Distributed Garbage Collection for Managed X10

2012/06/14
Kiyokuni Kawachiya, Mikio Takeuchi, Salikh Zakirov*, Tamiya Onodera
IBM Research - Tokyo

* Salikh Zakirov is currently affiliated with Google Inc.
This Talk Focuses on “Managed X10”

Managed X10 = X10 implementation on Java (i.e. managed runtime)
– X10 program is translated into Java source code and compiled into Java bytecode, then executed on multiple Java VMs
Outline of the Talk

Distributed Garbage Collection for Managed X10

- Distributed Processing in X10
  - APGAS programming model
  - Distributed processing with GlobalRef

- Distributed GC for Managed X10
  - Requirements
  - Core data structures: GOT and RRT
  - Behavior of the distributed GC

- Evaluation
  - Micro-Benchmark: basic behavior and overhead
  - Macro-Benchmark: distributed Fibonacci

- Conclusion

- Future and Related Work
Distributed Execution Model of X10

APGAS: Asynchronous Partitioned Global Address Space

- A global address space is divided into multiple places (≡ computer)
  - Place can contain activities and data (objects, structs, functions)
  - In Managed X10, each place is implemented by a Java VM
- An object belongs to a place where it is created
  - Object cannot be accessed from other places, but
  - Object can be remotely referenced from other places, using GlobalRef

Figure 1. APGAS Programming Model.
GlobalRef is a special struct to hold a *global reference* to an object
- Created by “GlobalRef[T](obj)” and cannot be modified
- The object can be accessed by “at (g) g()...”
Garbage Collection in Managed X10

- X10 data is represented by Java objects and collected by each JVM’s GC
  - However, remote reference is not a reference in the JVM level
  
  ![Diagram of Garbage Collection in Managed X10](image)

- In old X10, remotely-referenced (globalized) objects were registered into a management table and never collected

⇒ We needed better implementation
GlobalRef Implementation in Managed X10

- GlobalRef is implemented in Java code and has the following fields:
  - **obj**: Reference to the object
  - **home**: Place where the obj exists
  - **id**: Unique ID in the place

- How GlobalRef works

```
1 class GRefExample {
2   static class ResultBox { var value: Int = 0; }
3   public static def main(Array[String]) {
4     val place1 = here.next();
5     val o = new ResultBox();
6     val g = GlobalRef[ResultBox](o); // g
7     finish {
8       at (place1) async { // g'
9         val r = do_long_calculation(g);
10        at (g) g().value = r; // g''
11      }
12      do_some_calculation_locally();
13     } // end of finish
14     Console.OUT.println(o.value);
15   }
```

Figure 3. Transition of GlobalRef Fields.

Figure 7. Java-Level Definition of GlobalRef.
Requirements to the GlobalRef Implementation

1. **The same ID must be used for the GlobalRefs that hold the same object**
   - Necessary to check the equality of the GlobalRefs in remote places
   - Once an ID is assigned to an object, it should not be changed until the object is collected as garbage

2. **A local object can be searched for by using the ID**
   - Necessary to restore the obj field from id when a GlobalRef returns to its home place

3. **Objects whose GlobalRefs exist in remote places must not be collected as garbage**
   - Otherwise, an error will occur when the remote reference is used

4. **Objects that are not referenced either locally or remotely should be collected**
   - If not, OutOfMemoryError may occur
   - This distributed GC function is the main topic of this talk

5. **When the globalized object is collected, related management information (such as the object’s ID) should also be deleted**
Introduce “Globalized Object Tracker” (GOT)

- Centralized data structure to track a globalized object, which has
  - `weakRef` Weak reference to the globalized object
  - `strongRef` Strong reference to the object
  - `remoteCount` Number of remote refs to the object
  - `id` ID of the object

- How GOT works
  - `weakRef` is used to detect the object’s collection (Req. 5)
  - `strongRef` is used to prohibit the object’s collection (Req. 3)
  - `strongRef` is cleared if `remoteCount` is zero, which enables the object’s collection (Req. 4)
  - GOT is registered in two tables (ConcurrentHashMaps)
    - `id2got` to search for an object from its ID (Req. 2)
    - `got2id` to search for an ID from an object, if assigned (Req. 1)

- How to maintain the remoteCount?
  - We introduced another data structure, Remote Reference Tracker (RRT)
Introduce “Remote Reference Tracker” (RRT)

- Internal data structure to track a remote reference (remote GlobalRef), and provides two functions
  6. Detect that a remote reference is no longer needed
  7. Merge the same remote references

- RRT has the following fields
  - weakRef: Weak reference to the remote GlobalRef
  - home: Home place of the GlobalRef
  - id: ID of the object held by the GlobalRef
  - weightCount: Weight count assigned to the remote reference

- How RRT works
  - weakRef is used to detect that the GlobalRef is collected (Req. 6)
    - A notification is sent to home place to decrease the remoteCount of GOT
  - home/id are used for the notification, and to merge same GlobalRefs (Req. 7)
    - For the merging, RRT is registered in rrtTable
  - weightCount is used to reduce communication with the home place
    - GOT’s remoteCount field holds the total value of weightCounts of the corresponding RRTs
Management of Remote Count

**Weighted Reference Counting** is used
- GOT’s remoteCount = \( \sum \) RRTs’ weightCounts

- **GOT’s remoteCount is increased** when a local GlobalRef is being sent to another place
  - This should be done *speculatively* and *synchronously* to avoid racing condition
  - This can be done locally without additional inter-place communication

