Improving Liveness Based Garbage Collector in Java

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Bachelor of Technology - Master of Technology

by
Nikhil Pangarkar
Roll No. : Y9227374

under the guidance of
Prof. Amey Karkare

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

July, 2014

CERTIFICATE

It is certified that the work contained in this thesis entitled *"Improving Liveness Based Garbage Collector in Java"*, by *Nikhil Pangarkar(Roll No. Y9227374)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 15/7/2014

(Prof. Amey Karkare)
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur
Kanpur-208016

July, 2014

Abstract

Garbage collection is an important feature of Java which prevents memory exhaustion, combat heap fragmentation and provides a more secure way to use memory by the program. It frees up the programmer from manual memory management which can lead to memory leaks that are difficult to catch. An important part of garbage collection algorithms is how to identify if a memory is in use or not. Typical garbage collectors use reachability to approximate the heap memory that could be used in subsequent execution of the program. But this conservative approach sometimes doesn't reclaim all unused memory. Liveness of a variable can provide a more tighter bound than reachability on memory use by an executing program.

In this thesis we explore the possibility of using a liveness based garbage collector for Java. We tackle the problem of providing liveness information which is obtained at compile time to a garbage collector which is a runtime module. We model the tracing problem on CFL reachability and offer some improvements to the liveness based tracing algorithm. We have been able to obtain a significant improvement in the running time of the liveness tracing algorithm as compared to the naive algorithm used previously.

Dedicated to
my parents and my brother.

Acknowledgement

I would like to express my sincere gratitude towards my thesis supervisor Dr.Amey Karkare for his constant support and encouragement. I am grateful for his patient guidance and advice in giving a proper direction to my efforts.
I would also like to thank the faculty and staff of the Department of CSE for the beautiful academic environment they have created here.
I am indebted to all my friends for making these past five years a memorable one for me. I especially thank Vilay for his valuable input in my thesis.
Last, but not the least, I would like to thank my parents and siblings for their love, constant support and encouragement. Without their support and patience this work would not have been possible.

Nikhil Pangarkar

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

An important component of programming languages which run in a virtual execution environment is a Garbage Collector. Typically memory in heap that is no longer referenced is ’garbage’ and thus can be collected to allocate new objects in its place. This process is called Garbage Collection. Garbage Collection or memory-recycling typically handles memory leaks and is also responsible for heap de-fragmentation.

The perfect garbage collector(GC) would collect all objects that are not going to be used during the rest of the execution of the program. But it’s impossible to predict run-time scenarios. One popular conservative approach to garbage collection is to compute all the objects that are reachable from a root set of variables that are still currently in the scope of the program and earmark them. The rest of the heap is marked as garbage and is ready to be freed and used in allocation of objects. This leaves potentially large amount of garbage behind in the form of objects that are reachable but not live. Liveness based garbage collector which analyze the whole program for liveness data can potentially collect more garbage than reachability based garbage collector.

In this thesis we discuss an approach to use liveness information of the program to design a garbage collector for Java.

1.1 Motivational Example

Let us look at two scenarios which highlight the improvement offered by liveness based garbage collection.

Program 1: Reachable variables not live

```
1 public static void main(String[] args)
2 {
3   Tree x = new Tree();
4   doSomething(x);
5   Tree y = new Tree();
6   doSomethingElse(y);
7 }
```

In the simple example Program 1 above, we can observe that we are allocating a $Tree$ object which is referenced by the variable x then doing some computation over it and then allocating a new object which is referenced by y and performing some other computation over it. If the garbage collector is triggered at line number 5 then a reachability based GC would not be able to reclaim the memory occupied by the object referenced by x because the variable x is in the current local variable set. In other words since x is still in the current scope of the program it can't be collected. A liveness based GC would identity that x is not a live variable at line number 5 and the object would be freed to reclaim more memory.

Second example highlights the case when an object in heap is partially live. Program 2 shows a simple program and Figure 1.1 shows a snapshot of the heap memory at any program point after line number 4. The red path represents object references that can be accessed during the program execution.

Only the objects reachable via red edges are live. The rest can be garbage collected using a liveness based garbage collector.

Program 2: Partially live objects

```
public static void main(String[] args)
{
  // Let x be complete binary tree of depth 4
  BinaryTree x = new BinaryTree();
  BinaryTree p = x;
  while(p.left!=null) {
    p = p.left;
  }
  doSomething(p);
}
```

