Comprehensive Analysis of Objects for Efficient handling of Java Objects

Prasanna Kumar Kalle
IBM India Software Labs
Bangalore - 560017

Abstract:

JAVA is a highly Object Oriented Language and supports all functionality to be encapsulated within classes. Having encapsulated functionality within classes' results in access to all methods generally through objects, unless of course a method is declared with a static scope. The JAVA language, necessitates that the objects be created on the JAVA heap. This heap needs to be periodically scanned for stale objects and cleaned for optimum usage of the heap. It is the responsibility of the Garbage Collector (GC) to do a periodic cleanup of the JAVA heap. Object creation, allocation and cleanup are thus the prime responsibility of the storage component of the Java Virtual Machine (JVM).

This task impacts the performance of the JVM highly due to the involvement of locking and unlocking for each and every access to objects in the heap. This adds up to the performance restriction imposed due to the fact that JAVA is an interpreted Language.

This paper aims at providing a mechanism to do a comprehensive analysis of the Objects in the JAVA heap to detect the scope and degree of escapeness of objects. There have been multiple papers on this subject in the past. This paper aims at a simpler approach for analyzing the escapeness using the set theory based analysis implementation.

Set based analysis is simpler to both understand and implement due to the involvement of less complicated structures, than the traditional graph based approach. It reduces the amount of memory necessary for the structures during the analysis phase. Simpler structures and algorithm also reduces the amount of time needed for the analysis thus enhancing the performance.

Apart from the other uses like stack allocations, this paper also aims at using a reduced object layout for the stack object.

1. Introduction:

Dynamically allocating objects in the Java heap and freeing up memory is a time intensive activity. Though this cannot be totally reduced to zero due to the nature of the programming language as such, certain compile time optimizations can indeed be done so that we can reduce the impact of the same. The algorithm described below is termed *Escape Analysis.*

This algorithm analyses all the objects involved in the method under consideration and classifies them according to their scope of access and life time.

The information gathered from this analysis can them be used to do other optimizations like

- Stack allocation of Objects.
- Removal of unnecessary synchronization constructs on an object if the object under consideration is not accessed by multiple threads.
- Replacing calls to objects by scalars if we do not need any references to the object as such.

In this manner the number of heap controlled objects can be reduced. Depending upon the nature of the method getting compiled, this would have a significant impact on the performance of the application.

2. Escape Analysis:

2.1 Theory:

2.1.1 Source and Destination node sets:

This algorithm analyses one Java method at a time. During the analysis, the algorithm traverses all the basic blocks in the method and creates a set of all the objects involved in the method. This set is denoted by the Universal set **U.** This set consists of all those local variables in the intermediate language representation (IR), which

denote objects. The Universal set is then categorized into source set **S** and destination set **D.** This categorization depends upon whether the local variable represents a source variable or the destination variable in the IR.

The source and destination could have certain nodes in common. This is due to the fact that the destination value for a quadruple might be the source for subsequent quadruple. Also called as the three address notation, quadruple is a format of IR, the others being Abstract Syntax Tree (AST) and Directed Acyclic Graph (DAG). For example, consider the following snippet of IR.

```
LO3  = LO0         AMOVE
LI4  = +I0         IMOVE
       LO3         NULLCHECK c
LI5  = LO3         ARRAYLENGTH
       LI4 , LI5   SIZECHECK
LL7  = LI4 , +I3   LOP15-scale
LO3  = LO3 , LL7   AALOAD
LO2  = LO3         AMOVE
```

In the above quadruple representation, it can be seen that LI4 is the destination argument for the quadruple IMOVE and is the source argument for the SIZECHECK argument. Similarly, LO3 is the destination argument for AMOVE and the source argument for ARRAYLENGTH quadruple.

Since, there can be an overlap between the source and destination arguments, we define the Universal set **U** in terms of the source and destination sets as below.

