Static Analysis for Identifying and Allocating Clusters of Immortal Objects

Archana Ravindar Y.N. Srikant

IISc-CSA-TR-2005-15
http://archive.csa.iisc.ernet.in/TR/2005/15/

Computer Science and Automation
Indian Institute of Science, India

October 2005

Static Analysis for Identifying and Allocating Clusters of Immortal Objects

Archana Ravindar Y.N. Srikant

Abstract

Long living objects lengthen the trace time which is a critical phase of the garbage collection process. However, it is possible to recognize *object clusters* i.e. groups of long living objects having approximately the same lifetime and treat them separately to reduce the load on the garbage collector and hence improve overall performance. Segregating objects this way leaves the heap for objects with shorter lifetimes and now a typical collection can find more garbage than before.

In this article, we describe a compile time analysis strategy to identify object clusters in programs. The result of the compile time analysis is the set of allocation sites that contribute towards allocating objects belonging to such clusters. All such allocation sites are replaced by a new allocation method that allocates objects into the cluster area rather than the heap. This study was carried out for a concurrent collector which we developed for *Rotor*, Microsoft's Shared Source Implementation of .NET. We analyze the performance of the program with combinations of the cluster and stack allocation optimizations. Our results show that the clustering optimization reduces the number of collections by 66.5% on average, even eliminating the need for collection in some programs. As a result, the total pause time reduces by 62.8% on average. Using both stack allocation and the cluster optimizations brings down the number of collections by 91.5% thereby improving the total pause time by 79.33%.

1 INTRODUCTION

Garbage collection has come a long way since the time it was introduced for collecting lists in LISP. Now it has become a necessity in modern object oriented languages, since it successfully abstracts the problem of memory management from the user. Advances like collecting generations and concurrent collection were successful in bringing down the collection overhead and thereby making garbage collection practically usable in runtime systems.

All said and done, the program incurs performance penalty if it is garbage collected. So it becomes essential to keep the overhead at a minimum. This is possible if we reclaim the maximum amount of garbage with the least number of effective collections. Several previous work have tried to achieve this goal in their own way by looking at different object properties like connectivity [Hay91, Hir02, Hir03, Sam04], object types [Shu02], age [McK99] other than object traceability alone.

Our goal is to make each collection effective and thereby reduce the total number of collections required to reclaim garbage in the program. We achieve this by identifying long living clusters of objects and allocating them in a separate mature object space that is not subject to garbage collection. The idea is to avoid tracing objects that are going to live till the end. Segregating objects this way leaves the heap for objects with shorter lifetimes and now a typical collection can find more garbage than before, making collections more effective. Although we have studied clustering in a generational setting this elementary concept is applicable to incremental collectors too.

This paper describes a compile time clustering analysis algorithm based on the compositional pointer and escape analysis framework proposed in [Wha99]. The clustering algorithm makes use of the lifetime information of objects computed by the points-to escape analysis algorithm, that is constructed for every method. The objects that do not escape the longest living methods are designated as the root of the cluster. The objects that are reachable from the root are treated as cluster objects. All such cluster objects are statically allocated in a separate mature object space. When the stack frame of the method binding the lifetime of the root object is popped, the entire cluster is garbage and hence the mature object space can be reclaimed in its entirety.

The clustering scheme is evaluated using a baseline collector that can run in both stop-the-world and concurrent modes that we developed for *Rotor*, Microsoft's shared source implementation of .NET. The baseline collector has two generations and uses the copying scheme to collect both. We analyze the performance of the collector and the program with the cluster and the stack allocation optimizations. Our results show a marked decrease in the total number of collections and considerable improvement in the individual collection performance. It is observed that a combination of the clustering and the stack allocation optimization improves the performance even further.

The remainder of the paper is organized as follows. We begin by reviewing related work in Section 2. Section 3 describes the concept of clustering and how we extend the compositional pointer and escape analysis to identify clusters. In Section 4 we describe the baseline collector and the experimental platform. In Section 5 we present and evaluate the results. Finally we conclude in section 6 with possible avenues of future work.

2 RELATED WORK

Hayes introduced the term object clustering [Hay91]. The main observation was that large clusters of objects, pointed to by key objects were allocated at roughly the same time and lived for approximately the same amount of time. When the key objects became unreachable it indicated a good opportunity to collect. Hayes identified the cluster as the program executed and incrementally placed it in the mature object space. Our work tries to identify the cluster at compile time and statically allocates the cluster into the mature object space. The compile time clustering algorithm is used to find key objects. Unlike Hayes's scheme where the mature object space is collected, we do not subject the mature object space to garbage collection. We combine the concepts of escape analysis and clustering to reclaim the cluster.

Pretenuring tries to solve the problem of repeated collections of long living objects by directly allocating such objects into the old generation by using static and dynamic profiles [Bla98, Che98, Har00]. But the old generation is still subject to collection, so in spite of applying the pretenuring optimization, major collections might still occur. Our scheme tries to completely eliminate major collections by allocating these long living or immortal objects in a separate mature object space that is not subject to collection.

Dynamic object colocation [Sam04] allocates objects directly into the same area of an object that will reference it, by using a mix of compile-time and runtime optimizations. Static compiler analysis is used to compute connectivity information and the runtime component involves an allocation routine which takes a colocator object as an additional parameter and is responsible for dynamic colocation. The dynamic colocator can start placing objects into the mature object space only when some initial set of colocators are present. Hence it requires a warm up young generation collection to produce these initial colocators, whereas the intention of our scheme is to reduce the number of collections, even eliminating the need for collection if possible. [Sam04] reports a considerable increase in the number of intergenerational pointers for some of its programs. Our results indicate that clustering only reduces the number of intergenerational pointers but never increases it.

Connectivity based garbage collection makes use of the observation that connected objects die together. Based on this hypothesis it allocates objects that are connected together into a statically determined partition so that collecting a partition would be much faster than collecting the heap. [Hir03] works by building a hierarchy of partition dags and collects these partitions such that an ancestor is collected together with its descendants thereby eliminating the need for a write-barrier.

