Data Caching, Garbage Collection, and the Java Memory Model

Wolfgang Puffitsch
wpuffits@mail.tuwien.ac.at

JTRES ’09, September 23-25, 2009
Motivation I

- Sequential consistency is expensive
- Multi-processors often implement relaxed memory models
- JMM is a logical choice for a Java processor
Motivation II

- JMM specifies memory model for application
- JMM is agnostic of run-time system
- Minimal communication between application and GC
  - Asymmetric synchronization
The Java Memory Model

- *Happens-before* relation
- Similar to lazy release consistency
- Allows various optimizations
- Rules out a number of odd behaviors
  - Causality must be obeyed
Surprising Behavior

```java
int x = 0;

<table>
<thead>
<tr>
<th>Thread T1</th>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>int r1 = x;</td>
<td>int r2 = x;</td>
</tr>
<tr>
<td>x = 1;</td>
<td>x = 2;</td>
</tr>
</tbody>
</table>
```

Java memory model allows $r1==2$, $r2==1$
Data Cache Implementation 1

- Implemented for JopCMP
- Predictable, low HW cost
- Follows idea of lazy release consistency
- Invalidate cache on `monitorenter` and volatile reads
- Write-through cache
Data Cache Implementation II

- No global store order
- Accesses cannot bypass each other locally
  - Relatively simple memory model
- Good predictability
  - Consistency actions are always local
Moving Objects

- Only minimal communication between application and GC
- Avoid synchronization overhead for reads
- How to force application to see moved objects?
  - Invalidate cache for each moved object
  - Stronger memory model
  - Avoid movement of objects
void runGC() {
    // initiate new GC cycle
    startCycle();
    // retrieve roots
    gatherRoots();
    // trace the object graph
    traceObjectGraph();
    // clear objects that are still white
    sweepUnusedObjects();
    // optional memory defragmentation
    defragment();
}
Tricolor Abstraction

- *White* objects have not been visited
- *Gray* objects need to be visited
- *Black* objects have been visited
- After tracing, reachable objects are black and white objects are garbage
void traceObjectGraph() {
    // while there are still gray objects
    while (!grayObjects.isEmpty()) {
        // get a gray object
        Object obj = grayObjects.removeFirst();
        // iterate over all reference fields
        for (Field f in getRefFields(obj)) {
            Object fieldVal = getField(obj, f);
            // mark referenced objects
            if (color(fieldVal) == white) {
                markGray(fieldVal);
            }
        }
        markBlack(obj);
    }
}
void putFieldRef(Object obj, Field f, Object newVal) {
    // snapshot—at—beginning barrier
    Object oldVal = getField(obj, f);
    if (color(oldVal) == white) {
        markGray(oldVal);
    }
    // write new value to field
    putField(obj, f, newVal);
}
Tracing Requirements

The object graph can be traced correctly if

- a snapshot-at-beginning write barrier is used, and
- new objects are allocated non-white, and
- a consensus is established at the beginning of tracing
Objects must either be reachable from snapshot or newly allocated

Differences in object graph views must stem from updates $\Rightarrow$ write barrier

Concurrent updates must see snapshot
  
  Works for our cache implementation
  
  Not guaranteed in JMM!
Tracing – JMM Counterexample

\[
x.f == A;
\]

<table>
<thead>
<tr>
<th>Thread T1</th>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obj o1 = x.f;</td>
<td>Obj o2 = x.f;</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>x.f = B;</td>
<td>x.f = C;</td>
</tr>
</tbody>
</table>

Java memory model allows o1==C, o2==B!
Sliding Consensus

- Consensus is established by invalidating all caches
- How to make this non-atomically?
  - Sliding view root scanning
  - Invalidate cache at root scanning
- Assuming double-barrier
  - Both old and new value are shaded
Start of GC Cycle – Requirements

- Field updates from earlier GC cycles must be visible to write barriers of new GC cycle
- Field updates from earlier GC cycles must be visible to root scanning
- Field updates from earlier GC cycles must be perceived consistently
Start of GC Cycle – Consequences

- Clear separation of GC cycles
- Threads that are preempted while executing a write barrier delay start of a GC cycle
Start of GC Cycle – Future work

- Costs of implementation choices to be evaluated
- Avoid overlap of old and new barriers
  - Handshake or mutual exclusion
- Enforce consistent perception in write-barrier
  - Bypass cache or cache invalidation
Object Initialization

- Threads *must* see default values
- Avoid synchronization between allocation and potential uses
- Memory must not have been in use since last GC cycle
- Cache invalidation at GC cycle start \(\Rightarrow\) Cache cannot contain stale values
- Analogue consideration for final values
Internal Data Structures

- Inter-thread communication of GC algorithm
- Internal data structures can follow own memory model
  - E.g., bypass cache
  - Avoids merging application and run-time synchronization
  - Depends on capabilities of platform
Conclusion 1

- Cache that is consistent with JMM
- Moving of objects needs consistency enforcement
- Tracing works if JMM surprising behavior is avoided
- Start of GC cycle requires careful design
Conclusion II

- Object creation simple in some cases
- Run-time system synchronization can be separated from application synchronization
Thank you for your attention!