{U} = {S} U {D} **Eq 2.1**

For example, if we consider a simple HelloWorld program written in Java as below,

```
public class HelloWorld {
        public static void main(String [] args) {
                int i = Integer.parseInt (args[0]);
                while ( i-- > 0)
                   System.out.println        ("Hello
World " + i );
        }
}
```

The corresponding quadruple sequence is as below.

```
LO0   = AO0      ALOCALCOPY
LI6   = LO0      ARRAYLENGTH
LO2   = LO0  ,+I0    AALOAD
AI1   = +I10     IARGCOPY
AO0   = LO2      AARGCOPY
LI1   = AO0  ,AI1    IINVOKE
LI4   = LI1      IMOVE
LI1   = LI1  ,-I-1   IOP2-add
LO9   =          AGETSTATIC
LO3   =          NEW
LO7   = CO188B33DC SCONST
AI2   = LI1      IARGCOPY
AO1   = LO7      AARGCOPY
AO0   = LO3      AARGCOPY
LS10  = LO9      MTLOAD
LS11  = LS10,+S36    MBLOAD
AO1   = LO3      AARGCOPY
AO0   = LO9      AARGCOPY
```

In the above illustration,

{S} = { Set of all nodes to the right side of "="}
{D} = {Set of all nodes to the left side of "="}
{U} = {Set of all nodes with no repetition. }

However, since escape analysis deals only with objects, we redefine **{S}, {D}** and **{U}** by restricting them to only nodes that represent objects.

So, in general, we define **{S}, {D}** and **{U}** as

{S} = { Set of all nodes to the right side of "=", that represent objects }
{D} = {Set of all nodes to the left side of "=", that represent objects}
{U} = {Set of all nodes that represent objects with no repetition. } **Eq 2.2**

2.1.2 Categorization of nodes:

Depending on what is the functionality of the associated quadruple, the nodes can be classified as below.

a. **New object node:** This node is categorized by creation of a new object or an array object of basic data type or a derived data type. The quadruples involved generally are NEW, NEWARRAY, ANEWARRAY.

b. **Reference load node:** This node loads a reference to any object into another local variable. Few of the quadruples involved are

ALOCALCOPY, AGETFIELD, AMOVE, and AALOAD.

c. **Argument node:** Whenever an argument needs to be passed to a method being invoked, this is done in the argument node. This can be treated as a special kind of reference load node. Generally, this node is identified by AARGCOPY.

d. **Return node:** Whenever a method returns a reference to an object to the caller method, the node is called a return node and is identified by ARETURN.

2.1.3 Inside and Outside nodes :

Depending on where the actual object is created, whether within the current method under analysis or outside the method, the objects are classified into inside objects and outside objects. Accordingly, the corresponding nodes are termed *inside nodes* and *outside nodes* respectively.

Since, a new object node is the only node wherein a new object gets created, it is the only type of node which is an inside node.
Whereas, reference load node and argument node are outside node.

2.1.4 Escaping versus non-escaping nodes :

Nodes are classified into escaping and non-escaping nodes depending on the status of the object they represent.

Degree of escapeness of an object may be defined as the ability to predict the state of the object with regard to the method being analyzed. An object is said to be *escaped* if at any point in time during the analysis of the method, the status of the object cannot be determined. This generally happens in the following cases.

a. Object is assigned to any static (class) variable that can be accessed and altered from any other location within the application. Such an object is said to be *globally escaping*.
b. Object is returned from the method where it is getting created. In this case, the object escapes from the current scope of creation. However, this might then be captured further down the call tree. If it is captured, then it is said to be *locally escaping*.
c. An object might also escape if it is assigned as a field of another object that is escaping.
d. Any object is said to be *non-escaping* if the life span of the object is only with in the method of creation and cannot be accessed from anywhere outside the method.

Outside nodes are always treated as escaping nodes whereas inside nodes may be escaping or non-escaping. If escaping, they might be locally escaping or globally escaping. A globally escaping node will never be captured. A locally escaping node can be captured down the call tree whereas a non-escaping node is captured in the method where it is created.

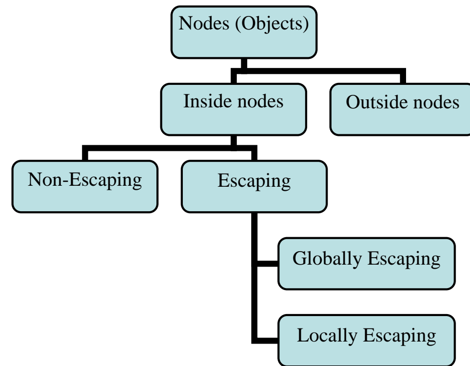

Fig 2.1.4

2.1.5 Set theory representation :

We represent new object node set, reference load node set, argument node set and return node set as $\{N_o\}$, $\{N_l\}$, $\{N_a\}$ and $\{N_r\}$ respectively. The union of all these nodes adds to the universal set.

$$\{U\} = \{N_o\} \cup \{N_l\} \cup \{N_a\} \cup \{N_r\} \qquad \textbf{Eq 2.3}$$

