

**Compiler Design**  
**Prof. Y. N. Srikant**  
**Department of Computer Science and Automation**  
**Indian Institute of Science, Bangalore**

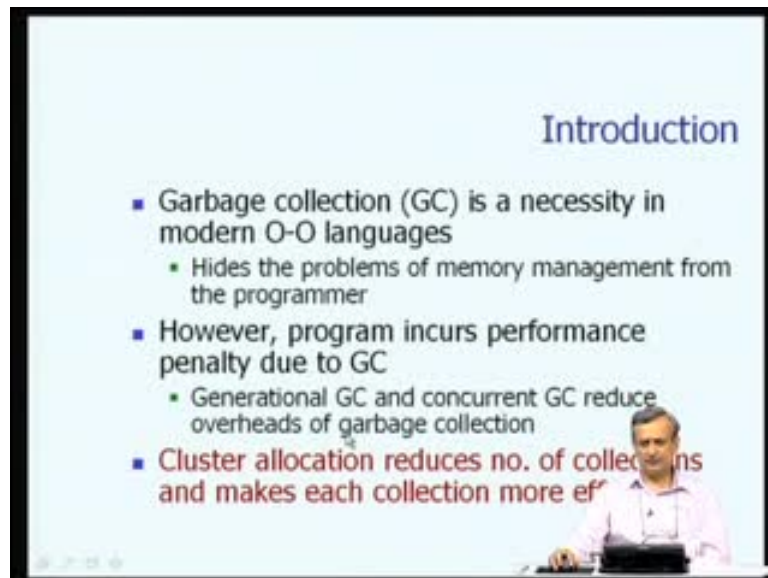
**Module No. # 19**

**Lecture No. # 37**

**Garbage collection**

Welcome to the lecture on garbage collection. This is going to be a lecture on the application of static analysis to garbage collection - a static analysis for identifying and allocating clusters of immortal objects.

(Refer Slide Time: 00:35)



Why do we really need to do garbage collection? This we have already discussed in one of the previous lectures on memory management. Garbage collection is a necessity in modern object oriented languages. It hides the problems of memory management from the programmer. That is very briefly put, what GC is all about. When we create dynamic data structures the nodes of the dynamic data structures such as tree or link list would be deleted as per the program requirements. And instead of placing the burden of returning the deleted nodes to the memory pool it is possible to arrange the garbage collector collect such deleted and unused nodes, and return them to the memory pool. However, there is a performance penalty due to garbage collection and this has been addressed by many methods. For example, generational garbage collection and concurrent garbage

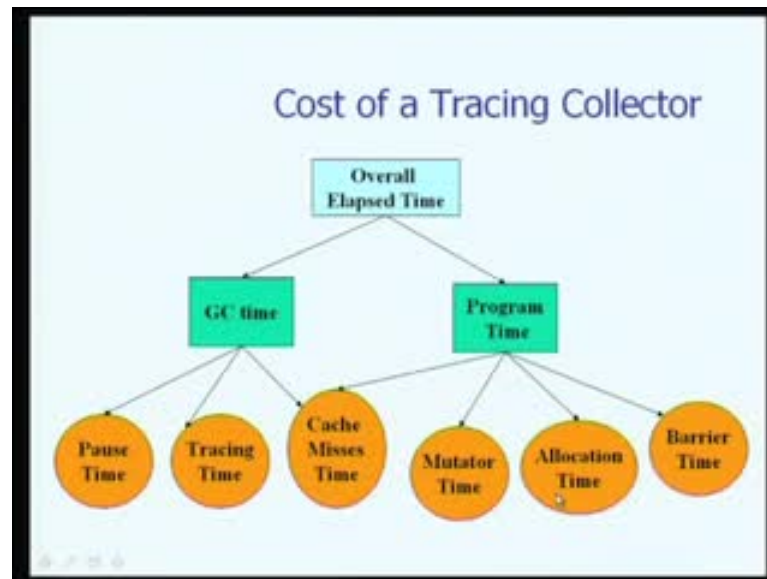
collection are two methods by which we try to reduce the overheads of garbage collection.

What is generational GC? We are going to see a little more of these in the near future. We try to arrange objects into many generations; that is, according to their age. So, some objects live very long, some objects live a very short life. If they are arranged according to their life, the younger generations, that is, the generations which store younger or short life time objects can be garbage collected frequently; whereas, the older generations need not be garbage collected so frequently. This in turn reduces the overhead of garbage collection.

Concurrent garbage collection is now the way GC is done. Basically, when we run a thread and that does all the garbage collection, we are really not coming in the way of the main program. The main program is not stopped, but at the same time garbage collection keeps happening; this is a little more challenging. When exactly should we stop the main program and how long - **it** is the question that a concurrent garbage collector has to address.

Now, cluster allocation reduces the number of collections and makes each collection more effective. What is cluster allocation? If we look at very long living objects and we are able to collect all of them together in one bunch, we do not allocate memory to those in the same heap that is used for other objects. Then it is generational in nature. So, the only thing is we need to do this cluster allocation separately. It is not exactly the same as generational, because in generational collection the aged objects are moved to older generation; but cluster allocation itself is done in a slightly different way, in a different heap structure art.