3 CLUSTERING

The concept of data is fundamental to every program. Programs feed on data, they build several data structures that assist them in performing their functionality. In the object oriented paradigm, objects store data. These data objects are seldom isolated, rather they are related to one another in some way and hence linked together to form *clusters*.

Most often a program is associated with a set of critical objects that are bound to stay till almost the end of the program. Such objects are said to be *immortal*. If these objects are treated in the same way as the default heap objects, they would unnecessarily be processed by the garbage collector, resulting in increased collection times. Figure 1 illustrates the impact of long living objects on the total collection time, measured as the fraction of time spent in scavenging live objects. We observe that the scavenge time accounts for a significant fraction of the total collection time (up to 83% in *_211_anagram*). Further investigation reveals that up to 88% of the objects were found to be live during the collection. Hence tracing immortal or long living clusters plays a major role in lengthening the total collection time.

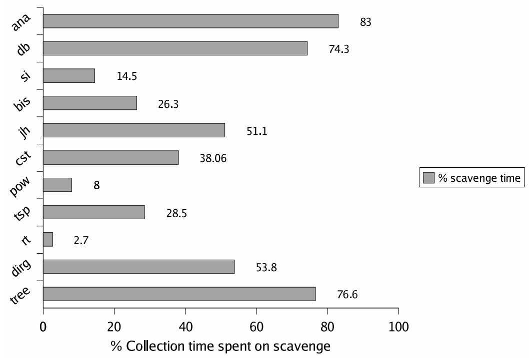

Figure 1: Proportion of Collection Time spent on Scavenge.

If we can recognize the allocation sites in the program responsible for creating long living clusters (highlighted in Figure 2) at compile time, we can statically allocate them in a region that is not processed by the garbage collector. The region can then be reclaimed in its entirety at program termination. Such a strategy allows the garbage collector to focus on objects that are volatile and objects whose lifetimes cannot be statically determined. We describe the clustering algorithm which identifies long living clusters in the next section.

```
Class anagram1 {
 ...
 public void read_file(String filename) {
 ...
 dict = new Hashtable();
 tab = new StringBuilder();
 for( i=0;  i<16;  i++ )
  tsb[i] = new StringBuilder(256);
 sb = new byte[256];
 m = new bool[256];
 FileStream sif = new FileStream(..);
 ..
 buffer = new byte[n];
 if( e < n ) {
  String  istr = new String(buffer, s, e-s);
  dict.Add(istr, istr);
 }
};
public Class anagram {
 ..
 public long run(String[] arg) {
  anagram1 agm = new anagram1();
  ..
 }
};
};
```

Figure 2: Set of Allocation Sites that contribute to Cluster Objects in _211_anagram

3.1 Extending Compositional Pointer Analysis To Identify Clusters

The algorithm to identify clusters in a program is based on the compositional and pointer escape analysis proposed for Java programs by Whaley and Rinard [Wha99]. The referencing behavior among objects and fields is abstracted in the form of a points-to escape or the PTE graph. Nodes in the PTE graph represent objects allocated by the program and edges represent references between them. Objects that are created within the currently analyzed region are represented by *inside* nodes in the PTE graph, whereas those created outside the currently analyzed region or accessed via outside edges are represented by *outside* nodes in the PTE graph. Similarly *inside* edges represent references created within the currently analyzed region. References created outside the currently analyzed region are represented by *outside* edges in the PTE graph. We restrict our analysis to programs that are single-threaded.

The algorithm is compositional in nature i.e. methods can be analyzed independently of their callers and callees. [Wha99] describes an intra-procedural algorithm that computes individual PTE graphs for each method and an inter-procedural algorithm that computes precise points-to escape information for each method. The inter-procedural algorithm combines the PTE graph for each method with the PTE graphs created for all its callees.

The ultimate objective of the algorithm is to determine for every allocation site A, the method M whose stack frame will outlive the object created at A. In such a situation, object created at A is said to be captured by M. If enough information is not available to ascertain whether an object escapes or not, it is allocated in the heap.

An object is said to have escaped a method M if it is a formal parameter or if a reference to the object is written into a static class variable or a reference to the object is passed to one of the callees of M say N and there is no information available about what N did to the object. The object will escape if M returns it. If the object satisfies none of the above conditions it is said to be captured within M.

In essence, when a complete points-to escape analysis graph is constructed for a method M it consists of the nodes that were either created within the method M or nodes created outside M but are reachable from within M. The clustering algorithm makes use of this fact to recognize a cluster.

3.1.1 Design

In this section we describe the clustering algorithm in the form of pseudocode as shown in Figures 3 and 4. To begin with, we need to preprocess the statements to include only those that will affect the PTE graph [Wha99]. The csharp compiler invokes CompileMethod for every method, that creates basic blocks, while it translates the source code into opcodes. We intercept at points where code is generated for statements that we are interested in and save the details of the statement in a separate data structure.

Once the code for the method is generated, we iterate through the statements that we created to compute the PTE graph. The graph is implemented as an adjacency list. Each node is a structure that stores the set of incoming and outgoing edges, node kind and information whether it was visited or not. Each edge is a structure that stores the head and the tail node, edge kind and the variable it represents.

During the intra-procedural analysis, when we encounter a call statement it is possible that the PTE graph for that call is not yet computed. The status of all such statements that have incomplete information is marked as *pending*. During the inter-procedural analysis we process only pending statements to compute the complete PTE graph. Finally, we process the PTE graphs of only those methods M that lie close to main in the call graph, to compute cluster information. This list of methods can be got by profiling. The PTE graph for all such M would consist of only those nodes that have escaped up to M, since they are reachable from within M. All other nodes that have been captured within methods lying below M would not be visible in the PTE graph for M. Hence the cluster algorithm correctly identifies only those objects that are going to live till the stack frame of M has been popped off and is bound to benefit the collection process.