Globally escaping nodes are represented by $\{N_{gl}\}$, locally escaping nodes are represented by $\{N_{lo}\}$ and non-escaping nodes are represented by $\{N_{ne}\}$.

$$\{N_{gl}\} \cap \{N_{lo}\} = \Phi$$
$$\{N_{ne}\} \cap \{N_{lo}\} = \Phi$$
$$\{N_{ne}\} \cap \{N_{gl}\} = \Phi \qquad \textbf{Eq 2.4}$$

$\{N_{ne}\} = \{U\} - (\{N_{gl}\} \; U \; \{N_{lo}\})$ **Eq 2.5**

Outside nodes are represented by $\{\mathbf{N_{out}}\}$ and inside node by $\{\mathbf{N_{in}}\}$.

$\{N_{out}\} \cap \{N_{in}\} = \Phi$
$\{N_{out}\} \; U \; \{N_{in}\} = \{U\}$ **Eq 2.6**

2.2 Algorithm:

The algorithm followed below is relatively simple to comprehend.

2.2.1 Phase 1 (Scan Phase):

In this phase, we scan through all the quadruples in the IR of the method that is being analyzed. The universal set **U** is populated in this phase. Each node in **U** is then further classified as a source node or a destination node and then **{S}, {D}** are accordingly populated.

Diagrammatically, this phase can be represented as in **Fig 2.2.1**.

So, at the end of this phase, we have fully populated **{S}** and **{D}** sets.

Also, we classify the nodes into new object node, reference load node, argument nodes (for invocation sites if any) and return nodes. So, we also have $\{\mathbf{N_o}\}$, $\{\mathbf{N_l}\}$, $\{\mathbf{N_a}\}$ **a**nd $\{\mathbf{N_r}\}$ fully populated.

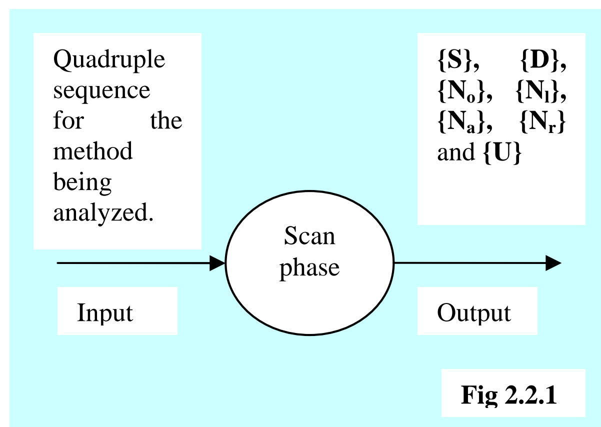

Fig 2.2.1

2.2.1(a) How loops are handled:

Loops are one special kind of programming constructs which result in cyclic control flow. It is but natural that such control flows need special treatment in any analysis done in programming languages. We could have any kind of the above listed nodes to be part of the loop body.

In general we could provide the following structure for a loop.

Begin Loop:

New object node
Return node
Argument node
Reference load node

End Loop:

If the structure is as simple as that then we would not need any special analysis. The default approach would suffice which we do in the algorithm.

However, if the looping is slightly more complicated then there are chances of getting misled due to the reverse control flow.

Consider a condition where, a reference load node, occurs prior to the object creation.

```
        LO1 = new Classx();
Begin Loop:
        LO2 = LO1
        Call xyz(LO2)
        LO1 = new Classy()
End Loop:
```

The above structure looks complex. If the graph or tree approach for analysis is employed, then we would need to take care of the control flow.

This is where the set theory approach is different. Since the nodes are initially appropriately categorized into sets, we just need to scan through the appropriate sets before deciding on if a node has escaped or not. There is not any chance of having potentially endless loops. This makes the entire algorithm and the implementation simpler.

2.2.2 Phase 2 (Analysis Phase):

In this phase, we initialize the various other sets that are necessary for our analysis. This includes the locally escaping node set **{N_{lo}},** the globally escaping node set **{N_{gl}}** and the non-escaping node set **{N_{ne}}.**

We initialize **{N_{lo}}, {N_{gl}}** and **{N_{ne}}** to Φ. Similarly, we initialize **{N_{in}}** and **{N_{out}}** to Φ.

$$\{N_{lo}\} = \{N_{gl}\} = \{N_{ne}\} = \Phi$$
$$\{N_{out}\} = \{N_{in}\} = \Phi \qquad \textbf{Eq 2.7}$$

Then, we analyze the actual flow of control in the program. During this we populate the **{N_{out}}** and **{N_{in}}** sets. We define a relation set **{R}**, between the source and destination sets **{S}** and **{D}.**

We define, relation set {**R**} as,

$$\{R\} = \{\ r_i : r_i < s_i,\ d_j > \text{ where } s_i \in S \text{ and } d_j \in D, \text{ for all } 0 <= (i, j) <= M\ \} \text{ where } M \text{ is the number of elements in } U \qquad \textbf{Eq 2.8}$$

Once **{R}** is populated then we analyze each element of this set with regard to the nodes that it connects.