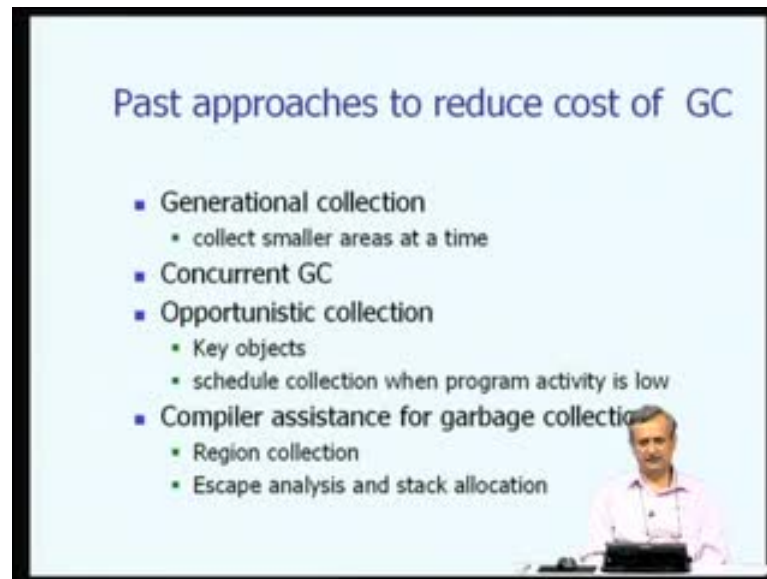
(Refer Slide Time: 04:15)



What is the cost of a tracing garbage collector? A tracing garbage collector really goes through the memory and then copies objects and so on and so forth. **The overall elapsed time and** then it consists of two parts garbage collection time and the main program time. Here we have in the GC time we have pause time, tracing time, cache miss time, etcetera. Pause time is the time for which the main program is stopped; tracing time is the time needed to go through the memory completely and determine what is garbage what is not and so on; and cache miss time is because of misses in the cache. In the program time - of course, cache miss time is a factor; mutator time is the actual program running time; allocation time is the memory allocator time; and barrier time is synchronization of thread etcetera.

The system overhead all these comprise the program time. So, when we talk about the effectiveness of a garbage collector we are not definitely going to look at the overall elapsed time for the program, but we will also, look at the pause time, because how many times is the main program stopped in between, and how long that is the pause time thus becomes very important.

(Refer Slide Time: 05:40)



What are past approaches to reduce cost of garbage collection? As we discussed just now generational collection small, we collect smaller areas at a time. The older areas are collected infrequently; younger areas are collected frequently. This definitely reduces the cost of garbage collection; concurrent garbage collection also reduces it. Then there is something called opportunistic collection in which the programmer is supposed to identify what are known as key objects. These keys objects are the roots of trees or headers of link lists etcetera.

The whole data structure is not very useful, then you know the key, if we can say all the objects pointed to by the key object can be garbage collected. Then we can schedule the collection of such key objects when the program activity is low, but this has to be done carefully. And there is a programmer intervention, a rather program specification of key objects which is required.

Compiler assistance for garbage collection - there is something called region collection. So, you take the regions of a program within that region a data structure may be created, used and then disposed off. So, if you are able to detect regions of this kind then, garbage collection can be based on that region. Escape analysis and stack allocation also very helpful. Here we intend to find out which objects actually are confined to procedures or functions, and they are not visible outside. So, such analysis called escape analysis is useful in determining objects which can be allocated on the stack, instead of on the heap. If they are allocated on the stack; as soon as, activation record dies, the space for that particular object also gets collected.

(Refer Slide Time: 07:45)



Now coming back to our scheme, we said we want to detect clusters of long living objects. Let us see, what is the effect of long living objects on garbage collection on various benchmark programs? Scavenging long life objects accounts for a significant part of garbage collection time. What is indicated here, for many benchmarks is the amount of time that is needed to scavenge such long living objects. As you can see this is a fairly large amount of time for some programs and very small for some other programs. So, the indication is some programs have long living objects and some programs do not. In such a case, it is necessary that the compiler detect whether the program has long living objects or it does not.

(Refer Slide Time: 08:36)

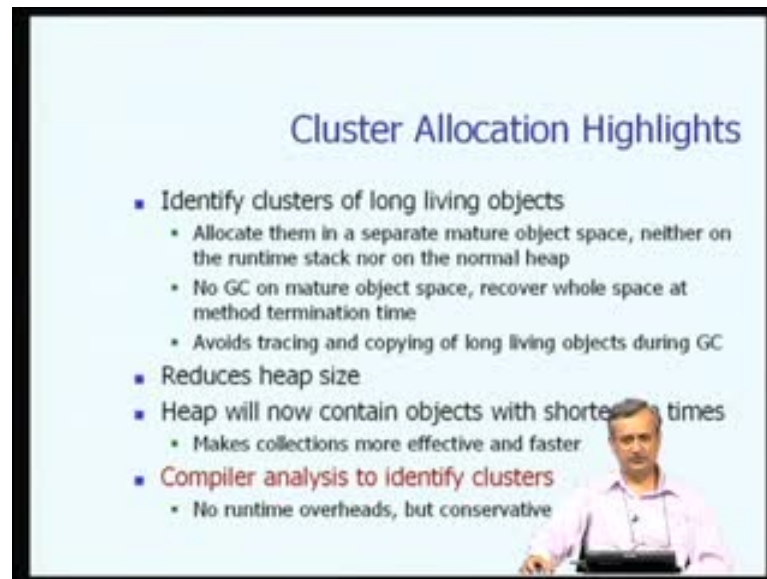
- ### Ways to Reduce Scavenge Time in Garbage Collection
- Compute life times of objects and bind them to the activation record of a method that uses them last
    - Handles volatile objects well, but not long living objects
    - Stack allocation instead of heap allocation
    - Cannot handle related objects that refer to each other from different methods, but die together (dynamic data structures)
  - Detect long-living clusters and allocate separately

How do we reduce scavenge time in garbage collection? We have to go through the entire memory find out what is garbage, what is not etcetera. Compute the life times of objects and bind them to the activation record of a method that uses them last. In other words once we know the life time of objects; there are many methods which use this object.