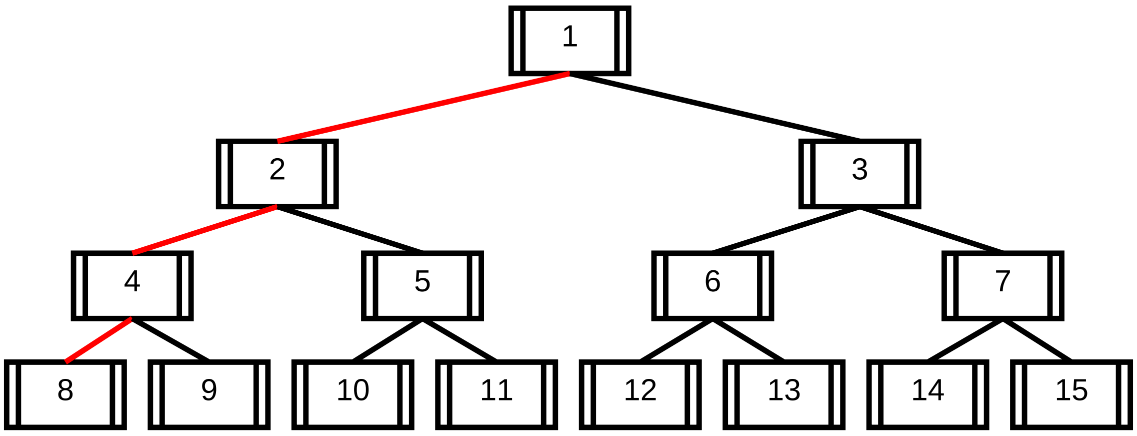


Figure 1.1: Example 2: Heap snapshot

1.2 Contribution of this thesis

The liveness data of a program is available in the form of access graphs for all the local variables which are live at a particular program point. These access graphs outline the sub-structure of the object which is live. In this thesis we describe an efficient algorithm to trace heap structure in a garbage collector according to the access graph of the object. We use JikesRVM, a research virtual machine for Java, to implement the algorithm and compare its performance with a naive approach and traditional reachability garbage collector approaches.

1.3 Outline of the Thesis

Chapter 2 describes prior work done in this field and describes the relevant portions of JikesRVM, the Java Virtual Machine used for this project. Chapter 3 describes the algorithm, it's implementation details and provides some theoretical analysis for the algorithm. Results are tabulated in Chapter 4 which contains comparison of the algorithm's performance with other approaches. Finally we conclude with some pointers towards future improvement of the project in Chapter 5.

Chapter 2

Background

In this chapter first we describe an inter-procedural liveness analysis for heap in a Java like imperative programming language. Subsequently we will describe briefly a garbage collector that would serve as the foundation for our improved algorithm.

2.1 Related Work

M. Hirzel et al. in [7] analyze and compare the improvement offered by type accuracy and liveness accuracy in the context of garbage collection for C and C++ languages. *Type accuracy* refers to the ability of the garbage collector to distinguish between references and non-references whereas *liveness accuracy* refers to the ability to identify live-references. Using a modified BDW garbage collector [5] their study found upto 62% reduction in heap size when using an inter-procedural liveness analysis to aid in garbage collection.

O.Agesen et al. [3] in a similar study for Java have found that adding a live variable analysis to enhance the accuracy of a garbage collector reduced the heap size by about 11%. They found that for most programs the improvement is not significant but it does avoid situations where the program uses surprisingly more heap space than required.

R. Asati et al. [4] provide a liveness analysis for a first-order functional language followed by a mechanism to summarize the liveness information into context free grammars (CFGs). They translated the CFGs into a finite state automata which was used to tighten up marking during garbage collection. They also proceeded to prove that a liveness based collector cannot do more collections than a reachability based collector.

U. P. Khedkar et al. [12] offer an improved liveness analysis to identify dead objects and explicitly set their references to *null* which is a technique called "Cedar Mesa Folk Wisdom" [6]. In our work we have used the explicit live reference analysis to achieve better garbage collection by using the analysis output directly in the garbage collector rather than setting references *null*.

2.2 Explicit Live Reference Analysis

Heap reference analysis [12] presents a method to summarize liveness data for objects. The access paths to all the live objects reachable from a variable is translated directly into an access graph. This access graph is similar in structure and function to a finite state automaton.

The access graph or the automaton has one start node. This node represents the object referenced by the variable in the program. Each outgoing edge of the graph represents a field of the object. Every node visited by the automaton will be accessed in the program and thus it is not to be garbage collected. Let us consider the simple example given below.

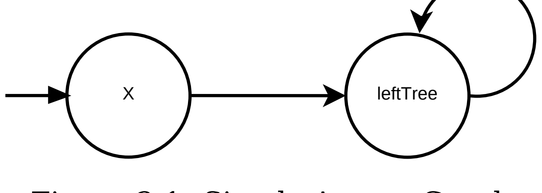


Figure 2.1: Simple Access Graph

In the heap we have a binary tree. Suppose figure 2.1 represents its access graph. Each node of the tree has two fields $leftTree$ and $rightTree$. In the given automaton Node X represents the node of the tree which is referenced by the variable X in the program. If we traverse the binary tree in accordance with the automaton we will visit all the nodes of the binary tree along the path from the X node to its left-most child. Every other node is garbage and hence can be reclaimed by the garbage collector. Thus the sub-structure of an object that is live at a particular program point is captured in the form of a finite state automaton.

2.3 Implementing Liveness Based Garbage Collection

LiveGC, a liveness based garbage collector is introduced in the concomitant thesis [11]. It uses Soot implementation of the heap reference analysis mentioned in the section above.

Soot [14] is an open source optimization framework for Java developed by Sable Research Group. Soot provides four intermediate representations for Java namely, Jimple, Baf, Shimple, Grimp. These intermediate representations allow different APIs for various optimization techniques. SOOT framework [2] allows users to write various data-flow analyses.

JikesRVM is an open source Java Virtual Machine used for research in various virtual machine technologies. It is one of the most extensive virtual machines available. A major feature of JikesRVM is that its written almost completely in Java language itself. In this project we have extended JikesRVM to implement a new liveness based garbage collector and then analyze its performance with respect to other traditional garbage collection techniques.