The marked nodes in M which are not pointed to by any other node in the PTE graph of M are said to be the roots of the cluster. They serve the same function as the key objects because they are the only way to reach a cluster. When the key object is garbage, all the objects connected to it are dead. Hence when the stack frame for method M is popped, the root object and hence the entire cluster associated with it is dead and can therefore be reclaimed.

The clustering analysis algorithm is conservative in the sense that some of the objects belonging to the cluster might die before the stack frame containing the root of the cluster is popped. This is especially true in cases where a dynamically growing structure like a stack or a list is part of the cluster. However, we shall shortly see that even this naive approach of identifying a clusters performs reasonably well for most programs.

```
Procedure compileNamespace(var nsdecln: namespace)
...
for each class of nsdecln, cls do
 compileClass(cls);
end for
...
for each class, C of nsdecln do
 for each method M of C do
/* interprocedural analysis */
```

```
  interProcedural(M)
/*consider only long living methods */
  if M belongs to the longest living method set then
   Append(methodList, M);
 end for
end for

/* compute cluster information */
for each method M in methodList do
 outputCluster(M);
end for
end compileNamespace

Procedure compileClass(var cls: class)
/* generate code for each method of cls */
end compileClass

Procedure compileMethod(var meth: method)
/* initialize basic blocks, set up locals and parameters */
/* and generate code    */
...
/* intraprocedural analysis */
createPTEGraph();
end compileMethod

Procedure createPTEGraph()
/* generate nodes and edges for load, store, return, assignment */
/* and object creation statements  */
/* enter statements into a global statement list  */
end

Procedure interProcedural
 for each stmt S in the global statement list do
  if S.status = pending then begin
   if S.kind = 'New' or S.kind = 'Call' then
    patch(S, S.callm);
  end
 end for
end interProcedural
```

```
Procedure outputCluster(var meth: Method)
 var g: Graph;
 ...
 g := LocalPTEGraph(meth);
 for each node in g do
  if (node.kind = 'inside' and node.visited = 'false') then begin
  /* node is the root of the cluster */
  DFS(node);
  end
 end for
end outputCluster

Procedure DFS(var nd: node)
 var st:Statement;
 if nd.visited = 'true' then return;
 for each edge E in nd.outgoing do
  st := E.var.parent;
 /* output allocation site number */
  if st.kind = 'New'
   print(st.stmNo);
  DFS(E.tail);
 end for
 for each edge E in nd.incoming do
  st := E.var.parent;
 /* output allocation site number */
  if(st.kind = 'New)
   print(st.stmNo);
  DFS(E.head);
 end for
end DFS
```

Example Figure 3 shows the local PTE graphs for two of the methods in *_211_anagram*. In the PTE graph for *read_file*, *sif* is captured. Despite being a local object, *istr* is linked to the dict variable by the library call *dict.Add* and hence becomes a part of the cluster. Since the reference is added outside the method, it is indicated as an outside edge. The intra-procedural analysis for *read_file* deems all nodes except *sif* as escaped. The dotted line in Figure 3 indicates how the nodes in run will be mapped onto the nodes of the callee *read_file* during inter-procedural analysis.

Interprocedural analysis is followed by the application of the clustering

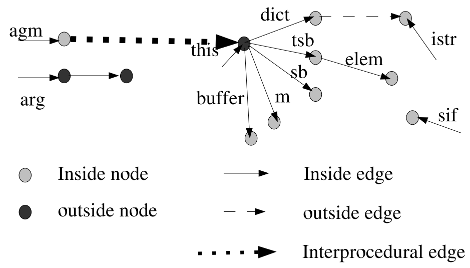

Figure 3: Identifying clusters using PTE graphs.

algorithm as described earlier, that marks all the nodes in the graph that corresponds to the cluster allocation sites. In this particular example, the clustering algorithm accesses the complete PTE graph of *run* and marks all nodes reachable from the node representing *agm* as cluster nodes. *agm* is designated as the root of the cluster. Since by definition each node is associated with an object and hence with an allocation site producing that object, one can output the set of allocation sites responsible for cluster allocation.

The fact that the analysis is compositional makes it possible to analyze libraries independently of the application. When analyzing an application, we use precomputed results for any library calls that it may make. Since the clustering algorithm can access the precomputed results for the library calls, it is possible for the algorithm to come up with cluster allocation sites within the library code, as we saw *dict.Add* in Figure 3. To support clustering completely, we create a new library that consists of additional functions to support cluster allocation.

Other changes to Rotor for implementing the clustering scheme include the introduction of two new opcodes *newclus* and *newst* that are wired to perform allocation in the cluster and in the stack respectively. In this implementation, we have simulated the allocation on the stack using a separate area apart from the heap and the cluster area. To measure the impact of simulating the stack allocation we ran the programs with a maximum heap size (so that there was no garbage collection) and compared the elapsed times with the baseline which has no stack allocation implemented. On average, the overhead of stack implementation was found to be -2.1%.

3.1.2 Issue with Boxing

In any implementation of CLI, when an instance of a value type is passed as a parameter to a method that expects a reference parameter, boxing is performed [Ecm03]. Boxed objects are implicit and are not evident in csharp source code. Since the clustering algorithm works on the source code, it does not have a handle to the boxed objects. Our implementation tackles this problem by converting implicit boxing to explicit boxing. We overload the existing methods that take a reference as a parameter, to take value types also. These additional methods now include code that performs explicit boxing. So now the clustering algorithm can access the boxed objects and include them in the analysis.

4 METHODOLOGY

Baseline Collector

The baseline collector is designed to work on the principles of concurrent replication collection [Too93]. It consists of two generations. The young generation is also known as *newspace*. This is where all the new objects are allocated. The old generation is comprised of two semispaces- *fromspace* and the *tospace*. Copying collection is used to collect both generations. When allocation in the newspace crosses a particular threshold, a *minor* collection is invoked that scavenges the live objects into *fromspace*. Eventually the *fromspace* gets filled up to its threshold value which invokes a *major* collection that collects the entire heap.