The last one which uses the object is can also, be responsible for disposing of the object and return it to the memory pool; so, that is the basic idea. This can handle volatile objects well, but not long living objects because long living objects you know may live through the entire program. If they are allocated on the same heap as any other object, then we would be really going through the memory which need not be disposed of every time, that is the long living objects will be scanned every time. Whereas, if we allocate such long living objects in a separate memory area; then we do not have to go through them during garbage collection. Then stack allocation instead of heap allocation is a possibility for some of the objects, but definitely long living objects do not benefit so much from this.

It is possible that we want to use both techniques that are stack allocation wherever possible; and heap allocation for wherever possible; and clusters allocation wherever possible. We cannot handle related objects that refer to each other in this stack allocation methodology. For example, dynamic data structures may refer to each other, but they die together. So, such objects cannot be allocated on the stack; they will have to be allocated on heap, but if we detect that these are long living objects, then the compiler helps in such detection, then they can be allocated in a separate area. So, the basic principle as I have been stressing till now is detect long living clusters and allocate them separately.

(Refer Slide Time: 10:51)



### Cluster Allocation Highlights

- Identify clusters of long living objects
  - Allocate them in a separate mature object space, neither on the runtime stack nor on the normal heap
  - No GC on mature object space, recover whole space at method termination time
  - Avoids tracing and copying of long living objects during GC
- Reduces heap size
- Heap will now contain objects with shorter lifetimes
  - Makes collections more effective and faster
- **Compiler analysis to identify clusters**
  - No runtime overheads, but conservative

Let us look at some highlights of our scheme:

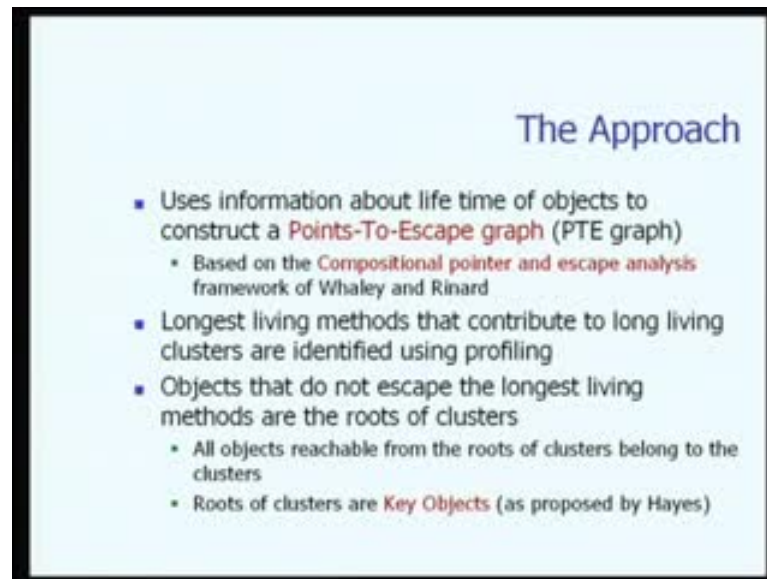
Identify clusters of long living objects - this identification is basically done using compiler analysis. Allocate clusters in a separate mature object space that is what we call them. Neither on the runtime stack nor on the normal heap it is a separate area. No garbage collection on mature object space; recover whole space at method termination time. So, you find out all the methods which actually use this mature object space, or a part of mature object space; and then once all of them terminate the last one in that sequence will return that entire area to the memory pool. It is possible again that we have long living objects which live through the entire program time, execution time or they actually live for one small part of the program; either way is possible.

It avoids tracing and copying long living objects during garbage collection; so, this is very obvious. Then allocating such long living objects in mature object space also, reduces the heap size. So, heap will now contain objects with shorter lifetimes; and makes collections even more effective and faster. So, since short life time objects are now on the heap, garbage collection now takes much lesser time. Compiler analysis to identify clusters has no runtime overheads, but it is conservative.

In other words within the cluster there may be some garbage, that means there is wastage of memory. And allocating them separately may not help in certain situations because the usage is not high. The memory requirement of the program also, may increase, but that would be only for certain types of programs.