One of the major challenge in implementing a liveness based garbage collector is providing compile time liveness information to garbage collector which is a runtime module. The LiveGC garbage collector uses the following design to achieve that goal.

LiveGC Design The Soot implementation of Heap reference analysis is responsible for computing access graphs. The liveness information for program points that invoke garbage collector is stored in Java objects(access graphs) which are stored in separate files using serialization. These files contain the access graphs of all the live variables at the program point. The JVM would de-serialize the files to get the access graphs.

Secondly a new bytecode instruction is inserted in the bytecode which provides the list of object references that are live and invokes the JVM routine to handle collection.

At runtime the JVM thus contains a list of all the live variables and their corresponding access graphs. The garbage collector then does a simple traversal of the heap in accordance with the access graphs.

In the next chapter we will discuss the tracing problem and describe an algorithm which improves the running time performance of the heap trace in LiveGC.

Chapter 3

Tracing Algorithm Design

In the previous chapter we described briefly the basic design of our solution to the problem of liveness based garbage collection. The next step in the process is to trace the heap memory in accordance with the liveness data. In this chapter we will first look at a classical problem over which we modeled the tracing problem and then discuss the algorithm for liveness based tracing of heap.

3.1 CFL Reachability

A reachability problem in the context of graph analysis deals with answering questions such as, is a particular node A reachable from node B along any path in the graph. In CFL reachability [13] we consider only those paths such that the concatenation of labels on the path edges belong to a particular context free language.

Definition 1. A CFL reachability problem (G, L, Σ) is defined as follows. We have context free language L over alphabet Σ and graph G whose labels are alphabets from Σ. A path in G is an $L - path$ if the word obtained after concatenating the labels of the path belongs to the language L. Two nodes are considered reachable if there is an $L - path$ connecting them.

Let us look at an example where the graph is as given in Figure 3.1 and the CFG is a simple context free language L for valid bracketed expression. Starting from A consider the path $ABCGH$ and $ABCDEFBCGH$. Concatenating the labels along both the paths yields a valid bracketed expression which belongs to the language L. Therefore both of them are $L - Paths$. Also note that only H is reachable from A via an $L - Path$

Expr	$\rightarrow$	Expr Expr
	|	'{' Expr '}'
	|	'[' Expr ']'
	|	'(' Expr ')'
	|	'e'
	|	ϵ

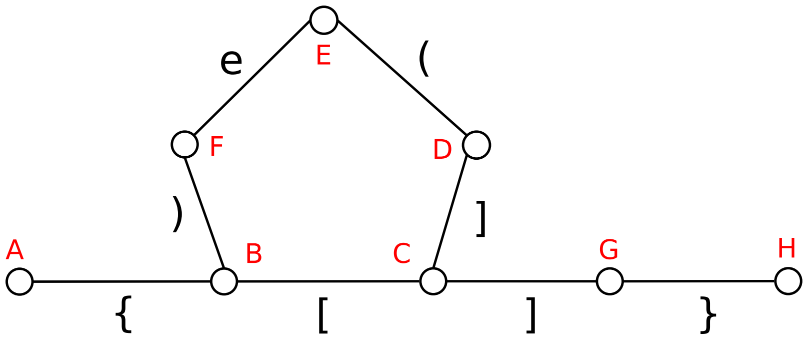


Figure 3.1: CFL Reachability Example

There are 4 types of CFL-reachability problems:

1. *all pairs L-path problem:* Finding all pairs of nodes n_1 and n_2 connected by an L-path

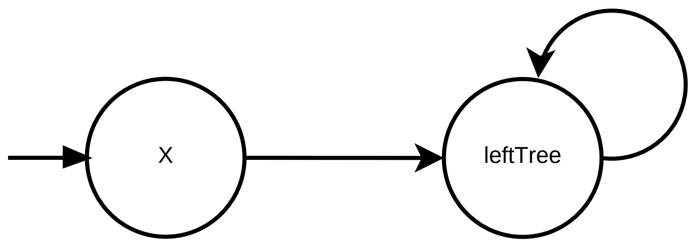
2. *single source L-path problem:* Finding all of nodes n_2 such that there is an L-path from from a particular node n_1 to n_2

3. *single target L-path problem:* Finding all of nodes n_1 such that there is an L-path from n_1 to a particular node n_2

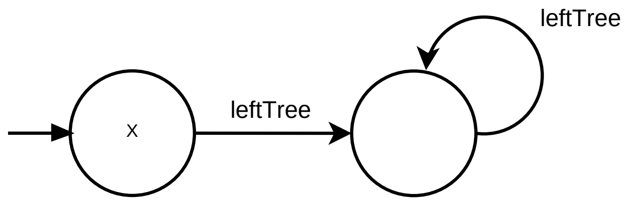
4. *single source single target L-path problem:* Finding if there exists an L-path from n_1 to n_2

3.2 Tracing Algorithm

As described in the previous chapter, using heap reference analysis we can compute the access graphs for each live variable. Concatenating node labels along any particular path starting from root node gives a valid access path. All paths starting from the root node are a conservative approximation of all access paths that could be used during the program execution. Take the example access graph 3.2a below.



