

Reducing and Eliding Read Barriers for Concurrent Garbage Collectors

Ian Rogers, Google
(work done at Azul Systems)



Here comes the marketing bit..

Azul Systems - Generation 3 Key Features

- Lots of cores - up to 864!
- Lots of RAM - 768GB!
- Proxy architecture - Java bytecode gets shipped to Vega appliance
- HotSpot JVM with tiered server compiler
- Real Time Performance Monitor
- Hardware supported Garbage Collection
 - Generational Parallel GC (from HotSpot)
 - Pauseless (see Click, Tene, Wolf - VEE 2005)
 - C4 (Continuous, Concurrent, Compacting Collector)
 - Stack allocation

Azul Systems - Generation 3 Key Features Continued

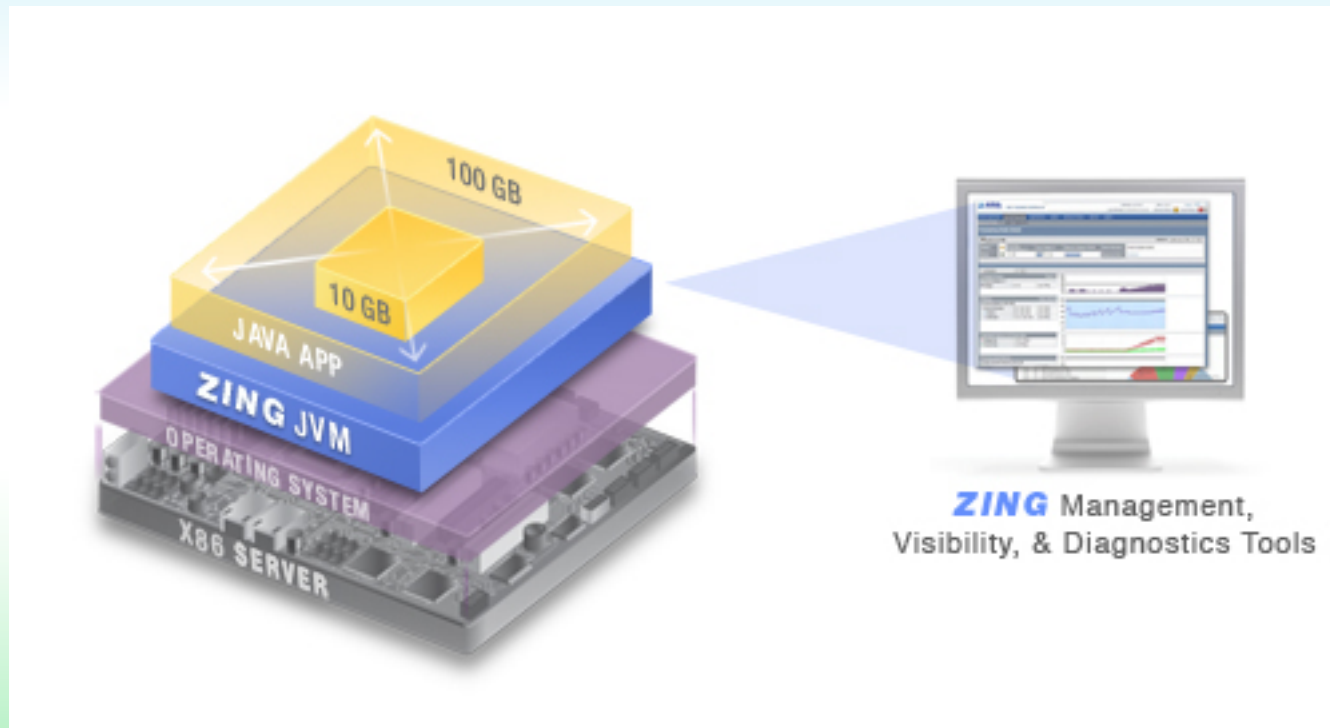
- Hardware supported Speculative Lock Elision (aka Speculative Multiaddress Atomicity)
 - Speculate that a synchronization operation wasn't necessary, change the hardware state to privatize the cache lines on access
 - Work on the private cache lines
 - Unprivatize the cache lines at end of synchronized region
 - Speculates until L1 cache is full, can handle TLB misses,..
 - Heuristics to avoid future speculation failures. Default is to switch to thin locking after 5 failed speculations on a particular object

Summary

- Arguably the state-of-the-art for problems requiring highly threaded, large memory problems
- In use for research (Purdue) and at many customer sites
- Expensive, but cost-effective for consolidation
- Pause free garbage collection is a necessity for large memory sizes
 - Assuming relocation pauses scale linearly, a 100ms pause for a 256MB heap would be 4seconds for a 10GB heap. At 768GB ?
 - Concurrent garbage collectors have safe point pauses and try to minimize the pause by heuristics
 - Heuristics are brittle - may be worse than naive GC
 - At scale the pause always becomes significant

The Zing VM - Azul's 4th Generation

- 64bit Intel/AMD processors have good (and always improving) multicore/socket story
- They have arrived at Azul scales of RAM



Key features of Zing

- Proxy architecture no longer required
- HotSpot VM with tiered compilation
- Zing monitoring tools
- Modified kernel support:
 - Make the OS and GC cooperate and avoid Inter-Processor Interrupts for TLB invalidation
 - Managed Runtime Initiative <http://managedruntime.org/> - Linux kernel source and HotSpot VM
 - ASPLOS RESoLVE 2011 paper - Rogers, Tene
- Parallel and C4 garbage collectors
 - ISMM 2011 paper - Iyengar, Tene, Wolf
 - contributions by Coha and Rogers

Challenges

- No hardware read barrier
 - There's a very good story on this, part of which is in this presentation.
- No hardware transactional memory
 - Various ideas for a software transactional scheme, but no solution for now

What is a read barrier?