(Refer Slide Time: 13:00)



The Approach

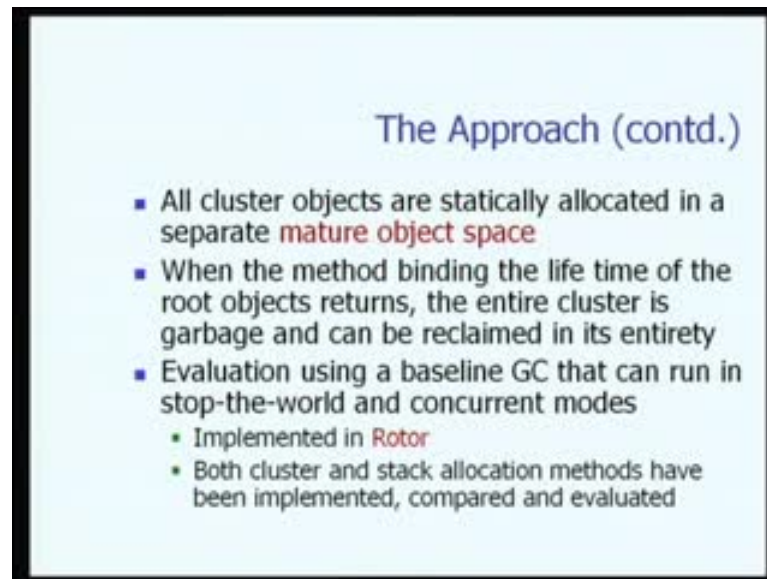
- Uses information about life time of objects to construct a **Points-To-Escape graph (PTE graph)**
  - Based on the **Compositional pointer and escape analysis** framework of Whaley and Rinard
- Longest living methods that contribute to long living clusters are identified using profiling
- Objects that do not escape the longest living methods are the roots of clusters
  - All objects reachable from the roots of clusters belong to the clusters
  - Roots of clusters are **Key Objects** (as proposed by Hayes)

What is our approach? It uses information about life time of objects to construct a what is known as a Points-To-Escape graph PTE graph. This PTE graph is well known in literature and it is based on the compositional pointer and escape analysis framework which was proposed by Whaley and Rinard a couple of years ago. So, when we construct this Points-To-Escape graph it gives us a lot of information. Let us see what information goes into this graph and how it is used. Longest living methods that contribute to long living clusters are identified using profiling. So, first of all we need to do some profiling on the program find out which methods are long living. And we also, have to find out what is their contribution to long living clusters; both are necessary.

The methods have to live long and also, the clusters to which they refer should be long living. The long living clusters are identified using this PTE graph and also, this profiling. Objects that do not escape the longest living methods are the roots of clusters. These long living methods refer to some data structures; and what we mean by escaping is that they are used within the long living methods and they are not actually passed out of this method. They die when the long living method dies that is the idea of escape. All objects reachable from the roots of clusters belong to the clusters. Once we know what the roots of these clusters are and we just do depth first search on the PTE graph and identify all the objects which are reachable and those are all in the same cluster. Roots of clusters are the key objects that I talked about; key objects can also be identified by our method.



(Refer Slide Time: 15:36)



### The Approach (contd.)

- All cluster objects are statically allocated in a separate **mature object space**
- When the method binding the life time of the root objects returns, the entire cluster is garbage and can be reclaimed in its entirety
- Evaluation using a baseline GC that can run in stop-the-world and concurrent modes
  - Implemented in **Rotor**
  - Both cluster and stack allocation methods have been implemented, compared and evaluated

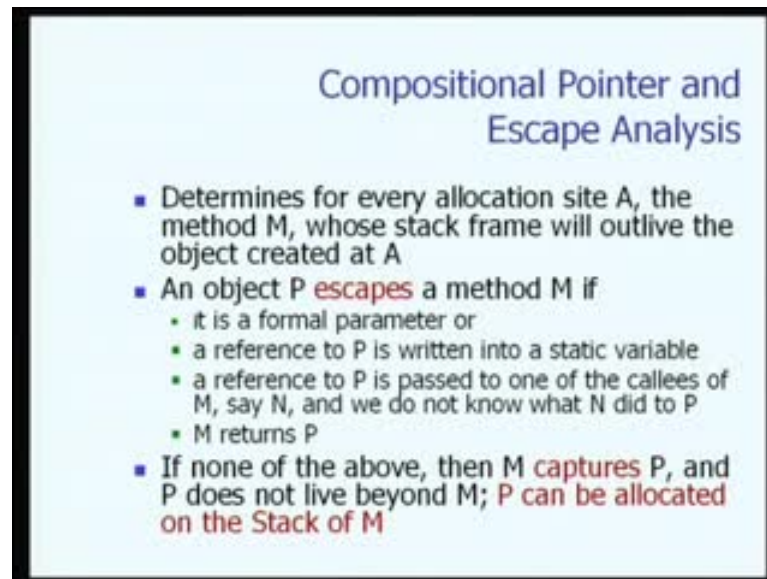
All cluster objects are statically allocated in a separate mature object space. This space is separate from either the stack or the heap and this is not collected by the garbage collector routinely. So, we actually allocate on the mature object space and we release that mature object space part when the method dies. So, the long living method dies then that area is also released. There is no garbage collection on the mature object space when the method binding the lifetime of the root objects returns the entire cluster is garbage and can be reclaimed in its entirety; so, that is what I was saying. Evaluation of our approach was done using a baseline garbage collector that can run in the stop the world and concurrent modes.

What is a stop the world GC? Stop the world GC does not have a thread running which runs the garbage collector. There is only one program that is the main program it is running when it runs out of memory as usual the garbage collector is called. Then the program is halted, it does not concurrently proceed with the garbage collector. So, now the garbage collector freezes the program corrects garbage returns all that garbage to the memory pool and then re invokes the program or restarts the program from the point where it was frozen. So, this is the stop the world program the garbage collector. And concurrent of course, I have already mentioned that the garbage collector runs in runs as a thread concurrently with the main program.

