Best practices for writing error free garbage collectors

Ian Rogers

Overview

Garbage collectors are multithreaded, concurrent and complicated enough to have bugs. How can runtimes be designed such that the chance for bugs is minimized? How can we diagnose bugs after they've occurred?

My background:

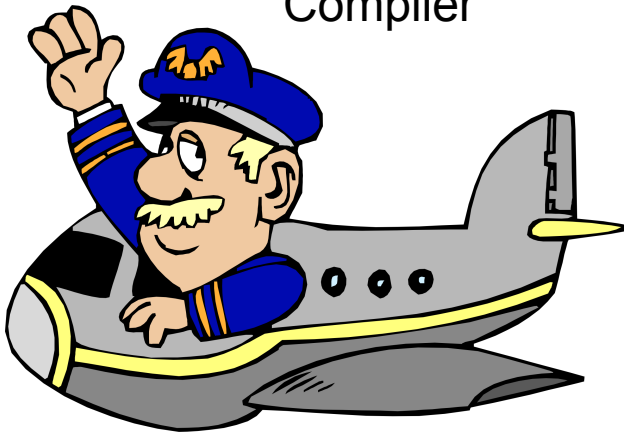
- Dynamite JVM - a binary translator based VM (Manchester University/Transitive).
  - C++ from 1995
- Jikes RVM and MRP - well-known metacircular VMs (OpenSource).
  - Java
- Zing VM and HotSpot - the de-facto JVM standard (Sun/Azul).
  - C++ from 1995
- ART/Dalvik - the runtime for Android (Google).
  - C++/clang from 2011

The general structure of a runtime team

Runtime

GC

Compiler



Images from openclipart.org

Google

What does a crash look like?

```
Stack: [0xffffffff76d00000,0xffffffff76e00000], sp=0xffffffff76dff240, free space=1020k Native frames: (J=compiled
Java code, j=interpreted, Vv=VM code, C=native code) V
[libjvm.so+0x5b236c] unsigned long CompactibleFreeListSpace::block_size(const HeapWord*)const+0x15c V
[libjvm.so+0x42a988] HeapWord* BlockOffsetArrayNonContigSpace::block_start_unsafe(const void*)const+0xf0 V
[libjvm.so+0xb21964] void CardTableModRefBS::process_chunk_boundaries (Space*,DirtyCardToOopClosure*,MemRegion,
MemRegion,signed char**,unsigned long,unsigned long)+0x1f4 V
[libjvm.so+0xb213f4] void CardTableModRefBS:: non_clean_card_iterate_parallel_work(Space*,MemRegion,
OopsInGenClosure*,CardTableRS*,int)+0x3dc V
[libjvm.so+0x518564] void CardTableModRefBS::non_clean_card_iterate_possibly_parallel(Space*,MemRegion,
OopsInGenClosure*,CardTableRS*)+0x64 V
[libjvm.so+0x519fe0] void CardTableRS::younger_refs_in_space_iterate(Space*,OopsInGenClosure*)+0x40 V
[libjvm.so+0x5f739c] void ConcurrentMarkSweepGeneration::younger_refs_iterate(OopsInGenClosure*)+0x54 V
[libjvm.so+0x706f54] void GenCollectedHeap::gen_process_strong_roots(int,bool,bool,bool,SharedHeap::ScanningOption,
OopsInGenClosure*,bool,OopsInGenClosure*)+0x1bc V [libjvm.so+0xb28bd0] void ParNewGenTask::work(unsigned)+0x150 V
[libjvm.so+0xcd8a0c] void GangWorker::loop()+0xa0 V [libjvm.so+0xb0786c] java_start+0x364
```

Example taken from: http://stackoverflow.com/questions/27001629/cassandra-solaris-crash-compactiblefreelistspaceblock-size

GC

Runtime

Compiler

Runtime

A broken GC map!

GC

Compiler

GC

A broken reference!

Compiler

Runtime

GC

Runtime

Compiler

Runtime

A broken GC map!

GC

Compiler

GC

A broken reference!

Compiler

Runtime

GC

Runtime

Compiler

Runtime

A broken GC map!

GC

Compiler

GC

A broken reference!

Compiler

Runtime

GC

Runtime

Compiler

Runtime

A broken GC map!

GC

Compiler

GC

A broken reference!

Compiler

Runtime

Terms

Bad root - points at nonsense such as the value of an int, or uninitialized memory

Stale root - a root that wasn’t processed during GC and is referencing the old location of an object or the location of a garbage collected object, frequently solved by the introduction of a handle

Memory corruption - program, or kernel/hardware bug, where memory is unexpectedly overwritten with bad data

Breaking the blame cycle

Check the “sanity” of a reference to an object at the point it is stored into the heap → maintains an invariant that the heap contains “sane” references or references broken by GC.