(a) Simple Access Graph



(b) Equivalent Automata

Figure 3.2: Access Graph and NFA

All the paths from root node x are of the format $x.leftTree$, $x.leftTree.leftTree$, $x.leftTree.leftTree$ etc. Which can also be represented by the regular expression $x.leftTree*$. Since CFG are strictly more powerful than regular expression we can say that the access graph in this example can be encoded into a CFG.

Another important thing to note about the access graphs is that they are directed graphs and there can be self edges and loops in the graph but it has to have a unique start edge which has no incoming edge. The access graphs can be translated to equivalent Non-deterministic finite(NFAs) automata by a simple method. If we label all the edges of the graph with the label of the destination node of the edge we can get a NFA which is shown in 3.2b. NFAs can be converted to regular languages which in turn are subset of context free languages [8]. Therefore we can conclude that the access graphs are encoded context free languages.

Our goal is to find all nodes in the heap that lie on some valid path on the access graph. This is equivalent to finding all objects on heap whose reference paths belong to the CFL encoded by the access graph. Thus the problem reduces to single source L-path problem. By finding all the nodes reachable from root node via a valid access path and all the nodes on these paths we complete our objective.
We propose the following algorithm for the problem which is a modified DFS algorithm.

Algorithm 1: Garbage Collector Algorithm

```
Function GCTrace(Object[] LiveObjs, AccessGraph[] GraphList)
    for $index \leftarrow 0$ to LiveObjs.size() do
        X = LiveObjs[$index$]
        G = GraphList[$index$]
        root = G.rootNode()
        TraceObject(X, G, root)
```

The input to the garbage collector is a list of live variables and their corresponding access graphs. The garbage collector iterates over the list and handles tracing for each object through the function GCTrace. For each object we invoke the function TraceObject.

Algorithm 2: Tracing Algorithm for an Object

```
Function TraceObject(Object X, AccessGraph G, GraphNode i)
    label [X, i] = visited
    for each edge (i, v) in G do
        if X has a field v: then
            Object f = X.v
            if label [f, v] != visited then
                TraceObject(f, G, v)
```

For each object X we have a corresponding access graph G. A 2D array $label$ keeps track of the nodes visited and the state of the automaton when the node was visited. $label[X, i] = visited$ represents object X was visited and the state of the automaton was i. Then we iterate through all the outgoing edges from the state i of the automaton. If an outgoing edge of the automaton is present in X (by present in X we mean that there is a field of X with the same name as the label of the outgoing edge) and its not yet visited via that state of the automaton then we traverse that node recursively. Every node visited by TraceObject is marked so that the garbage collector during the collection phase doesn't collect it as unused memory.

3.3 Implementation

We used JikesRVM, a Java virtual machine written in Java, for implementing our algorithm. The code is implemented in the `org.jikesrvm.mm.mmtkinterface` package in the `GCMonitoration` class.

The TraceObject algorithm discussed in the previous section is a recursive function. This could lead to stack overflow exception for data structures that have a large depth. To counter this the algorithm was translated to an equivalent iterative procedure using a Stack. The final algorithm is shown below.

Algorithm 3: Optimized Tracing Algorithm

```
Function TraceWithGraphOpt(Object X, AccessGraph G)
    Map map = new HashMap()
    objectStack.push(X)
    nodeStack.push(G.root())
    while !objectStack.isEmpty() do
        ProcessNode(G,map)

Function ProcessNode(AccessGraph G, Map map)
    object = objectStack.pop()
    root = nodeStack.pop()
    label = map.get(object)
    label[root.index()] = visited
    for field f in object.fields do
        for target t in root.targets do
            if f == t then
                child = f.loadObject()
                label = map.get(child)
                if label[t.index()] != visited then
                    objectStack.push(child)
                    nodeStack.push(t)
                    label[t.index()] = visited
                    map.put(label)
```

Chapter 4

Results and Analysis

Experiments carried out with LiveGC without the optimized algorithm descibed in this thesis has been discussed in [11]. The results show a performance increase in terms of the number of objects collected but the running time performance is not encouraging. We have tested our algorithm against the same set of benchmarks and compare its performance with naive method and also with a reachability based trace.

For the sake of brevity we use LiveGC to refer to the liveness based GC used in [11]. The optimized version discussed in this thesis would be referred to as LiveGCOpt and the reachability based GC would be referred as RGC. We used 5 benchmarks - BiSort, Loop, DLoop, Reverse and TreeAdd.

4.1 Results

BiSort This benchmark contains a program that sorts using bitonic sort algorithm [10]. The program creates a large binary tree and sorts them so that the inorder traversal of the values is sorted. Results are shown in 4.1.