4.0.3 Implementing Concurrency

Copying is a concurrent operation, hence the program and the collector thread need to be synchronized to ensure that things work correctly. Our approach for synchronizing the program and the collector is an extension of the Dijkstra's tricolor scheme. We associate each object with a color that is used to indicate object state information to both the collector and the program.The color associated with every object serves to communicate its state information to the program and the collector.

1. *White*: Initially all objects are colored white. This indicates that the collector has not yet considered the object.

2. *Gray*: When the collector starts copying an object into the destination space, it assigns the object, the color gray. A gray object indicates to the program that the collector is right now copying that object.

3. *Black*: When the collector has finished replicating an object into the destination space, it assigns the object the color black. A black object

indicates to the program that the collector has already copied that object.

4. *Red*: When the program mutates an already copied object, it is assigned the color red. This indicates to the collector that the live object graph has been changed and the same changes have to be reflected on to the replica graph. At the end of scavenging the collector processes the red objects and re-scavenges them back to the destination area so that the live object graph and the replica are the same.

To illustrate how these colors help in synchronization, let us assume that the collector has finished copying object A and is now copying object B as shown in Figure 4. The collector then goes ahead and copies the entire structure into the destination area as shown in Figure 5. Now the object A is written by the program such that it now points to object C instead of B as shown in Figure 6. Since A and its children were already copied to the destination area, it is colored red. All such red objects are eventually processed by the collector at the end of scavenge phase, making the final heap look like Figure 7. So this is how the proposed coloring scheme substitutes the mutation log in ensuring synchronization. Two bits are needed in the object header to store the color information apart from a single bit used to store the generation and a word to store the new address.

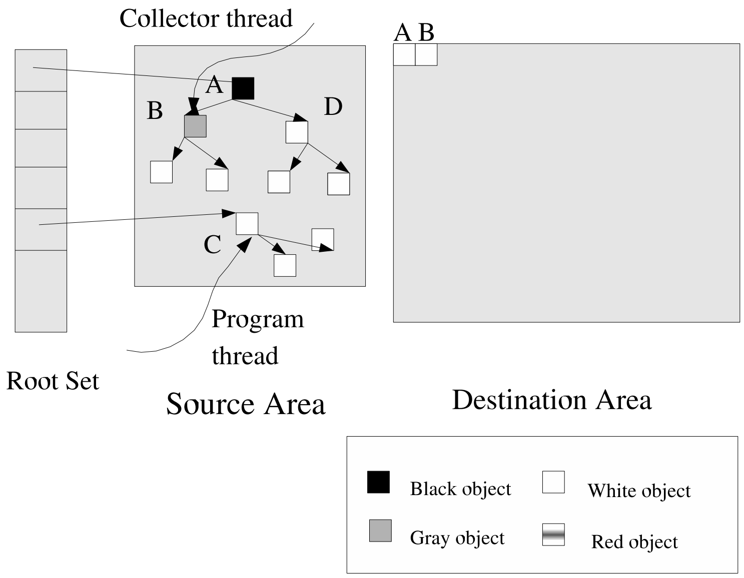

Figure 4: In the midst of tracing object structure 'A'

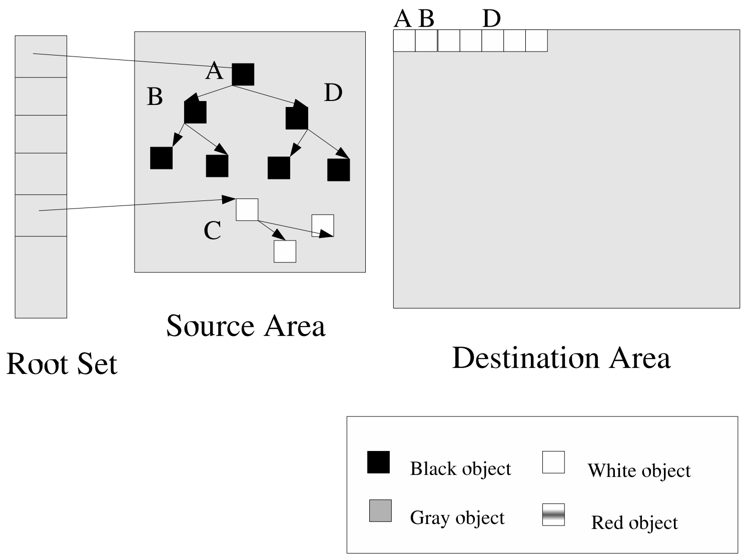

Figure 5: After tracing object structure 'A'