We had a garbage collector; rather we built a garbage collector that could be run in stop the world mode and it could be run in concurrent mode; and these were implemented in the rotor shade source common language interface which is available in rotor. So, both cluster and stack allocation methods have been implemented compared and evaluated.

We could run the stack allocation; and we could run the cluster allocation; and we could run without any of these as well and then compare the results that are what we really did.

(Refer Slide Time: 18:31)



Compositional Pointer and Escape Analysis

- Determines for every allocation site A, the method M, whose stack frame will outlive the object created at A
- An object P **escapes** a method M if
  - it is a formal parameter or
  - a reference to P is written into a static variable
  - a reference to P is passed to one of the callees of M, say N, and we do not know what N did to P
  - M returns P
- If none of the above, then M **captures** P, and P does not live beyond M; P can be allocated on the Stack of M

What is compositional pointer and escape analysis? It is very difficult to rather out of scope of this lecture to provide a complete algorithm for compositional pointer and escape analysis. But let me try to give you a flavor of what exactly this compiler analysis is all about. It determines for every allocation site A, the method M whose stack frame will outlive the object created A. So, we are looking at allocation sites in a method or in a program; for each one of these memory allocation sites such as stack malloc or new, we are going to determine the method whose track frame will outlive the object created at that particular A.

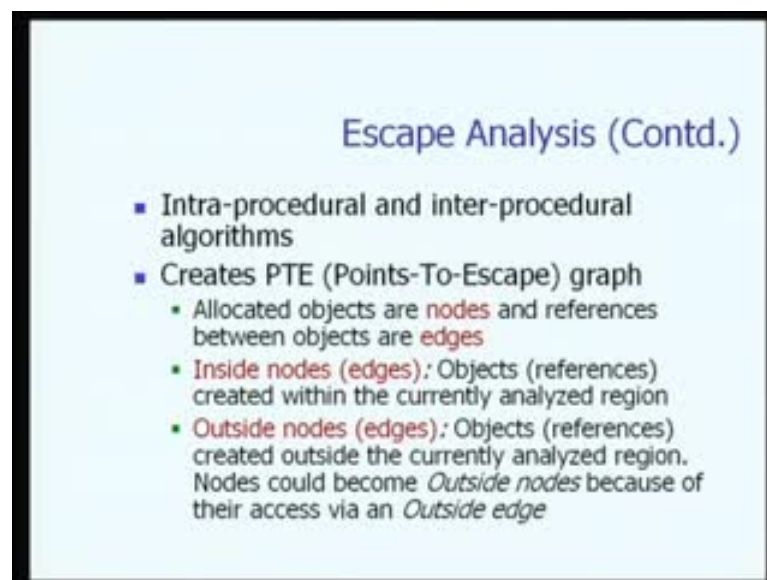
If allocation is done inside a method, then it is possible that the object is used within the method and does not dies within the method; it is not useful after that, that is what I mean. It is also, possible that this method calls a different method, and there is a series of such methods which are called; then there is a return to the original method. So, in that sequence somewhere the object that has been created originally will not be useful anymore. So, that is the method M that we are trying to find out. So, this the method M, whose stack frame will outlive the object created at A.

Each one of these methods you know M some M 1 calls M 2, M 2 calls M 3, etcetera. M 1 will use A the object created at A, it calls M 2 possibly. Then you know M 2 also, uses it and so on. It returns to M 1 and then to M; maybe then again M 3 is called. So, M 1 and M 2 actually are the methods that we are looking for, because A is used beyond the methods M 1 and M 2 as well; it is still used in M 3. So, what we want is an object P

escapes a method M, that is what we want to define. So, when does it escape? It is a formal parameter; so, it is passed to some other method; we do not know what happens to it- just going outside. a reference to P is written into a static variable. So, again we do not know who reads the static variable, and what is done with it. a reference to p is passed to one of the callees of M, say N, and we do not know what N did to P. So, a reference was passed and then something happens to it again; we do not know. And M returns the object P; so, in all these cases, we say that the object p escapes a method.

We are trying to find out if and when an object escapes a method; that is the idea behind this compositional pointer and escape analysis. If none of the above, then m captures P, and P does not live beyond M. So, P can be allocated on the stack of M; this is what I was saying. So, if there is a chain of methods you want to find out whether that object allocated at site A really goes beyond the methods or dies within the method.