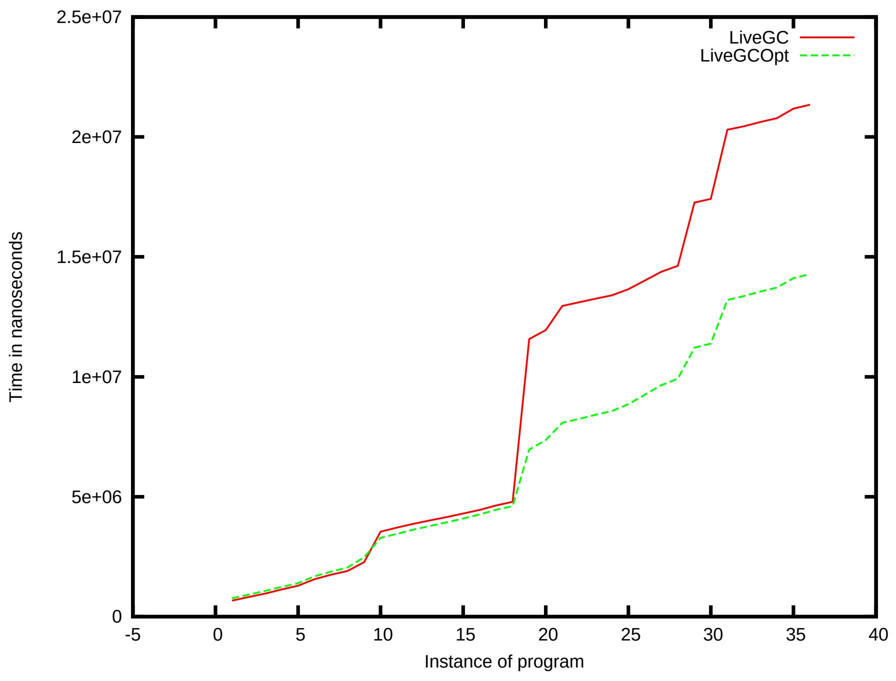


Figure 4.1: BiSort Program

Loop and Reverse Loop consists of large singly linked lists. The program then traverses them starting from one end. The part of the list already traversed represents garbage nodes which is captured by LiveGC and not by RGC. The Reverse program reverses a single linked list. Results for the three programs are shown in 4.2 and 4.3.

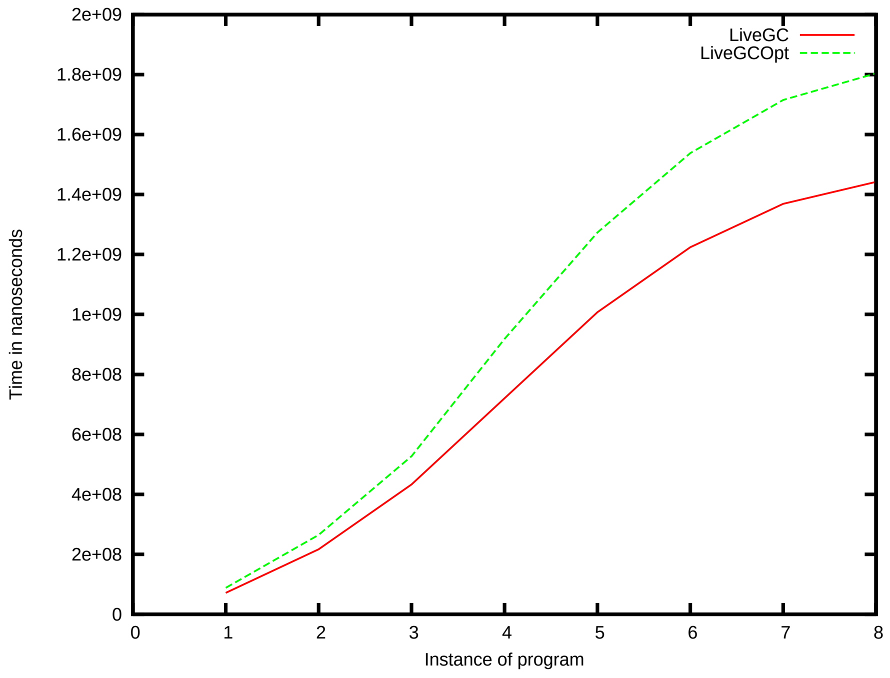


Figure 4.2: Loop Program

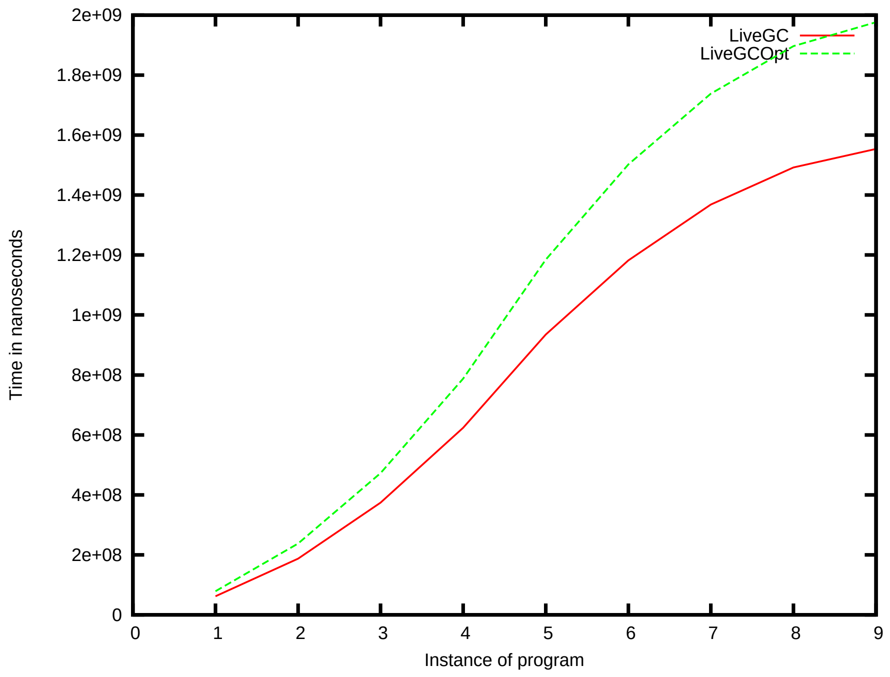


Figure 4.3: Reverse Program

BST and TreeAdd TreeAdd computes the number of nodes in the tree using a recursive algorithm. Results for BST are shown in 4.4 and for TreeAdd are shown in 4.5.