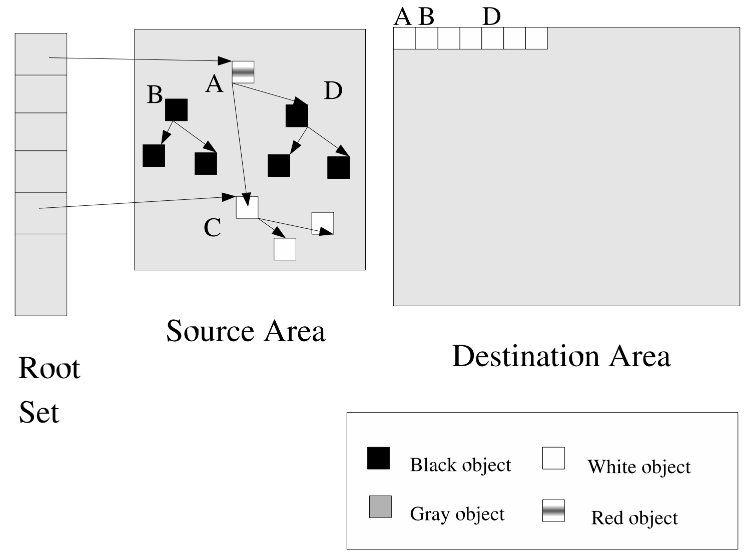

Figure 6: coloring 'A' red

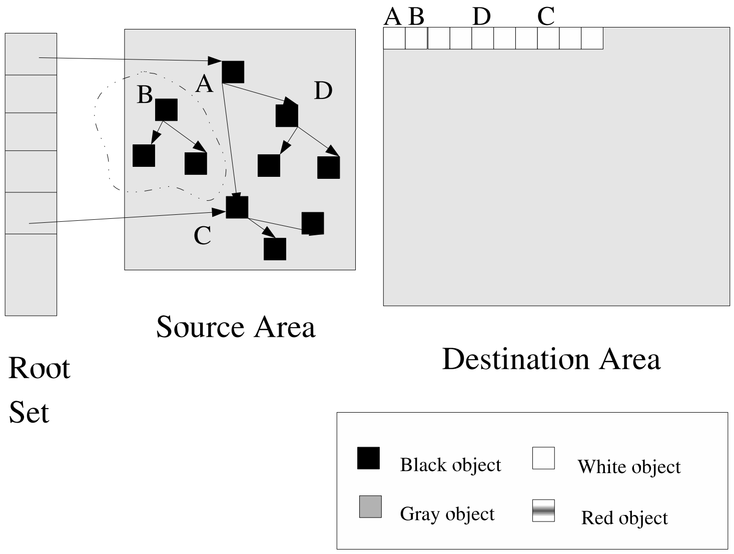

Figure 7: Handling red objects

4.0.4 Write Barrier

The baseline collector is a generational collector and hence needs a write barrier [1] to keep track of intergenerational pointers. We take the card tables approach [Hos92] for this purpose. In addition to using the write barrier to track intergenerational pointers, we make use of it to support concurrency in the collector. So now the write barrier in the baseline collector monitors object writes and ensures synchronization by performing color checks apart from recording possible intergenerational pointers.

The details of how the barrier functions, is explained as follows.

1. Synchronization:

   - A write to a white object is ignored; no synchronization is required.
   - A write to a gray object waits until the object becomes black; the program waits for the collector to finish copying the object completely before it writes onto it.
   - A write to a black object colors it red, for further processing by the collector.

[1] A piece of code executed with every pointer write

- A write to a red object is ignored; Anyway the object is going to be processed by the collector.

2. Intergenerational pointers:

   - A write to a young object is ignored.
   - A write to an old object causes the corresponding card table entry to be set [2]

4.0.5 Object Model

The collector has to adapt to the existing object model since it is plugged into *Rotor* and the object model is widely used in all the other layers of *Rotor*. The object header in *Rotor* stores the method table address and a slot that stores the relocated address of an object. This slot at the same time stores an integer known as the syncblock index, used for object synchronization. The garbage collector in *Rotor* is of stop-the-world kind. Hence it can afford to use the same word for two purposes. During collection, the slot is used to store the new address of the copied object. Once the collection process is over and all the objects are updated to the new addresses, it resumes the function of storing the syncblock index.

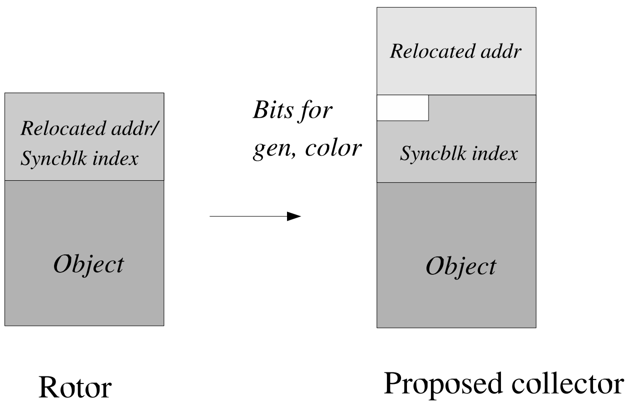

Figure 8: The Object Model in the proposed collector.

Since the collector is concurrent, both the relocated address and the syncblock index might be required for use at the same time. Hence an additional header word is needed to store the relocated address. The size of the

[2] one can perform an additional comparison to check whether the pointer being written into the old object is a young pointer. But that increases the cost of the write barrier.

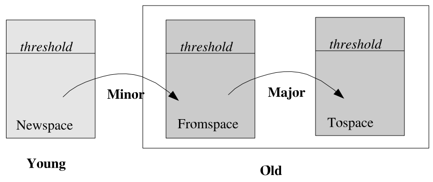

Figure 9: Baseline Collector Organization with Clustering Incorporated.

object header for the proposed collector increases by one word (4 bytes). To store the generation and color, the free bits located in the slot for syncblock index are used. The object model in *Rotor* and the proposed collector is shown in Figure 8.

The baseline collector supports finalization, weak pointers and interior pointers. However, unlike the rotor garbage collector, it does not support large objects allocation and pinning. Incorporating clustering into the garbage collector adds a new mature object space to the existing heap. The baseline collector can also run in the stop-the-world mode. The final memory model of the collector is as shown in Figure 9.

Experimental Platform

This study was conducted on Rotor version 1.0 [Rot01]. We ran the programs on an Intel pentium III 450 Mhz processor with 128MB of main memory and a 512KB cache, running Free BSD 4.5.

5 RESULTS

In this section, we evaluate the baseline collector by comparing its performance with Rotor's garbage collector. We evaluate the clustering optimization w.r.t. the collector and program performance. We also study the impact of the stack allocation optimization along with the clustering optimization. To carry out this study, we used the C# versions of the Java programs from Spec JVM98 [Spc98], Java olden [Jolden], Java grande [Jgrande] and the gc test suite provided with Rotor [Rot01]. The benchmarks and their runtime parameters are summarized in Table 1.

