Indian Institute Of Technology, Madras

Department Of Computer Science & Engineering

Undergraduate Research In Computer Science - I

Compile Time Memory Management

Chathurvedhi Talapaneni - CS20B021

under Dr. V. Krishna Nandivada

CHENNAI, JANUARY 2024

Contents

Abstract

The current memory management system in Java is handled by a garbage collector which reclaims objects based on reachability from the root set. This paper introduces an algorithm for compile-time memory management that allows objects to be freed at their last point of use during program execution rather than the point when it becomes unreachable. The algorithm involves an interprocedural flow-sensitive and value context-sensitive analysis along with interprocedural liveness analysis to determine the last usage point of an object using allocation abstraction.

1 Introduction

Garbage Collection is the primary memory management mechanism in Java which is responsible for automatically reclaiming memory occupied by objects to ensure efficient resource utilization. When identifying objects for potential reclamation, the garbage collector flags them as *garbage* if they're unreachable via any pointer traversal by the active program. This process preserves potentially reachable objects preventing the program from accessing deallocated memory through any pointer traversals.

However, garbage collection only collects the object when it is unreachable from the root set which is different from the potential earliest point during the runtime of the program it can be collected. Often due to long living objects during runtime, several other objects are determined reachable from the root set but are no longer used during the program execution. Prior works [3;4] measure the difference between the actual collection time and the potential earliest collection time of the objects indicating the potential memory savings. The results found in the paper: Heap profiling for Java [4] show potential for space savings from 23% to 74% for the SPECjvm98 benchmark suite if objects could be collected at their last use.

This paper goes over the previous works in compile-time analysis for liveness-based memory management for Java and presents a static analysis to insert viable *free* calls while ensuring program soundness. The algorithm first performs an interprocedural flow-sensitive and value context-sensitive analysis to determine the points to information at each program point. Then it performs an interprocedural liveness analysis to determine the last usage point of an object using allocation abstraction. Given an input Java program, our algorithm produces an output program with *free calls* inserted in the program.

2 Background and Related Work

2.1 Region-based analysis

Region-based systems [1;5;6] group heap objects together in regions and deallocate all of the objects in a region together. This approach is motivated by several advantages. First, it may improve data locality. Second, it can provide performance gains by deallocating entire regions as a whole rather than individual objects. Third, if used in conjunction with static analysis, it provides automatic memory management.

The paper by Cherem and Rugina [1] presents a region analysis and transformation system for Java programs. Given an input Java program, it produces an output program augmented with region annotations.

The program augmentations include:

- Creating regions: `create r;`
- Removing regions: `remove r;`
- Object Allocations: `new C(e1,..,en) in r;`
- Method Declarations: `T m<r1,..,rk>(T1 p1,..,Tn pn) ..`
- Method Calls: `m<r1, . . . , rk> (e1, . . . , en);`

The program analysis and transformation occurs in 3 steps:

- Flow-insensitive and Context-sensitive Region-based points-to analysis
  - A Points-to graph is created for each method declaration with nodes representing regions and field edges between nodes.
  - This step determines the number the region for each allocation site and region parameters for each region.
- Flow-sensitive Region Liveness Analysis
  - Perform a liveness analysis to determine the live regions at each program point.
- Region Translation
  - Utilizing liveness information, add `create r and remove r` and region parameters.

Limitations

The performance of the algorithm is close to GCs and performs well on programs with short-lived objects and successfully deallocates regions based on liveness. However, the algorithm performs poorly on programs with long-lived objects. The algorithm also does not deal with all possibilities of objects being reachable but no longer used after the program point as shown in section 3.

2.2 Free-Me Static Analysis

The paper by Guyer, McKinley and Frampton[2] performs lightweight pointer analysis with liveness analysis to add *free* calls to garbage-collected systems. The analysis determines the points where short-lived objects die and add appropriate *free* calls into the program.

The program analysis and transformation occurs in 3 steps:

- Flow-insensitive pointer analysis
  - Intraprocedural analysis representing the local objects in the heap model.
  - Builds a connectivity graph with each allocation site, input parameters and one node representing all globals.
  - Simple method summary for each method giving information about how the method connects the objects passed to it as parameters.
- Flow-sensitive pointer analysis
  - Traditional liveness analysis to determine live variables and objects on each edge of the control flow graph.
- Allocating free calls
  - Determining the earliest program point to deallocate objects in the program.

Limitations

This analysis also has similar limitations as the region-based analysis. The analysis primarily observes gains in dealing with only short-lived objects.