- **RRT’s weightCount is divided** when a remote GlobalRef is further sent to another place
  - No need to increase the GOT’s remoteCount if the remaining weightCount > 1

- **GOT’s remoteCount is decreased** when a remote GlobalRef is deleted
  - This needs additional inter-place communication, but can be done *asynchronously*
  - If the remoteCount becomes 0, the GOT’s strongRef field is cleared to make the tracking object collectable
Behavior of the Distributed GC

Creating and using a GlobalRef

```java
1 class GRefExample {
2   static class ResultBox { var value:Int = 0; }
3   public static def main(Array[String]) {
4     val place1 = here.next();
5     val o = new ResultBox();
6     val g = GlobalRef[ResultBox](o); // g
7     finish {
8       at (place1) async { // g'
9         val r = do_long_calculation(g);
10        at (g) g().value = r; // g''
11       }
12       do_some_calculation_locally();
13     } // end of finish
14     Console.OUT.println(o.value);
15   }
```

Figure 3

Figure 9. Data Structures for the Distributed GC.
Behavior of the Distributed GC

Collecting the globalized object, and related data

---

1 class GRefExample {
2   static class ResultBox { var value:Int = 0; }
3   public static def main(Array[String]) {
4     val place1 = here.next();
5     val o = new ResultBox();
6     val g = GlobalRef[ResultBox](o); // g
7     finish {
8       at (place1) async { // g'
9           val r = do_long_calculation(g);
10          at (g) g().value = r; // g''
11       }
12     } // end of finish
13     do_some_calculation_locally();
14   } // end of main
15 } }

---

Extra inter-place comm. is performed only when a remote ref. is removed.
Evaluation: Basic Behavior

Micro-benchmark which creates remote references 100,000 times

```
1 class MicroBench {
    2   static class ResultBox { var value: Int = 0; }
    3   public static def main(Array[String]) {
        4     val place1 = here.next();
        5     for (var i: Int = 0; i < 100000; i++) { // do 100,000 times
        6         val o = new ResultBox();
        7         val g = GlobalRef[ResultBox](o); // 1
        8         at (place1) { // send the GlobalRef // 2
        9             at (g) g().value = 1; // use the GlobalRef // 3
        10         }
        11     if (o.value == 0) Console.OUT.println("error");
        12     } // end of for loop
        13  }
```

Figure 11. Micro-Benchmark for the Evaluation.

Results
- Dist. GC can save heap consumption
- Small overhead in creating and using remote references

Table 1. Cost for Using GlobalRef

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Dist. GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a GlobalRef</td>
<td>41ns</td>
<td>41ns</td>
</tr>
<tr>
<td>2. Create a remote ref</td>
<td>73μs</td>
<td>84μs</td>
</tr>
<tr>
<td>3. Use the remote ref</td>
<td>74μs</td>
<td>82μs</td>
</tr>
</tbody>
</table>
Evaluation: Distributed Fibonacci

- Calculates \( n \)-th Fibonacci number recursively using multiple places
  - Field \texttt{root} is a GlobalRef that points to the “root object”
  - Field \texttt{v} of the root object is accessed by “at (root) root().v ...”

```scala
1 class DistFib {
2   static val atomicI = new x10.util.concurrent.AtomicInteger(0);
3   static def getAvailPlace() { // returns next available place
4     val i = at (Place.FIRST_PLACE) atomicI.incrementAndGet();
5     return Place(i % Place.MAX_PLACES);
6   }
7
8   private val root = GlobalRef[DistFib](this); // ref to root obj
9   transient var v:Int; // in-out param, exists only in the root
10  def this(n:Int) { v = n; }
11  def compute() { // global method, can be invoked at any place
12     val n = at (root) root().v; // get the input from the root object
13     if (n < 2) return; // Fib(0)==0, Fib(1)==1
14     val f1 = new DistFib(n-1), f2 = new DistFib(n-2);
15     finish {
16       at (getAvailPlace()) async f1.compute(); // compute remotely
17       f2.compute(); // compute here
18     }
19     val r = f1.v + f2.v; // result = Fib(n-1) + Fib(n-2)
20     at (root) root().v = r; // set the result to the root object
21   }
22
23  public static def main(args:Array[String](1)) {
24    var n:Int = 10; if (args.size > 0) n = Int.parseInt(args(0));
25    val f = new DistFib(n); f.compute(); // this line is measured
26    Console.OUT.println(f.v);
27  }
28}
```

**Figure 13. Distributed Fibonacci.**
Result: Distributed Fibonacci

Execution times to calculate $n$-th Fibonacci numbers using 16 places

- For very small $n$'s, base X10 is faster by about 10%
  - Because of the small overhead in the Dist. GC version
- For $n=25$, distributed GC version is 13% faster
  - Because heap is not consumed by uncollectable objects
  - Heap size of Place 0 reaches 468MB in base, but 148MB in Dist. GC
Conclusion

✓ Proposed a distributed GC to collect remotely-referenced objects in Managed X10, running on multiple JVMs
  – No modification to the underlying JVMs
  – Globalized Object Tracker (GOT) centralizes the control of a globalized (remotely-referenced) object

✓ Implemented the proposed distributed GC on Managed X10
  – Remotely-referenced objects are correctly collected with almost no performance overhead
  – The change is included in the latest X10 source code
    • See x10.runtime/src-java/x10/core/GlobalRef.java