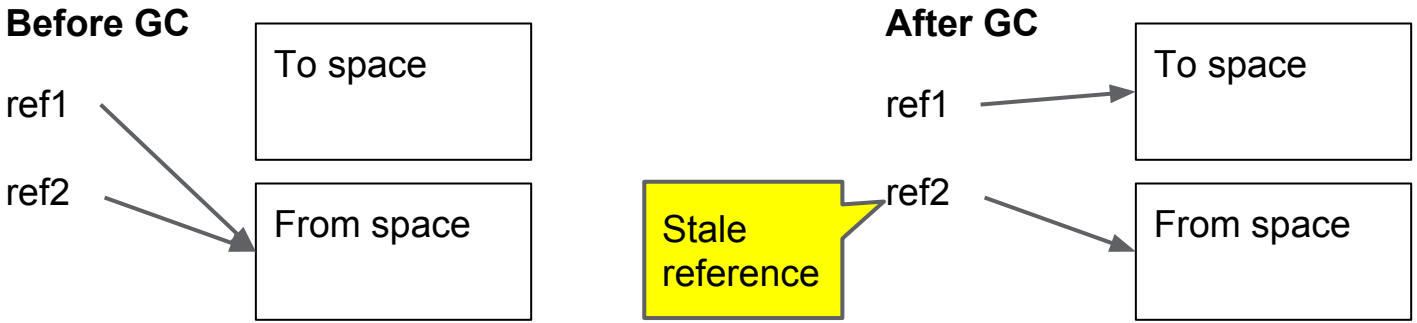
Example “sanity” check:

```
assert(value.getClass().getClass() == Class.class)
```

Fast enough to enable in debug builds.

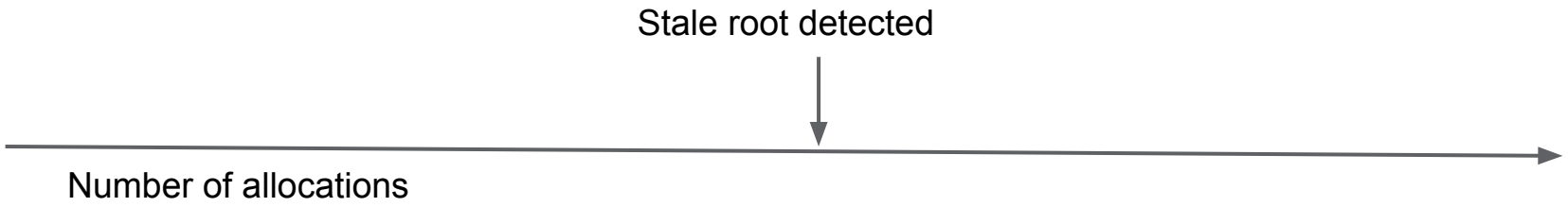
Make untracked references stale by moving objects

Its common in code working with naked pointers into the heap to miss handles. With a semispace GC we can move objects and protect the old space.
More spaces can mean more data on when reference became stale.



Make “unvisited” references stale by moving objects more frequently

Perform GC as often as possible, in the extreme at every allocation. To speed the search, binary search when to start/stop the intensive GC moving. Investigate what’s going on where the stale root appears earliest.



Working out where bugs lie once they are identified

Walk the heap and describe when things don’t appear as they should.

As you walk the heap carry around context to describe what a bug is.
For example:

- visiting thread roots describe the thread,
- visiting fields, describe the object containing the references.

Walk the heap before and after GC.

Working out where bugs lie once they are identified

Missed write barrier

- for example, the invariant that something in the new gen referenced from the old gen is in a remembered set is broken
- find references in the old gen that violates the invariant and die
- check the invariant a lot so as to narrow down where the bug lies

Finding missed read barriers

Reference poisoning:

- make values in the heap poisoned and remove the poison when they are loaded by the read barrier
- ‘xor -1’ or negation
- can cause (surmountable) issues with compilers

Invariant checking

- perform sanity checks, such as which space should an object be in
- already said we can catch at stores with sanity checks
- can add sanity checks into method calls, crossing safepoints, etc. to narrow down where a broken reference occurs

Working with naked (unhandlized) object pointers

Use TLS to record when you're doing this so that safepoint code can assert that you don't expect a safepoint.

Compilers can help, @Uninterruptible in Jikes RVM disallows safepoints within a method - new is a disallowed operation.
The compilers assert that all calls within @Uninterruptible code is to @Uninterruptible code.

Did metacircularity solve the missed Handle problem?

No:

```
Address x  = ObjectReference.fromObject(o).toAddress();
… // safepoint
… = x; // use of x
```

But they make it less likely as bugs only exist in clearly defined contexts.

The anti-handle

mandate that pointers to objects from local variables use a specific type, ie.

disallow Object*, but allow ObjectPtr

In debug builds make ObjectPtr a smart pointer that when created associates itself with a thread, on destruction removes itself. When a safepoint is crossed “zap” all ObjectPtrs on a thread.

Thread safety annotations

Ensure a lock is held when accessing a data structure.
Create partial order over lock/unlock operations to inhibit deadlock.

Only allow naked pointers when code has a lock on the heap, can be modeled with a ReaderWriterLock with mutators having shared/read access and the GC wanting exclusive/write access.

Partial order can be checked at runtime to work around limitations around function pointers, etc.

Memory checking tools

Valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

LLVM Sanitizers

- AddressSanitizer (ASan) is a fast memory error detector based on compiler instrumentation (LLVM).
- MemorySanitizer (MSan) is a detector of uninitialized reads. It consists of a compiler instrumentation module and a run-time library.
- ThreadSanitizer (TSan) is a tool that detects data races. It consists of a compiler instrumentation module and a run-time library.

Discussion

How else can we solve the problem of writing correct garbage collectors?

Is this a problem worth solving?

What other tricks do others know of?

Some notes from the discussion after the talk

Model checking proposed for testing certain classes of bugs, inparticular data races.

Forcing a runtime into a particular mode of operation such as interpret only.

Use patterns, .. to inhibit incorrect code coming into runtimes in the first place.

The earlier bugs are caught the better.