3 Motivating Example

The mentioned works above are prior attempts at automatic memory management that have their benefits, however, they do not reclaim all the objects that are reachable but are no longer used after a certain program point. The example presented below shows a simple Java program.

```java
class DualNode
{
    Node head1;
    Node head2;
};

class Node
{
    int data;
    Node(int item)
    {
        data = item;
    }
};

public class example {
    public static void main(String Args)
    {
        DualNode dual = new DualNode();      - O1
        dual.head1 = new Node(5);            - O2
        dual.head2 = new Node(10);           - O3
        PrintList1(dual);                    - Call 1
        PrintList2(dual);
    }

public static void PrintList1(DualNode dual)
{
    Node currNode = dual.head1;
    System.out.println(currNode.data);       - Last usage point
}
public static void PrintList2(DualNode dual)
{
    Node currNode = dual.head2;
    System.out.println(currNode.data);
}
};
```

Here we see a case where we have a program point after which the object O3 is no longer used but is reachable until the end of the program. As we can see the last usage is in the statement `Node currNode = dual.head1;` in the `PrintList1(dual);` call. Here we have the potential to free the object O3 after the statement for efficient memory management.

Region-based analysis [1] performs flow-insensitive points-to analysis of functions and is unable to determine the last usage of an object within a callee function. Thus it is unable to determine that O3 can be freed at the last usage point in **Call 1** of `PrintList1()`. Similarly, Free-Me analysis [2] deals with the deallocation of short-lived objects using flow-insensitive points-to analysis. Both compile-time memory management techniques are incapable of handling scenarios like the one demonstrated above.

Let us take a modification of the first example to highlight the problems with implementing compile-time memory management.

```
public class example {
    public static void main(String Args)
    {
        DualNode dual = new DualNode();      - O1
        dual.head1 = new Node(5);            - O2
        dual.head2 = new Node(10);           - O3
        PrintList1(dual);                    - Call 1
        PrintList1(dual);                    - Call 2
        PrintList2(dual);
    }

public static void PrintList1(DualNode dual)
{
    Node currNode = dual.head1;
    System.out.println(currNode.data);       - Last usage point
}
public static void PrintList2(DualNode dual)
{
    Node currNode = dual.head2;
    System.out.println(currNode.data);
}
};
```

Here we face the problem where the last usage point for the object O3 is during the same statement during the **Call 2**. In the previous example, we could free the object by inserting a *free* call in the method definition of `PrintList1()` but now we must insert the *free* call right after *Call 2* to ensure program soundness.

4 Algorithm

The algorithm consists of mainly 3 steps.

- Flow-Sensitive and Value Context-Sensitive Points-to Analysis
- Reference Liveness Analysis
- Freeing the object

4.1 Points-to Analysis

First, we perform an interprocedural flow-sensitive and value context-sensitive points-to analysis over the entire program. Along with this, we develop a control flow graph identifying each function call and linking them to the appropriate points-to analysis.

From this, we have the points-to information after every statement in the entire program. The information is represented using an abstract stack(ρ) and an abstract heap(σ).

Let V be the set of variables, R be the set of references, F be the set of all fields and P be the power set.

$$\rho : V \to P(R) \tag{4.1}$$

$$\sigma : R \times F \to P(R) \tag{4.2}$$

This information will be used to determine the last usage point of the references in the Liveness Analysis.

4.2 Liveness Analysis

The main objective of the liveness analysis is to determine the last usage point of all references.

We perform backward liveness analysis but instead of keeping the set of live variables, we maintain a set of references that are live at each program point. We start the analysis from the end of `main` and progressively move backward updating the live set based on given rules.

Let the live set of references be called *Ref Set* and is initialized to an empty set at the start of the analysis at the end of `main`.

4.2.1 Rules for Analysis

The below table represents the *Use Set* for the simple statement set in Java.

Statement	Use Set
`x = new A()`	-
`x = y`	$\rho(y)$
`x = y.f`	$(\cup[\sigma(o, f)]\ \forall o \in \rho(y)) \cup \rho(y)$
`x.f = y`	$\rho(y)$

The *Def Set* is only defined for the statement `x = new A()` and it is the object linked to the allocation abstraction. Utilizing the *Def Set* and *Use Set*, we can build the *Ref Set* at every program point.