5.1 Performance of the Concurrent Collector

In this section we describe the performance of the baseline collector w.r.t. pause times and elapsed times. The results for both the stop the world and

Source	Program	Runtime parameters
Rotor gc test suite	directedgraph	No. of vertices=100
Spec JVM98	_208_cst	No of iterations=1, speed=1
	_209_db	No. of iterations=1, Speed=10
	_211_anagram	Speed=1
	_210_si	Speed=10
Java Olden	bisort	No of nodes=4, size=2500
	jhealth	MaxLevel=5, MaxTime=100, seed=23
	power	No of feeders= 5, No of laterals= 10, No of branches= 3, No of leaves= 5
	tsp	Size= 600
	treeadd	No of levels=16
Java Grande	raytrace	Width= 25, height= 25

Table 1: Set of Benchmarks used and their Configuration

concurrent modes are presented. The heap sizes are chosen such that both the Rotor garbage collector and the baseline collector have the same number of collections.

5.1.1 Pause Time

The main objective for choosing a concurrent gc algorithm for the baseline collector was to reduce the pause times. Almost all the programs report significant reductions in pause times for the concurrent mode, except for *raytrace* which shows an increase of 4.62%. The average reduction in pause times for the concurrent mode is 36.24%. However pause times increase by 2.14% on average when the collector is run in the stop-the-world mode.

5.1.2 Elapsed Time

The baseline collector introduces a very small overhead of 1.11% when run in the stop-the-world mode. However, the overhead is slightly worse in the concurrent mode. That is because of the additional synchronization code that needs to be executed. The average overhead on the elapsed time is 1.75% for the concurrent mode. It can be observed that in spite of a substantial improvement in the pause time, the elapsed times do not change by much. That is because the collection time constitutes a very small portion of the elapsed time.

5.2 Performance of Clustering

In this section we describe the performance of the programs when the clustering optimization and the stack allocation optimizations are performed. The programs are run with the heap sizes as shown in Table 2. Clustering reduces the total heap requirement by 12.6% on average.

Program	Young gen size (MB)	Old gen size (MB)	Young gen with clustering (MB)	Old gen with clustering (MB)	Max Cluster size(MB)
_211_anagram	2	8	0.7	1.4	3.8
_209_db	1	10	1	2	2.5
_210_si	1	2	1	2	0.9
bisort	1	2	1	2	0.05
jhealth	1	2	0.3	0.6	2.6
_208_cst	1	40	0.7	1.4	12.7
power	1	2	0.3	0.6	0.07
tsp	1	2	0.7	1.4	0.05
raytrace	0.8	1.6	0.8	1.6	3.6
directedgraph	1	2	1	2	0.15
treeadd	4	8	0.19	0.38	1.4

Table 2: Heap and Mature Object Space sizes

5.2.1 Fraction of bytes allocated in the mature object space

Almost all programs allocate significantly into the cluster area as shown in Figure 10. However there are programs such as *power*, *tsp*, *directedgraph* in which the bytes allocated for temporary objects outnumber the bytes allocated for cluster objects by a huge margin. These programs benefit from the stack allocation optimization that is described in [Wha99]. The amount of temporary objects are similarly huge in programs such as *bisort* but they cannot be considered for stack allocation, since their sizes are not known at compile time. Three patterns of programs seem to emerge at this point.

- Programs purely suited for cluster allocation.
- Programs suited primarily for stack allocation
- Programs that are a mixture of both.

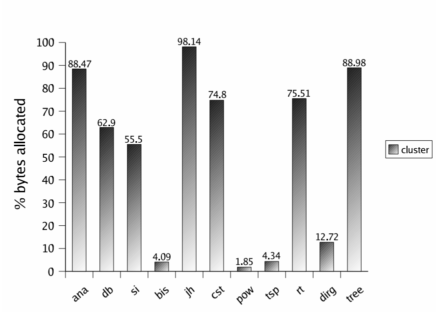

Figure 10: Fraction of bytes allocated in the cluster area.

5.2.2 Reduction in the Number of Collections

Both clustering and the stack allocation optimizations are geared towards reducing the load on the garbage collector. For certain programs where the total population of objects is dominated by clusters, clustering optimization yields a lot of benefit. For programs where volatile objects dominate, stack allocation yields similar benefit. The average reduction in the number of collections for programs where only the stack allocation optimization and the clustering optimization is used is 75% and 66.5% respectively. A combination of the stack and cluster allocation yields the highest reduction of collections at 91.56%. The results are the same for the collector when operated in the concurrent mode. The impact of the clustering and stack allocation optimizations on the number of collections is as shown in Figure 11

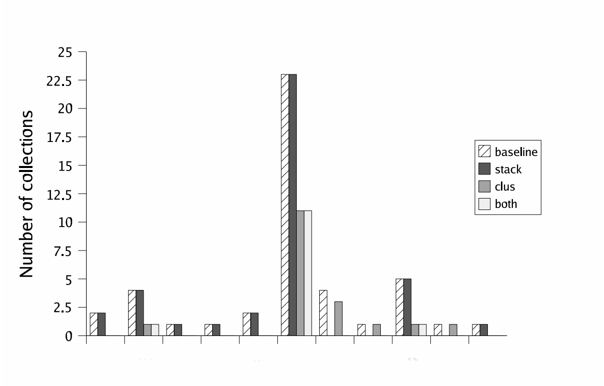

Figure 11: Impact of the Clustering and the Stack Allocation Optimization on the Number of Collections

5.2.3 Reduction in Collection and Pause Times

Figure 12 indicates the impact of the clustering and stack allocation optimizations on the collection time. One of the direct consequences of the reduction in the number of collections is the reduction in the total collection time and the total pause time. Reduction in the number of objects scavenged also contributes to reduction in the collection time. The average reduction in the total collection time using only the stack allocation optimization is 60.9%; with only the cluster optimization it is about 60.6%; with both optimizations on, the reduction is about 79.27%. The corresponding average reductions in the pause times are 63.55% with only the stack allocation optimization, 62.82% with only the cluster optimization and 79.33% with both optimizations applied.