(Refer Slide Time: 22:21)



The slide is titled "Escape Analysis (Contd.)" and contains a bulleted list of concepts related to escape analysis. The list includes:

- Intra-procedural and inter-procedural algorithms
- Creates PTE (Points-To-Escape) graph
  - Allocated objects are *nodes* and references between objects are *edges*
  - *Inside nodes (edges)*: Objects (references) created within the currently analyzed region
  - *Outside nodes (edges)*: Objects (references) created outside the currently analyzed region. Nodes could become *Outside nodes* because of their access via an *Outside edge*

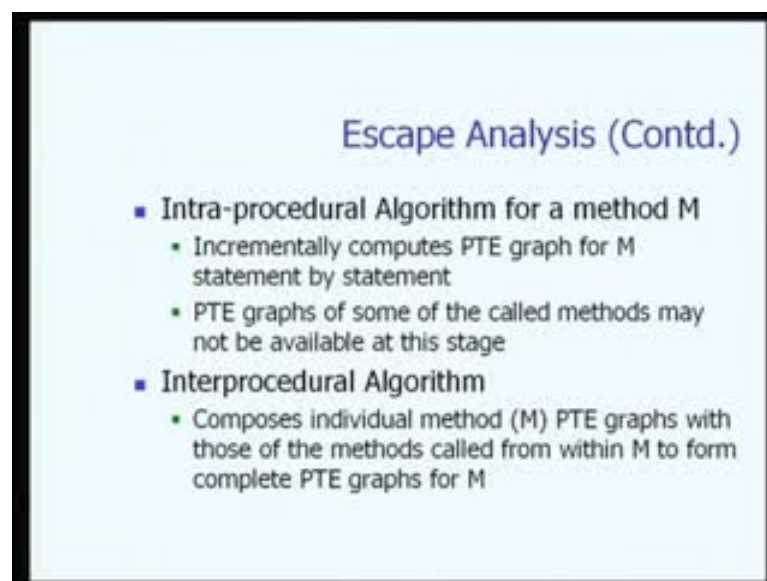
This PTE requires both intra-procedural and inter-procedural analysis. It is not possible, to do this analysis, in a pure intra-procedural way; we have not seen inter-procedural analysis so far; we will do that in one of the future lectures. This algorithm creates the Points-To-Escape graph, so, that is what it does. So, what are the nodes and objects of the edges of this particular graph? Allocated objects are nodes and references between objects are the edges, so, as simple as that. So, we allocate an object and then one object refers to another objects through various possible pointer mechanisms, these are the edges. So, there are two types of nodes and two types of edges.

There are inside nodes and outside nodes; there are inside edges and outside edges. Inside nodes are objects, created within the currently analyzed region, as simple as that.

So, you are analyzing let us say a method, we create the object within that method; then it corresponds to an inside node. What is an inside edge? These are references created within the currently analyzed region. So, again we are creating a reference to an object within a method, so, that would be an inside edge.

What is an outside method node? Objects created outside the currently analyzed region are outside nodes; and of course, what are outside edges? They are references created outside the currently analyzed region. It is possible that nodes could become outside nodes because of their access via an outside edge. So, outside edge comes first because of that the node could become an outside node.

(Refer Slide Time: 24:43)



**Escape Analysis (Contd.)**

- **Intra-procedural Algorithm for a method M**
  - Incrementally computes PTE graph for M statement by statement
  - PTE graphs of some of the called methods may not be available at this stage
- **Interprocedural Algorithm**
  - Composes individual method (M) PTE graphs with those of the methods called from within M to form complete PTE graphs for M

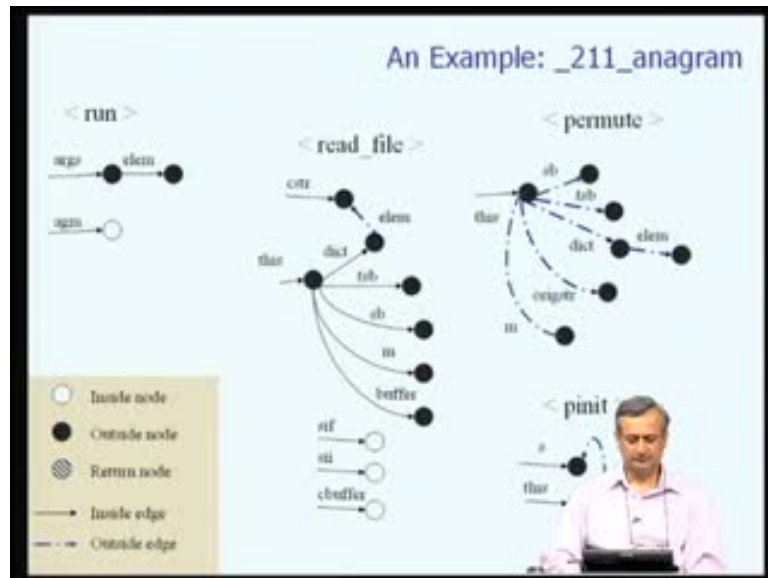
An intra-procedural algorithm for a method M for escape analysis of course; incrementally computes the PTE graph for M in a statement by statement manner. It goes through statements finds out whether they are related to memory allocation or memory reference, object allocation or object reference; and then these are the statements which contribute to the PTE graph; other statements do not contribute to the PTE graph. PTE graphs of some of the called methods may not be available at this stage. So, remember we are doing intra-procedural analysis. So, we are not going to analyze parameters trace other methods and so on and so forth.

When we are doing this analysis, obviously some information will be missing. Just like when you are doing data flow analysis for reaching definitions or something like that. If there is a sub routine call or procedure call within that control flow graph, we do not have any information about what that procedure call does, we will have to make some worst case assumptions saying it kills all reaching definitions; or something like that. So,

that is the part that we are talking about. Since some of the PTE graphs of the called methods may not be available. We either say, we will not do any interprocedural analysis, or we simply say this information will be filled in a little later. We actually fill in the information a little later, after when we do interprocedural analysis.