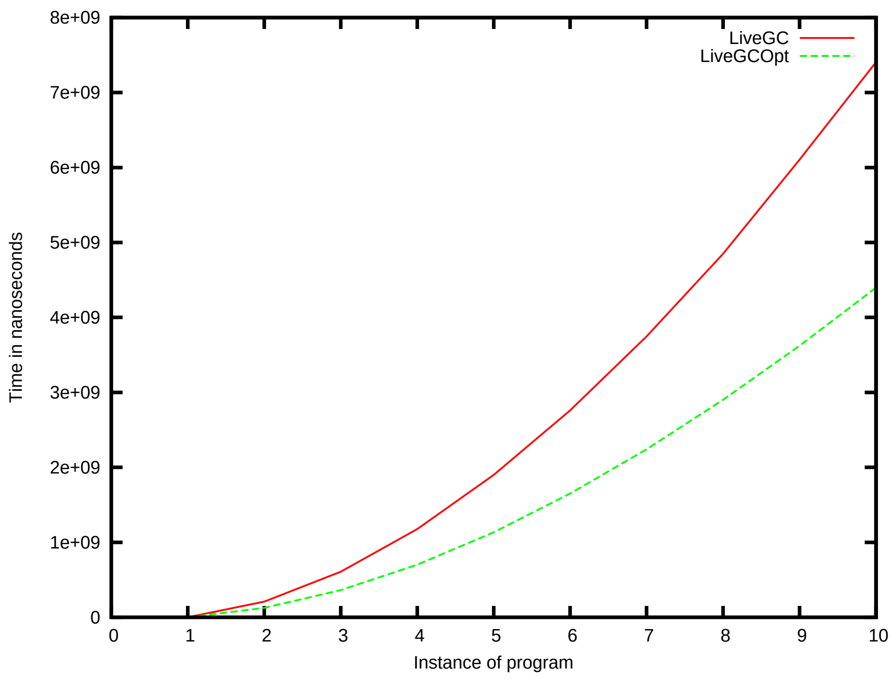


Figure 4.4: BST Program

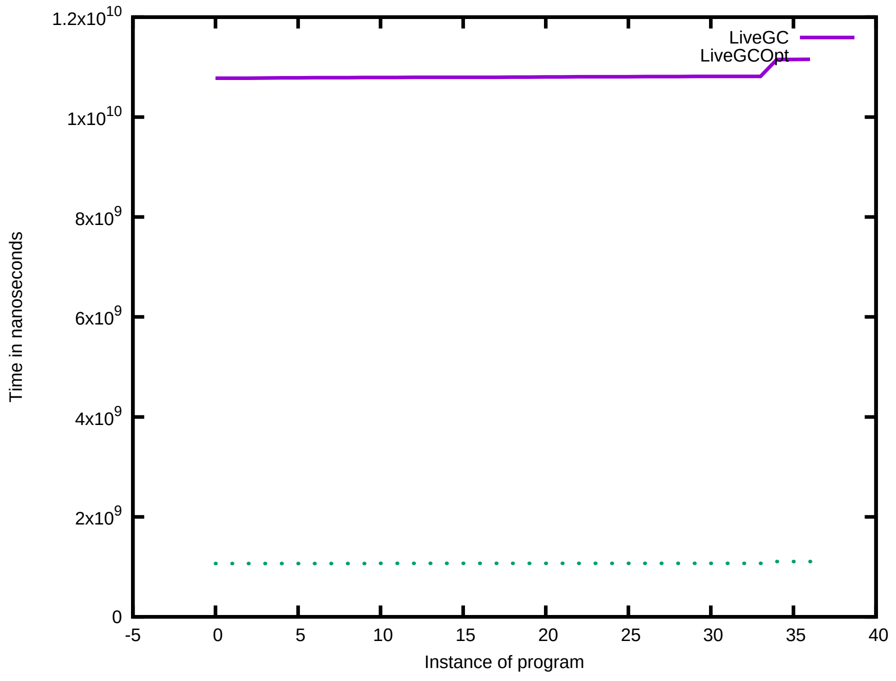


Figure 4.5: TreeAdd Program

4.2 Analysis

We will discuss the results for three examples - BST, Loop and CircularLoop and analyze the results obtained for the above programs.

4.2.1 BST

As we can see in 4.4 our modified algorithm has improved running time performance of LiveGC. 4.6 shows one of the access graph used. Revisiting objects using the same state in the access graph is redundant computation which is avoided by the LiveGCOpt . This improves running time of the algorithm. TreeAdd and BiSort also benefit from a similar situation.