When the collector operates in the concurrent mode, the average reductions in pause times are 60.09%, 60.9% and 79.27% with only the stack allocation, only the clustering optimization and both optimizations applied respectively.

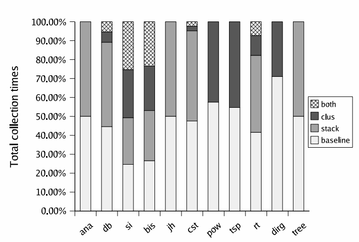

Figure 12: Impact of Clustering and Stack Allocation Optimizations on the Total Collection times

5.2.4 Reduction in Copycounts

Once the clustering optimization is done, there is greater chance for a collection to find more garbage than earlier. Since the long living clusters are exempted from collection, only those objects that are relatively volatile remain in the heap. This causes a reduction in the number of objects copied as indicated in Figure 13. Copy counts can also reduce due to the reduction in the number of collections as we saw in the previous section. Copy counts reduce by almost 60.11% with only the stack allocation optimization applied and by 91.37% with the cluster optimization applied. A combination of both reduces the copy counts further by 94.02%. The results are almost the same for the collector when operated in the concurrent mode.

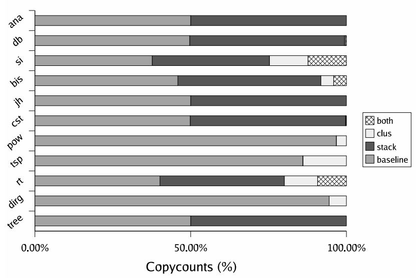

Figure 13: Total Copy Counts with Clustering and Stack Allocation Optimizations

5.2.5 Impact on Inter-region References

A profile of the inter-region references indicate very minimal interaction between the cluster objects and the heap objects (Table 3). The number of such cluster to heap pointers is critical to the success of clustering. The cluster is reclaimed in its entirety and not collected as in the case of the heap that is collected from time to time. Just as we track inter-generational pointers to ensure complete collection, we need to track cluster to heap pointers. Hence, if the number of such cluster to heap pointers are large, the collection time is bound to increase. The average number of cluster to heap references that the clustering algorithm achieves is 1.54. Clustering is also found to reduce the total number of inter-region pointers as shown in Table 3. The impact on the number of inter-region pointers is studied only for those programs in which the number of collections are reduced to a non-zero value with the application of clustering. The average reduction in the number of interregion pointers is found to be 33.72%.

5.2.6 Impact on Allocation Time

The cluster allocation routine is straightforward and need not populate objects with extra header information which would otherwise be required for heap objects. So the time required to allocate a cluster object is less than

Program	Total No. of c-h [b] references	Total i-r [a] pointers without clustering	Total i-r pointers with clustering	% Reduction in barriers	% Garbage within cluster
_211_anagram	5	-	-	-	25
_209_db	1	6916	14	99.79	11.2
_210_si	2	44731	39324	12.08	19.38
bisort	0	7	7	0	0
jhealth	0	-	-	-	77.5
_208_cst	6	403912	169319	58.08	33.3
power	0	1	1	0	0
tsp	0	8	8	0	0
raytrace	3	163798	293	99.82	99.5
directedgraph	0	-	-	-	0.02
treeadd	0	-	-	-	0

[a] inter-region
[b] cluster to heap

Table 3: Interregion References and Effectiveness of the Clustering Scheme

the time required to allocate a heap object. Clustering improves the total allocation time by 14.99% on average. Stack allocation improves the total time by 12.61% on average. A combination of both optimizations results in an improvement of 20.63%.

5.2.7 Impact on Elapsed Time

Clustering has little effect on the total elapsed time, on average it increases the elapsed time by 0.44%. Using only the stack allocation optimization improves the elapsed times by 1.75% on average. A combination of both the optimizations improves the elapsed time by 1.018%. The main reason for this is that the collection time is only a small portion of the overall elapsed time. Only if there is a drastic improvement in the collection time, elapsed times improve visibly, for example the number of collections in _208_cst with the clustering optimization decreases from 23 to 11. Hence, in this case the elapsed time reduces by 12.19%. The other reason is that the addition of a separate cluster area introduces overheads w.r.t. the elapsed time. Since an object can now reside in the cluster area apart from the heap, the garbage collector code needs to recognize objects in the cluster area and also in the stack, if the stack allocation optimization is applied.

Additional barrier code to keep track of cluster to heap pointers also contributes to increased elapsed times. The effect on elapsed time is more or less the same for the concurrent mode. Using just the escape analysis optimization, the average elapsed times decrease by only 0.377%; clustering increases the elapsed times by 1.19%. Using both optimizations the average elapsed time decreases by 0.59%. The impact of the clustering and the stack allocation optimizations on elapsed time is shown in Figure 14

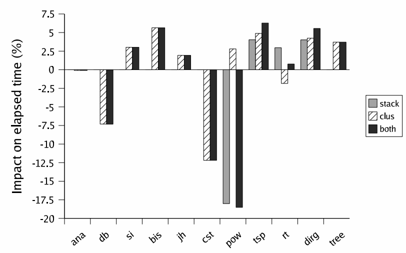

Figure 14: Impact of the Clustering and Stack Allocation Optimizations on the Elapsed Time

5.2.8 Effectiveness of the Clustering Algorithm

To evaluate the effectiveness of the clustering algorithm and to verify its claim of retaining genuinely long living objects right up to the end, additional instrumentation is added to the code. At the time of reclamation of the cluster area, instead of freeing it up, the cluster area is collected to find the amount of garbage generated within itself. The amount of garbage generated in the cluster using our algorithm is found to be 24.17% on average. Ideally it should be 0%.

