Gmail
  - Javascript application…
  - Running in a browser written in Java…
  - Running in VMWare…

Thank You