According to the nodes $< s_i, d_j >$ that r_i connects, and the quadruple involved, we can different kind of relations.

Accordingly, the relations are as below.

- **Copy Relation:** Relation in which destination node d_j is equated to s_i.
  LO2 = LO1 AMOVE
- **Store Relation:** Relation in which destination node d_j is assigned as a field of s_i.
  LO2 = LO1 APUTFIELD x
  In simpler terms, LO1.x = LO2
- **Load Relation:** Relation in which destination node d_j is a loaded with a field of s_i.
  LO2 = LO1 AGETFIELD y
  In simpler terms, LO2 = LO1.y
- **Return value Relation:** Relation in which destination node d_j is the return value of a method that takes s_i as an argument.
  LO2 = M1(LO1)

Once the relations in the relations set **{R}** are classified as above, we apply the principles of escapeness to determine which objects have escaped. These are the below principles.

a. Any outside node is always treated as escaped node since we do not have the information regarding the history of the object prior to the invocation of the current method.

b. Any class variable globally escapes.

c. Object which is assigned to a static member of a class is treated as having escaped. This is denoted by the *copy relation.* If LO1 is a static member then LO2 globally escapes.

d. Object setup as a field of an escaped object always escapes. This is denoted by the *store relation.* If LO1 escapes then LO2 is also treated as an escaped object.

e. The *return value relation* is a slightly different kind of a relation because it involves invocation of another method. Under such scenarios, the arguments passed to the method being invoked may or may not escape depending on the below conditions.

i. If the method to be invoked is not yet resolved, then all arguments passed to this method are treated as *globally escaped.* This is because of the fact that for an unresolved method, the characteristics or escapeness behavior is unknown.

ii. If the method to be invoked is resolved, then check if the *escape summary* for this method is already available. If the *escape summary* is already available, then map and correlate the argument nodes from the caller method to the callee method. If a node in the

callee globally escapes and is mapped to a node in the caller, the the caller node globally escapes. If a callee node locally escapes, then the escape data for the corresponding mapped caller node needs to be computed

iii. If the method to be invoked is resolved and the *escape summary* is not yet available, then do a recursive escape analysis this method to obtain the *escape summary*.

Any node that globally escapes is added to the set of globally escaping nodes **$\{N_{gl}\}$.** Similarly, the **$\{N_{lo}\}$** and **$\{N_{ne}\}$** are populated.

The set theory equivalent of the above is as below.

Assume, *x* is a class variable. So, it is added to the set of globally escape nodes.

Then with intention of determining the escapeness of all the nodes in the method which is currently being analyzed, we consider each member r_i of the relations set **{R}.** Consider relation r_i as a relation between source s_i and d_j as below.

$$r_i = < s_i, d_j > \qquad \textbf{Eq 2.9}$$

Translating rules **a. to e** above, results in the following equations.

a. ***If $x \in \{N_{out}\}$, then $\{N_{lo}\} \cup x$*** **Eq 2.10**

b. ***If x is class variable, then $\{N_{gl}\} \cup x$*** **Eq 2.11**

c. ***If x is class variable, and y is another object such that $y \in \{N_{in}\}$, then, $\{N_{gl}\} \cup y$*** **Eq 2.12**

d. ***If $x \in \{N_{lo}\} \,||\, x \in \{N_{gl}\}$ and $y \in \{N_{in}\}$ and $r_i = < x, y >$, then $x \cup \{N_{lo}\}$ or $x \cup \{N_{gl}\}$ respectively.*** **Eq 2.13**

e. ***If $x \in \{N_{in}\}$ in the caller method, and is sent as an argument to the callee method then once of the below holds good.***

i. ***If callee not resolved then $\{N_{gl}\} \cup x$*** **Eq 2.14**

ii. ***If callee is resolved and escape summary for callee is Φ, then escape summary for callee is computed.***

iii. ***If callee is resolved and escape summary exists then the mapping is done as below***

If node x(caller) is mapped to node y(callee) and $y \in \{N_{gl}\}$ in the callee then $\{N_{gl}\} \cup x$ **Eq 2.15**

If node x(caller) is mapped to node y(callee) and $y \in \{N_{lo}\}$ in the callee then escape info for x needs to be computed. **Eq 2.16**

Analysing all the nodes in the method and their corresponding relation, we can finally populate the $\{N_{lo}\}$ and $\{N_{gl}\}$ sets.