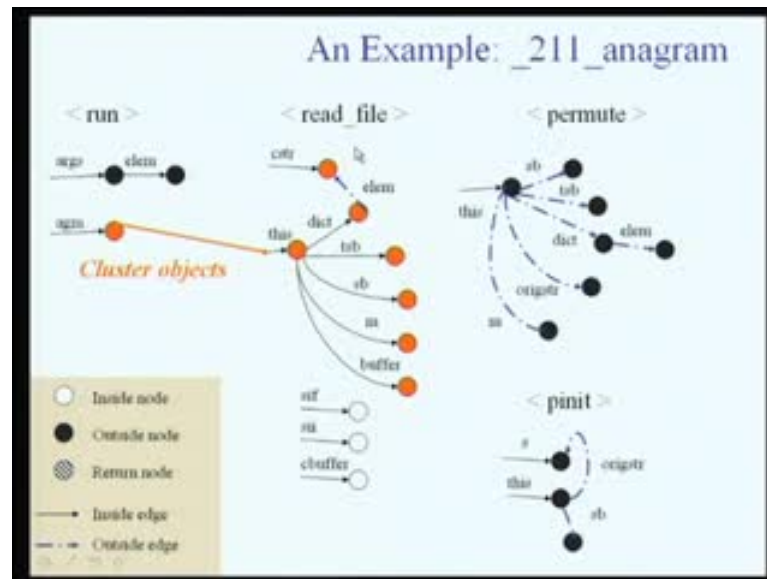
What is the interprocedural algorithm? It composes individual method PTE graphs with those of the methods called from within M to form the complete PTE graph. Basically it patches together these graphs of various methods in the called sequence; and then builds the bigger PTE graph that is the interprocedural algorithm.

(Refer Slide Time: 27:01)



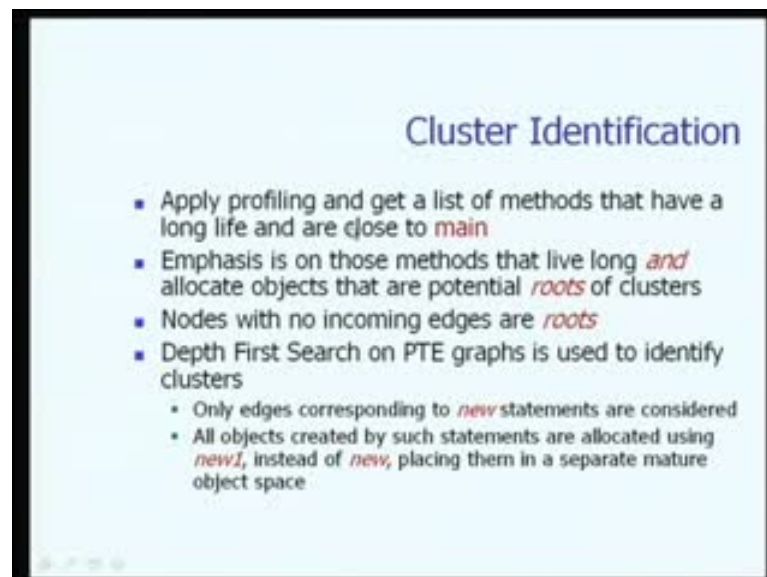
It is not possible to explain the complete construction of PTE for an example program. It suffices to know the structure of these PTE graphs. There is a program called anagram which has many of these functions methods inside. These black nodes are all outside nodes and the white nodes are all inside nodes; then there are some return nodes. The solid edges are inside edges and the dot dash edges are the outside edges. This is the structure, here there are only inside edges and then there are outside edges here and here and so on and so forth.

(Refer Slide Time: 27:54)



And suppose this node is identified as the root of a cluster, then we trace all these in a interprocedural manner; and find that these are all reachable. So, this entire set of objects forms a single cluster for this particular program. So, that is how we actually determine clusters.

(Refer Slide Time: 28:21)



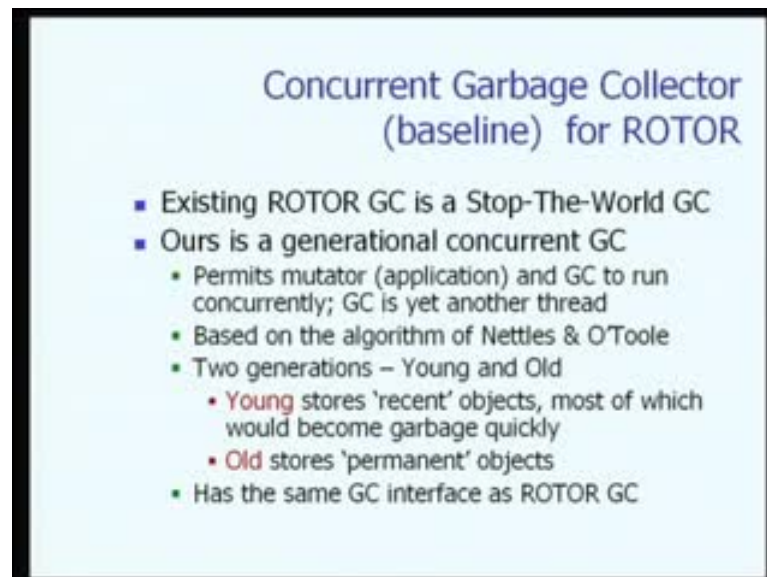
How is cluster identification done? We apply profiling and get a list of methods that have a long life and are close to the main program. From the main program we are going to actually call many methods and we do not really want methods which are deep inside the call chain from main, but we are close to the main method. So, emphasis on those methods that live long; and allocate objects that are potential roots of clusters. So, first of all the methods must live long. In other words when they return to main, we assume that



there will be a call chain from main; which calls say method A; A may call many other methods. We are really looking at the calls which are made from main because A may make several calls, but A will live until all it is calls are completed. So, A will be long living in that sense; so, that is a heuristic of course. And then not only that, the objects allocated inside these methods, the long living methods are potential roots of these clusters; they are possibly are long living objects because they are going to live until this long living method which is called from main lives.