The clustering algorithm presented here does not capture dynamic growth of clusters. More complex pointer analysis is required to come up with an ideal cluster. Since the clustering algorithm is by nature static, Allocation site homogeneity is an issue. For example *raytrace* includes an allocation site that is called in two different contexts. In one it creates a captured object, in the other it creates a cluster object. If we decide to allocate the object in the heap, the number of cluster to heap references shoot up, thereby degrading the performance of the collector. On the other hand, if we decide to cluster allocate the object, huge amount of garbage would be generated within the cluster area due to the volatile nature of the object. In such cases dynamic object colocation [Sam04] might perform better since it makes colocation decisions on the fly at runtime.

5.2.9 Impact on Static Compilation Time

The clustering algorithm extends the compositional pointer analysis framework proposed by [Wha99] and hence involves construction of PTE graphs for all methods and a series of intraprocedural and interprocedural analysis. The baseline to evaluate the time taken to perform static analysis is the plain *Rotor Csharp* compiler without any code for graph creation or cluster detection. The average impact of performing points-to escape analysis to detect clusters on the static compilation time is found to be 6.73%.

6 CONCLUSIONS

For a garbage collector to work effectively, it has to be aware of object properties and not just object traceability. The compiler plays an important role in providing valuable information about object properties to the garbage collector. This paper describes and evaluates a compile time technique that recognizes clusters in a program and statically allocates cluster objects separately. Our results demonstrate that the clustering optimization reduces the number of collections considerably and also improves the individual collection times by a fairly large amount. When applied along with the stack allocation optimization it produces even better results. Clustering also improves the total number of interregion pointers. However, elapsed times do not improve in the same vein as the collection times. Only those programs in which there is a drastic reduction in the number of collections show a considerable improvement in the elapsed time.

6.1 Future Work

Our work can be extended in several directions. The current clustering algorithm identifies clusters that are created only in those methods that have the longest lifetimes. One can extend the clustering concept to all other methods to discover scoped memory regions. The current compiler analysis itself can be made more sophisticated so that it not only outputs the allocation sites but also provides information to the programmer whether the cluster optimization would prove beneficial for that program or not. Several parameters are indicative of whether a cluster would prove as an advantage or as a penalty. Some of them are the number of cluster to heap references, allocation site homogeneity, the fraction of the objects that are allocated in the cluster, dynamic growth of clusters that might contribute garbage within the cluster area. However, the compiler would require complex pointer analysis to infer some of this information. Allocating cluster objects in a separate area brings in the need for additional barrier code to track cluster to heap references. Static analysis can be used to eliminate the write barriers wherever unnecessary and hence improve elapsed times.

7 ACKNOWLEDGEMENTS

This work was funded by Microsoft Research. We thank the anonymous reviewers for their comments and suggestions.

References

[App89] A.Appel, Simple generational garbage collection and fast allocation. Software: Practice and Experience, 19(2):171-183, Feb 1989.

[Bla98] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 342-352, Tampa, FL, Oct. 2001. ACM.

[Che98] P.Cheng, R. Harper and P. Lee, Generational stack collection and profile-driven pretenuring. In ACM Conference on Programming Languages Design and Implementation, pages 162-173, Montreal, Canada, May 1998.

[Det02] Morgan Deters and Ron K. Cytron, Automated Discovery of Scoped Memory Regions for Real-Time Java. In ACM International Symposium on Memory Management, pages 25-35, Berlin, Germany, June 2002.

[Ecm03] ECMA C# and Common Language Infrastructure Standards. http://msdn.microsoft.com/net/ecma/

[Gay01] D. Gay and A. Aiken. Language Support for Regions. In Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation, pages 70-80, 2001.

[Har00] T. L. Harris. Dynamic adaptive pre-tenuring. In ACM International Symposium on Memory Management, pages 127-136, Minneapolis, MN, Oct. 2000.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1991.

[Hir02] M.Hirzel, J.Hinkel, A.Diwan and M.Hind, Understanding the connectivity of heap objects. In ACM International Symposium on Memory Management, pages 36-49, Berlin, Germany, June 2002.

[Hir03] M.Hirzel, A.Diwan and M.Hertz. Connectivity based garbage collection. In ACM Conference on Object Oriented Programming Systems , Languages and Applications, pages 359-373,Anaheim, CA, Oct 2003.

[Hos92] Antony L. Hosking, J. Eliot B. Moss and Darko Stefanovic, A comparative performance evaluation of write barrier implementations. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 92-109, 1992

[Jgrande] http://www.epcc.ed.ac.uk/javagrande

[Jolden] Brenden Cahoon, Java Olden benchmarks, http://www.cs.utexas.edu/users/cahoon/

[Lie83] H. Lieberman and C. E. Hewitt. A real time garbage collector based on the lifetimes of objects. Communications of the ACM, 26(6):419-429, 1983.

[McK99] D. Stefanovic, K. McKinley, and J. Moss. Age-based garbage collection. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 370-381, Denver, CO, Nov. 1999.

[Jon96] Richard Jones and Rafael Lins, Garbage Collection: Algorithms for Automatic Memory Management, John Wiley and Sons, Inc., New York, 1996.

[Rot01] http://www.sscli.net

[Rug87] Cristina Ruggieri and Thomas P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects, pages 285-293, ACM SIGPLAN'88

[Sam04] Samuel Z. Guyer and Kathryn S. Mckinley. Finding Your Cronies: Static Analysis for Dynamic Colocation. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 237-250, Vancouver, British Columbia, Canada, October 2004.

[Shu02] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In ACM Symposium on the Principles of Programming Languages, pages 295-306, Portland, OR, Jan. 2002.

[Spc98] http://www.spec.org/osg/jvm98

[Stu03] David Stutz, Ted Neward and Geoff Shilling. Shared Source CLI Essentials.

[Too93] James O' Toole and Scott Nettles. Concurrent Replicating Garbage Collection. In ACM Symposium on LISP and Functional Programming

[Wha99] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs, In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 187-206, Denver, CO, Nov. 1999.