EscapeAnalysis

Escape Analysis

From: Vladimir Kozlov
Date: May 15, 2009 1:47:21 PM PDT

_C2 implements the flow-insensitive escape analysis algorithm described in:

```
[Choi99] Jong-Deok Shoi, Manish Gupta, Mauricio Seffano,
         Vugranam C. Sreedhar, Sam Midkiff,
         "Escape Analysis for Java", Procedings of ACM SIGPLAN
         OOPSLA  Conference, November 1, 1999
```

The analysis requires construction of a "connection graph" (CG) for the method being analyzed. The nodes of the connection graph are:

```
    -  Java objects (JO)
    -  Local variables (LV)
    -  Fields of an object (OF),  these also include array elements
```

C2 does not have local variables. However for the purposes of constructing the connection graph, the following IR nodes are treated as local variables:

```
    Phi       (pointer values)
    LoadP, LoadN
    Proj#5    (value returned from call nodes including allocations)
    CheckCastPP, CastPP, EncodeP, DecodeN
    Return    (GlobalEscape)
```

The LoadP, Proj and CheckCastPP behave like variables assigned to only once. Only a Phi can have multiple assignments. Each input to a Phi is treated as an assignment to it.

The following node types are JavaObject:

```
    top()
    Allocate
    AllocateArray
    Parm                (for incoming object arguments, GlobalEscape)
    CastX2P             ("unsafe" operations, GlobalEscape)
    CreateEx            (GlobalEscape)
    ConP, ConN          (GlobalEscape except for null)
    LoadKlass, LoadNKlass (GlobalEscape)
    ThreadLocal         (ArgEscape)
```

AddP nodes are fields.

After building the graph, a pass is made over the nodes, deleting deferred nodes and copying the edges from the target of the deferred edge to the source. This results in a graph with no deferred edges, only:

```
   LV -P> JO
   OF -P> JO (the object whose oop is stored in the field)
   JO -F> OF
```

After that escape analysis makes a pass over the nodes and determines nodes escape state:

- GlobalEscape - An object escapes the method and thread (stored into a static field or stored into a field of an escaped object or returned as the result of the current method).
- ArgEscape - An object passed as argument or referenced by argument but not globally escape during a call (by analyzing the bytecode of called method).
- NoEscape - A scalar replaceable object.

After escape analysis C2 eliminates scalar replaceable object allocations and associated locks. C2 also eliminates locks for all non globally escaping objects. C2 does NOT replace a heap allocation with a stack allocation for non globally escaping objects.

Some scenarios for escape analysis are described next.

- The server compiler might eliminate certain object allocations. Consider the example where a method makes a defensive copy of an object and returns the copy to the caller.

```
      public class Person {
        private String name;
        private int age;
        public Person(String personName, int personAge) {
          name = personName;
                      age = personAge;
        }

        public Person(Person p) { this(p.getName(), p.getAge()); }
        public int getName() { return name; }
        public int getAge() { return age; }
      }

      public class Employee {
        private Person person;

              // makes a defensive copy to protect against modifications by caller
              public Person getPerson() { return new Person(person) };

              public void printEmployeeDetail(Employee emp) {
                Person person = emp.getPerson();
                // this caller does not modify the object, so defensive copy was unnecessary
                      System.out.println ("Employee's name: " + person.getName() + "; age: "  + person.
getAge());
              }
      }
```

The method makes a copy to prevent modification of the original object by the caller. If the compiler determines that the getPerson method is being invoked in a loop, it will inline that method. In addition, through escape analysis, if the compiler determines that the original object is never modified, it might optimize and eliminate the call to make a copy.

- The server compiler might eliminate synchronization blocks (lock elision) if it determines that an object is thread local. For example, methods of classes such as StringBuffer and Vector are synchronized because they can be accessed by different threads. However, in most scenarios, they are used in a thread local manner. In cases where the usage is thread local, the compiler might optimize and remove the synchronization blocks.