- If objects are relocated during parallel GC then they may be in two locations
- Baker proposed invariant that mutator never sees white references
 - various impracticalities, not least that black-white pointers could occur
- Brooks proposed a forwarding pointer accessed through prior to use of a reference
 - IBM's RealTime GC (aka Metronome - Bacon et al) uses a Brooks barrier with weakened scheduling (scheduling restrictions forced by null pointer and safepoints)
- Appel-Ellis-Li propose using a pointer to black-only objects invariant

What is a Loaded Value Barrier

- Hardware instruction on Vega
- Fault if NMT state is wrong
 - ensure GC knows of all objects
- Fault if page is protected
 - ensures "only to-space" property of Appel-Ellis-Li
- Generational
- Scheduled prior to any use but not over safepoints (ala Metronome), can't cause NPEs (unlike Brooks/Metronome)
- Self-healing - a fault on the mutator will "fix" a reference
 - change mark state, update to relocated address, copy to relocation address and rewrite the original reference
- Can compact entire heap with 1 page of free memory
 - GC doesn't trigger memory allocation
- Cheap
 - We know how to make very cheap read barriers. E.g. Cycles to recycle: garbage collection on the IA-64 - Hudson, Moss, Subramoney, Washburn

Can we do better?

- LVBs are very fast, one non-memory instruction on Vega
- But we'd still prefer not to have one instruction

The C4/LVB Invariant - Traffic Lights

- A reference loaded from the heap may be in a red or green state
- A green reference needs no work by the barrier
- A red reference needs work by the barrier
 - To update NMT bit
 - To remap (update with new location) a reference from old to new location
 - To relocate (memcpy) an object from old to new location
- Similar to the "mutator only sees black objects" invariant of Appel-Ellis-Li, except performed at load rather than access (load-through/write-through) time

Removing LVBs on null

- null has the bit pattern of zero and is always green
- If loaded value is null then no fault is possible
 - `LVB(null) == null`

- `cur = head;`
- `size = 0;`
- `while(cur != null) {`
 - `cur = cur.next;`
 - `size++;`
- `}`

Removing LVBs on null

- null has the bit pattern of zero and is always green
- If loaded value is null then no fault is possible
 - `LVB(null) == null`
- `cur = LVB(head);`
- `size = 0;`
- `while(cur != null) {`
 - `cur = LVB(cur.next);`
 - `size++;`
- `}`

Removing LVBs on null

- null has the bit pattern of zero and is always green
- If loaded value is null then no fault is possible
 - `LVB(null) == null`
- `cur = head;`
- `size = 0;`
- `if(cur != null) {`
 - `do {`
 - `cur = LVB(cur);`
 - `size++;`
 - `cur = cur.next;`
 - `} while (cur != null);`
- `}`

Reducing LVBs on equality

```
class Holder {  
    Object x;  
    bool equals(Object y) { return x == y; }  
}
```

- y is a reference and must be green by the invariant

Reducing LVBs on equality

```
class Holder {  
    Object x;  
    bool equals(Object y) { return LVB(x) == y; }  
}
```

- y is a reference and must be green by the invariant
- x is loaded and so must have a LVB
- we know that equals is typically called when two objects have matching hash codes
- therefore the probability that $x == y$ is high
 - measure by profiling
- $x == y \Rightarrow x$ must be green
- $x != y \Rightarrow x$ may be green or red
- **return $(x == y) || (LVB(x) == y)$**

Reducing LVBs in loops

```
bool contains(Object array[], Object value) {  
    for(int i=0; i < array.length; i++) {  
        if (LVB(array[i]) == value) return true;  
    }  
    return false;  
}
```

Reducing LVBs in loops

```
bool contains(Object array[], Object value) {  
    for(int i=0; i < array.length; i++) {  
        if (array[i] == value) return true;  
    }  
    for(int i=0; i < array.length; i++) {  
        if (LVB(array[i]) == value) return true;  
    }  
    return false;  
}
```

Reducing LVBs in loops

```
bool contains(Object array[], Object value) {  
    for(int i=0; i < array.length; i++) {  
        if (array[i] == value) return true;  
    }  
    for(int i=0; i < array.length; i++) {  
        if (LVB(array[i]) == value) return true;  
    }  
    return false;  
}
```

- LVB fault is statistically unlikely
- `array[i] == value` \Rightarrow `array[i]` was green
- if we miss then `array[i]` may have been red, retry loop with LVB
- may execute more code than inline LVB
- profile to find out

Going further...

```
X = test ? LVB(Y) : LVB(Z);
```

to:

```
X = LVB(test ? Y : Z)
```

- May break invariant if load spans safepoint

Going further 2...

- Use integers rather than references for commonly accessed metadata. For example:
 - class pointer becomes class identifier (KID)
 - KID to class uses a lookup table
 - lookup table seldom used as:
 - KID equality serves to guard polymorphic inline caches
 - for instanceof/checkcast the class is a literal
- other improvements for megamorphic method dispatch, ArrayStoreCheckException...

Conclusions

- Big heaps are here and lead to significant GC pauses
 - A 4 second pause may look like a missed heartbeat and a machine may be rebooted
- We can build **very** cheap read barriers
- We can reduce the frequency of recurring GC faults (self healing) and other improvements with the LVB
- Compiler optimizations can remove LVBs on:
 - compare with null
- Compiler optimizations can make LVBs less common on:
 - reference equality operations
 - iterating over arrays
- We can design out read barriers from metadata

Conclusions 2

- Large heaps don't mean avoid GC
- Zing is often faster than regular HotSpot but with no GC pauses
- Zing is further refined so that VM operations don't cause pauses (found by profiling) trading performance for pause-free behavior
- Yet more is possible and being done