Then using **Eq 2.5** we can compute the elements in $\{N_{ne}\}$.

Thus finally we have the escape summary data for all the objects nodes in the method being analysed.

2.2.3 Phase 3 (Cleanup or final Phase):

This phase is basically used as a phase to clean up the data structures involved. The *escape summary* information gathered in the above phase is then associated with the method block that was analysed.

3. Uses of Escape Summary:

The information that we gather by the above analysis can further be used to do other optimizations to improve the performance. Following are the uses of the same.

a. ***Stack Allocation:*** If an object is diagnosed to be a member of $\{N_{ne}\}$, then this cannot be accessed by any other location in the code. Hence this object can be allocated on the stack frame corresponding to the method that is being compiled. This reduces the associated overhead of storage component of the JVM controlling the object.

b. ***Eliminate Synchronization:*** In cases wherein an object has been allocated on a method's stackframe, we can rest assured that there will be no other thread that will be accessing the method. Hence, under such scenarios we can remove any synchronization operations on the object. This reduces the overhead of locking.
c. ***Reduced Object Layout:*** Also, generally, the heap allocated objects tend to have lot of fields in the object layout which are primarily necessary since they are on the heap. This includes the implementation specific fields which help the GC to control the lifetime of the object. Since stack allocated objects are not controlled by GC, we can even have a reduced object layout which can reduce the amount of storage needed.
d. ***Scalar Replacement:*** Also, under certain scenarios it is observed that certain objects remain captured in the method. But a certain field in the object is returned as a return value or passed to other programs for analysis etc. Under these circumstances wherein the field of the object is a scalar, we can eliminate the entire object itself. Instead we could use a simple scalar to replace the object's field in the generated code.

3. Conclusion:

This paper has explained a simple algorithm for Escape Analysis using basic Set Theory concepts. It is a simple to understand and easy to implement algorithm. The complications of implementation using more complex data structures are overcome by using the concept set theory and relations.

Due to the simplicity of the algorithm, the memory requirements are reduced and the time needed to do this analysis is reduced. Since just-in-time compilation takes time for compilation from the actual runtime of the application, it is necessary that we have an algorithm that takes fairly less amount of time for compilation yet producing efficient code.

The algorithm described above precisely does the same. It analyses the code using fairly less amount of memory, while it is capable of producing fairly efficient code. Implementations can further tune in order to restrict the analysis for a specific number of basic blocks, specific number of invocations, specific number of local variables etc. However, all this is specific to implementations.

3. Scope for enhancement:

The current algorithm necessitates that there be no references from the heap allocated objects to the stack allocated objects. This restriction is because of the fact that the lifetime of a heap object is generally greater than that of a stack object. Hence, if a heap object has a reference to a stack object, we might endup in a scenario where in the stack frame is no longer existent and hence the object on the stack frame has been removed. But the heap object might still hold a stale reference to the erstwhile stack object. There needs to be some mechanism to eliminate this limitation. Once this is eliminated, it is possible for a more exhaustive use of the Escape Analysis.

Since this algorithm recursively tries to compute the escape summary for each resolved method invocation, this analysis might become tedious in cases where in we have large code size and invocation tree. This is a possible disadvantage because of the fact that it could result in larger compile times. Since in dynamic compilation, the compilation time is a part of the actual execution time, larger values of compile times are not really appreciated. On the other hand, having complete escape information for all methods invoked might let us do a better object allocation and better performance. Hence, appropriate analysis needs to be done to reduce the dependency of the compile time on the code size and invocations.

4. References

Whaley, John and Rinard, Martin. *Compositional Pointer and Escape Analysis for Java Programs*. In Proceedings of Object-Oriented Programming Systems, Languages, and Applications. November 1999.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
IBM is a trademark of International Business Machines Corporation in the United States, other countries, or both.
Other company, product, or service names may be trademarks or service marks of others

****

This document was created with Win2PDF available at http://www.daneprairie.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.