Let Ref_s be the *Ref Set* after the statement s and Ref_s^1 be the *Ref Set* before the statement. Let Use_s and Def_s be the *Use and Def Set* for the statement s. For updating the *Ref Set*:

$$Ref_s^1 = (Ref_s - Def_s) \cup Use_s \tag{4.3}$$

From this analysis, we can ascertain the last usage point of an object. For a given statement s, the set of all objects that can be freed after the statement is:

$$Ref_s^1 - Ref_s \tag{4.4}$$

Let us now go over the rules for the analysis of other statements such as function calls, if-else cases, and while loops.

4.2.2 If-Else case

Let us take the general case of encountering multiple paths due to if-else loops and call them *L1, L2, L3, ..., Ln.*

The analysis performs the basic liveness analysis over all the possible control paths to determine the *Ref Set* at the start of each control path. Let the *Ref Sets* at the start of all the paths be:

$$Ref_S = [Ref_1,\ Ref_2,\ Ref_3,\ ....,Ref_n] \tag{4.5}$$

The *Ref Set* before all the control paths can be defined as:

$$\cup[Ref]\ \forall\ Ref \in Ref_S \tag{4.6}$$

Let us take an example to show backward analysis in if-else cases and determine the last usage points of objects.

```
// Ref Set = {O1, O2, O3, O4, O5, O6, O7}, Union of Ref Sets of all
   paths
if(cond1) // L1
{
// Ref Set = {O1, O2, O3, O4}
.
// Ref Set = {O1, O2, O3}
}
else if(cond2) // L2
{
// Ref Set = {O1, O2, O3, O5}
.
// Ref Set = {O1, O2, O3}
}
else // L3
{
// Ref Set = {O1, O2, O3, O6, O7}
.
// Ref Set = {O1, O2, O3}
}
// Ref Set = {O1, O2, O3}
```

In this given example, the *Ref Sets* are calculated in each path.
Deaths detected along individual paths are as follows

- L1 - O4 can be freed
- L2 - O5 can be freed
- L3 - O6, O7 can be freed

Thus, along with placing the appropriate free calls, we must also free the other objects along the different paths as shown below.

```
// Ref Set = {O1, O2, O3, O4, O5, O6, O7}, Union of Ref Sets of all
    paths
if(cond1) // L1
{
// Free O5. O6, O7
// Ref Set = {O1, O2, O3, O4}
.
// Ref Set = {O1, O2, O3}
}
else if(cond2) // L2
{
// Free O4, O6, O7
// Ref Set = {O1, O2, O3, O5}
.
// Ref Set = {O1, O2, O3}
}
else // L3
{
// Free O4, O5
// Ref Set = {O1, O2, O3, O6, O7}
.
// Ref Set = {O1, O2, O3}
}
// Ref Set = {O1, O2, O3}
```

4.2.3 While-Loop case

In the analysis of a while loop, we must run multiple iterations of liveness analysis to determine the last usage points of objects.

- First iteration: Determines the objects that can be freed after the loop.
- Multiple iteration:
  - After the first iteration, we must run multiple iterations using the *Ref Set* at the start of the loop.
  - The process is repeated until there is no change in the *Ref Set* at all program points in the loop.
  - This final iteration determines the objects that can be freed in the while loop.

Let us take an example to show the analysis:

```
// x.f points to O4
while(cond)
{
    y = new A(); // O5
    .
    Use(x.f)
    .
    Use(y)
    .
}
// Ref Set = {O1, O2, O3}
```

First Iteration:

```
while(cond)
{
    // Ref Set = {O1, O2, O3, O4}
    y = new A();
    .
    // Ref Set = {O1, O2, O3, O4, O5}
    Use(x.f)
    // Free O4
    .
    // Ref Set = {O1, O2, O3, O5}
    Use(y)
    // Free O5
    .
}
// Ref Set = {O1, O2, O3}
```

Final Iteration:

```
while(cond)
{
    // Ref Set = {O1, O2, O3, O4}
    y = new A();
    .
    // Ref Set = {O1, O2, O3, O4, O5}
    Use(x.f)
    .
    // Ref Set = {O1, O2, O3, O4, O5}
    Use(y)
    // Free O5
    .
    // Ref Set = {O1, O2, O3, O4}
}
```

From the analysis, we can place the *free* points as follows:

```
while(cond)
{
    y = new A(); // O5
    .
    Use(x.f)
    .
    Use(y)
    // Free O5
    .
}
// Free O4
```

4.2.4 Function Calls

For analyzing function calls, the analysis uses the interprocedural points-to analysis. To demonstrate the analysis, let us use an example:

```
main()
{
    .
    foo(a,b,c); //Call 1
    // Ref Set = {O1, O2, O3}
    .
    foo(a,d,c); //Call 2
    // Ref Set = {O1, O2}
    .
}
foo(int x, int y, int z)
{
    .
    .
}
```

Each function call is analyzed based on value and liveness context. In the given example, both call 1 and call 2 have different analyses.

```
foo(int x, int y, int z) // Call 1 Analysis
{
    .
    .
    // Ref Set = {O1, O2, O3}
}

foo(int x, int y, int z) // Call 2 Analysis
{
    .
    .
    // Ref Set = {O1, O2}
}
```

Ref Set calculated in the function analysis at the top of the callee function is used before its function call in the caller function.