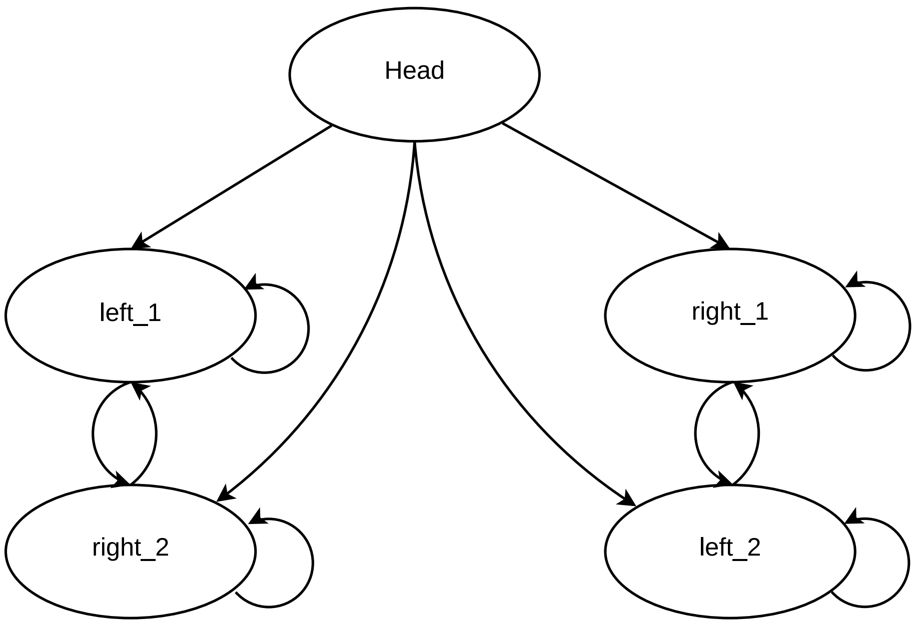


Figure 4.6: Access graph for BST

4.2.2 Loop

The results shown in 4.2 show that the running time was slightly more in case of the modified algorithm as compared to the simple algorithm. Analyzing the access graphs and the program shows that the structure of the heap is simple and program flow in the heap is linear. With little or no circular traversal of the heap the simple LiveGC executes similarly to LiveGCOpt. The improvement for

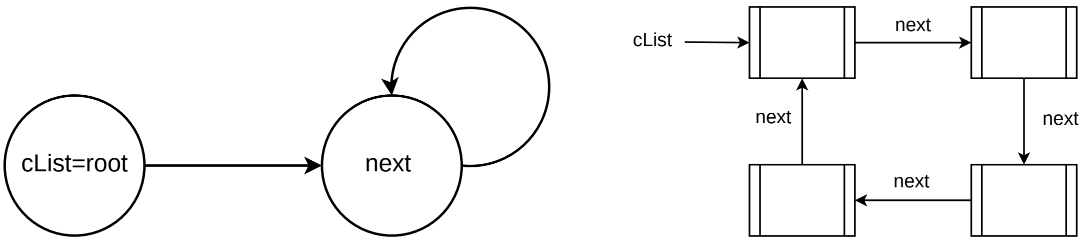
LiveGCOpt comes from the fact that heap objects that are visited again and again during the tracing which is not applicable in this case. But LiveGCOpt takes more time because of the added overhead of updating and looking up in the map for visited objects. Similar explanation applies for DLoop and Reverse program.

4.2.3 CircularLoop

Circular Loop contains a circular linked list which is traversed around using a loop. The simple LiveGC fails for this program because the circular structure of heap and the cycle in the access graph causes it to go into an infinite loop. Observe the Program 3 given below and it's corresponding access graph 4.7a as output by the explicit live reference analysis.

Program 3: CircularLoop.java

```
<some-code>
CircularLoop clist = new CircularLoop(n);
// creates a circular linked list of length 'n'
for(i=0; i<n; i++) {
  clist = clist.next;
}
// some code which uses 'clist'
doSomething(clist);
```



(a) Access Graph (b) Heap snapshot assuming n=4

Figure 4.7: CircularLoop.java

The access graph 4.7a clearly outlines that all field references named 'next' of the object '$cList$' are to be traversed recursively. But since in reality the structure in heap is circular, as illustrated in 4.7b this would lead to an infinite loop without marking objects that have been visited by the algorithm. Which is why LiveGCOpt works for this program whereas LiveGC gets trapped into an infinite loop.

4.3 Summary

For calculating space performance we use the same approach as used in the accompanying thesis [11]. In this approach number of objects calculated that are marked by the RGC and LiveGC are separately counted and divided by the number of GC calls. This gives an average measure for garbage collection. The results tabulated in 4.1 show that LiveGC and LiveGCOpt both perform better than RGC for all benchmarks except one for which performance is as good as RGC. There is no extra garbage collected in case of LiveGCOpt as compared to LiveGC which is expected behavior because the algorithm discussed in this thesis works only on improving running time.