4.3 Freeing the objects

Now that we have the object deaths for the entire program which are placed in several positions in different analyses of function, we must use this information to produce an output program that frees the objects appropriately while maintaining program soundness.

Object death to `free(x)` call: As the analysis provides us with the location of object deaths in terms of allocation abstraction, we must convert them into `free` calls.

Say after a statement s, we have an object death. We first check the below condition:

$$Ref_s^1 - Ref_s == Use_s \tag{4.7}$$

- **True:** We can place a `free(z)` call where z is the R.H.S of the statement s.
- **Not True:** We must perform a graph traversal using the points-to information available to insert a `free(z)` call where z is the largest subset of $Ref_s^1 - Ref_s$. Currently, the analysis can perform a brute force approach to find the largest subset, but we must develop a more time-efficient algorithm.

Multiple analyses of a function: During the program analysis, we could analyze a given method with a different value context or liveness context which could lead to one of the function analyses having a program point where it had an object death but the other analyses don't have an object death. Let us demonstrate using an example.

```
main()
{
    .
    // x points to O1, x.f points to O2
    foo(x);          // Call 1
    foo(x);          // Call 2
    .
}

foo(node x)          // Call 1 analysis
{
    .
    Use(x.f)
    .
}

foo(node x)          // Call 2 analysis
{
    .
    Use(x.f)
    // Free O2
    .
}
```

Here we can see that both analyses do not have the same object death calls. Thus here, we must attempt to free the object *O2* after their respective caller functions. The final position to free *O2* can be placed after *Call 2* as shown below.

```
main()
{
    .
    foo(x);          // Call 1
    foo(x);          // Call 2
    // Free O2
    .
}
```

Freeing the objects: We must convert the object deaths in terms of `free(x)` and build a runtime support. For this implementation, we can take inspiration from the techniques used in the Free-Me analysis paper[2] to integrate the freeing of objects with a garbage collector.

5 Conclusion and Future Work

The paper presents a program analysis technique that determines the last usage of an object using allocation abstraction. From the last usage points, the program can be modified to introduce *free* calls. However the algorithm currently only shows the last usage of objects demonstrated as assigned to allocation points and attempts to make the object unreachable from the root set to allow garbage collection.

Graph Traversal Algorithm: Currently, to convert object deaths to `free()` calls, we use a brute force approach on traversing the points-to information available to ascertain the appropriate statement. But we can improve the analysis by implementing a more efficient graph traversal algorithm to significantly shorten analysis time.

Runtime Support: One direction to expand on the current algorithm is to enable runtime support for *free* calls taking inspiration from the Free-Me analysis paper[2]. With this, we can efficiently free the objects in tandem with the garbage collector.

Implementation: The algorithm must be tested using recent benchmarks in terms of performance, memory management and analysis time. The Soot Framework can help implement this analysis in the compilation process.

6 Acknowledgements

The work done during this course was possible due to the support and invaluable guidance provided by Dr. V. Krishna Nandivada. The culmination of this research was made possible through extensive support and profound discussions with Dr. V. Krishna Nandivada. Additionally, I would like to express my gratitude to Dr. Mitesh M. Khapra for coordinating the Undergraduate Research Course.

References

[1] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, page 85–96, New York, NY, USA, 2004. Association for Computing Machinery.

[2] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-me: A static analysis for automatic individual object reclamation. *SIGPLAN Not.*, 41(6):364–375, jun 2006.

[3] Niklas Röjemo and Colin Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *ACM SIGPLAN International Conference on Functional Programming*, 1996.

[4] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient java. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 104–113, New York, NY, USA, 2001. Association for Computing Machinery.

[5] Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. Safe and efficient hybrid memory management for java. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, page 81–92, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Xi Wang, Zhilei Xu, Xuezheng Liu, Zhenyu Guo, Xiaoge Wang, and Zheng Zhang. Conditional correlation analysis for safe region-based memory management. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 45–55, New York, NY, USA, 2008. Association for Computing Machinery.