SrNo	**Program**	**Avg number of objects on heap**			**Percentage of extra garbage**
		RGC	**LiveGC**	**LiveGCOpt**	
1	BiSort	11.47	6.36	6.36	14.32
2	Loop	32500	25000	25000	18.75
3	DLoop	37999	25000	25000	24.99
4	Reverse	38889	27778	27778	22.22
5	TreeAdd	3.87	3.7	3.7	1.02
6	BST	33534	33534	33534	0
7	GCBench	1173	1	1	117300

Table 4.1: Average performance of LiveGC over complete program

We used Java system utility functions to calculate precisely the running time of the three tracing algorithms used namely - RGC, LiveGC and LiveGCOpt. The results are tabulated in 4.2. As discussed in previous section for programs which do not involve circular structure in heap and access graph like Loop and Reverse the LiveGCOpt performs slower because of the overhead incurred due to using a map. For BiSort and BST the performance increase is drastic as the redundancy in tracing is reduced. RGC is faster than both LiveGC and LiveGCOpt at the expense of collecting less garbage. GCBench benchmark for garbage collector allocates large data structures on heap which are never deallocated and not used over the full range of the program. This kind of situation is tailor made for liveness based garbage collection and both liveness based collector perform sharply better than RGC in this case.

SrNo	**Program**	**Running Time in ms**			**Percentage of time improvement over LiveGC**
		RGC	**LiveGC**	**LiveGCOpt**	
1	BiSort	0.2	31	14	54.83
2	Loop	48	1653	1762	-6.59
3	DLoop	77	1666	1763	-5.82
4	Reverse	88	1837	1966	-7.02
5	TreeAdd	9	1115	1104	0.98
6	BST	94	7832	4237	45.90
7	GCBench	206	22	21	4.76

Table 4.2: Comparison of running time performance between LiveGC, LiveGCOpt and RGC

Another heuristic was implemented to improve performance. In this heuristic LiveGCOpt trace traversed according to the given algorithm and reverted back to RGC tracing if the number of times a particular node was visited is greater than a fixed constant k. We ran our tests on the same set of benchmarks for a variety of values for k. The results are tabulated below in Table 4.3.

We can observe from the table below that as the value of k increases the running time performance gradually becomes more closer to LiveGCOpt. But the improving time performance is not without its trade-off. For $k = 1$ the reduction in heap space from garbage collection is identical to that of RGC showing no improvement. Whereas for $k = 3$ heap size reduction is identical to LiveGCOpt but so is the time performance. For $k = 2$ performance is identical to LiveGCOpt except for the performance of BiSort and TreeAdd.

Program	**Running Time in ms**				
	RGC	**LiveGCOpt** $(k = 1)$	**LiveGCOpt** $(k = 2)$	**LiveGCOpt** $(k = 3)$	**LiveGCOpt**
BiSort	0.2	4	12	14	14
Loop	48	49	1800	1823	1762
DLoop	77	75	1795	1792	1763
Reverse	88	81	1997	1997	1966
TreeAdd	9	11	13	1126	1104
BST	94	90	4591	4378	4237
GCBench	206	221	22	22	21

Table 4.3: Comparison of running time performance between LiveGC with heuristic, LiveGCOpt and RGC

Apart from the programs tabulated above we also tested CircularLoop program which doesn't execute on LiveGC because of the limitations discussed in previous section.

Chapter 5

Conclusion

One disadvantage of garbage collectors is the performance hit programs have to incur due to a parallel GC thread that can also stop execution for a while as in the case for Stop-the-World type of garbage collectors. For a liveness based garbage collector to be viable alternative to reachability based collector we have to improve the performance of the gc both in terms of memory reclaimed and execution time. The algorithm we proposed has improved the running time of the tracing algorithm in most which is the most time consuming phase of the collection process. Also the results show that the garbage collected is around 20% more as compared to a reachability based garbage collector. So we have improved running time without compromising on the memory efficiency of LiveGC. The LiveGCOpt outperforms LiveGC by almost 50% in some cases. In cases where is performs slower the difference is less than 8%. Also LiveGCOpt is a more complete tracing algorithm as it also handles cases such as CircularLoop which is not handled by LiveGC.

There is scope for some more improvement which would improve the performance of the algorithm. The number of vertices in the access graph directly correlates to the running time of the algorithm. We can reduce the number of nodes in the access graph by converting the automaton to a more conservative automaton which would give a better performance albeit at the expense of gathering less garbage. Each object in the VM has an object header [1] which contains bits of information which are used for storing type information, hashcode, locks, and GC bits. Using GC bits in the object header instead of a separate map for storing the visited labels could lead to a better performance.

Bibliography

[1] JikesRVM: Object Model. `http://jikesrvm.org/Object+Model`.

[2] SOOT: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[3] Ole Agesen, David Detlefs, and J Eliot Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. In *ACM SIGPLAN Notices*, volume 33, pages 269–279. ACM, 1998.

[4] Rahul Asati, Amitabha Sanyal, Amey Karkare, and Alan Mycroft. Liveness-based garbage collection. In *Compiler Construction*, pages 85–106. Springer, 2014.

[5] Hans Boehm, Alan Demers, and Mark Weiser. A garbage collector for c and c++, 2002.

[6] David Gadbois, C Fiterman, D Chase, M Shapiro, K Nilsen, P Haahr, N Barnes, and PP PIRI-NEN. The gc faq, 2007.

[7] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):593–624, 2002.

[8] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[9] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 20(4):104–115, October 1995.

[10] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[11] Vilay Kandi. Liveness based garbage collection in java. Master's thesis, IIT Kanpur, 2014.

[12] Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):1, 2007.

[13] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11):701–726, 1998.

[14] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.