In the PTE graphs nodes with no incoming edges are possible roots. If they are actually attached to some of these long living methods, then they become actually roots. Then depth first search from the PTE graphs are used to clusters. So, only edges corresponding to new statements are considered; new is the allocation statement; and all objects created by such statements are allocated using some other procedure called new one, instead of the regular procedure new; placing them in a separate mature object space. So, instead of trying to pass information to new and overload it, we statically replace the new statement by a statement new one, which allocates the space on that mature object space, rather than on the regular heap.

(Refer Slide Time: 30:56)



**Concurrent Garbage Collector  
(baseline) for ROTOR**

- Existing ROTOR GC is a Stop-The-World GC
- Ours is a generational concurrent GC
  - Permits mutator (application) and GC to run concurrently; GC is yet another thread
  - Based on the algorithm of Nettles & O'Toole
  - Two generations – Young and Old
    - Young stores 'recent' objects, most of which would become garbage quickly
    - Old stores 'permanent' objects
  - Has the same GC interface as ROTOR GC

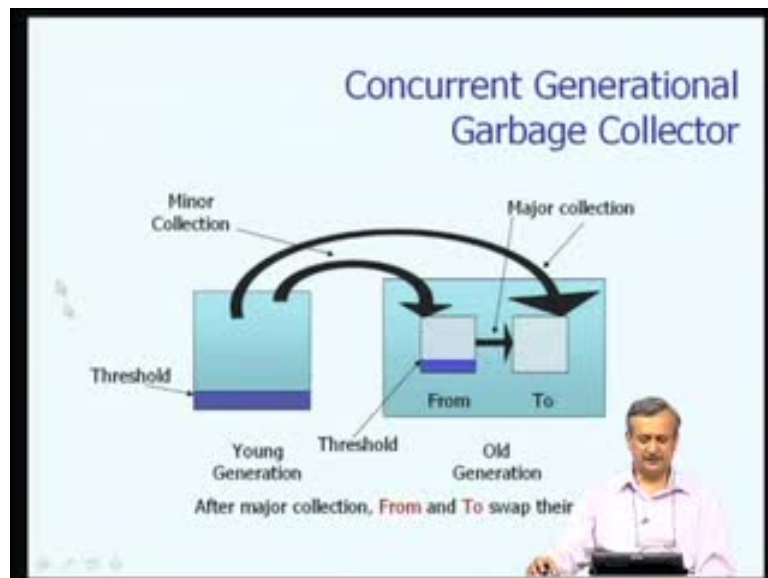
Now let us look at the garbage collector and its structure. Finally, look at how these clusters have affected the garbage collection process. Concurrent garbage collector for rotor, this is the base line garbage collector that we are using the existing rotor GC is a stop the world GC. It actually has very large pause times, whereas ours is a generational concurrent garbage collector. It permits the mutator that is the application program and the garbage collector GC to run concurrently; so, that is precisely what



concurrency is all about. So, GC is yet another thread and it is based on the algorithm of netles and o tool. So, this is a fairly well known concurrent garbage collector, so, that is what we are using. And we are using only two generations, one is the young generation and other is the old generation.

Young generation stores recent objects most of which would become garbage very quickly, so, the life time of these objects which reside in young actually is very small; that is the basic idea. We possibly temporary objects which are created or possibly nodes which are created in a link list and get deleted very quickly and so on and so forth. The old generation stores permanent objects, so, in other words relatively permanent they are long living objects; and when i say long living it is not the cluster that I am talking about. the long living object is relative to the young. So, as I said in a generational garbage collector, as the objects mature they become older and they are not deleted or released by the program, they become older and they will be moved to this permanent object space. It has the same garbage collector interface as the rotor garbage collector, so, interfacing was easy.

(Refer Slide Time: 33:14)



Here is a picture, this is the young generation and this is the old generation. Young generation holds objects with low left lifetimes; It gets filled up quickly and then it is it can be collected very quickly as well. So, most of the objects would be deleted and only a small portion of the objects are not deleted; and those are all transferred to the older generation. If they live through one cycle of one or two cycles of garbage collection, then they are transferred to the older generation.

There is a threshold set here, if the memory availability goes below this particular threshold, then the garbage collector is activated on the young generation. So, again let me repeat that if some nodes actually survive the garbage collection they are considered as old and they are transferred to the older generation. So, the hope is that they would be living for some more time; and therefore, they actually impede the garbage collection process in this by raising the overheads, so, why not transfer them to the other generation, where all its colleagues are also, slightly longer living objects. So, again let me repeat that this is not the mature object space; mature object space will be separate. It is further separate from any of these areas.

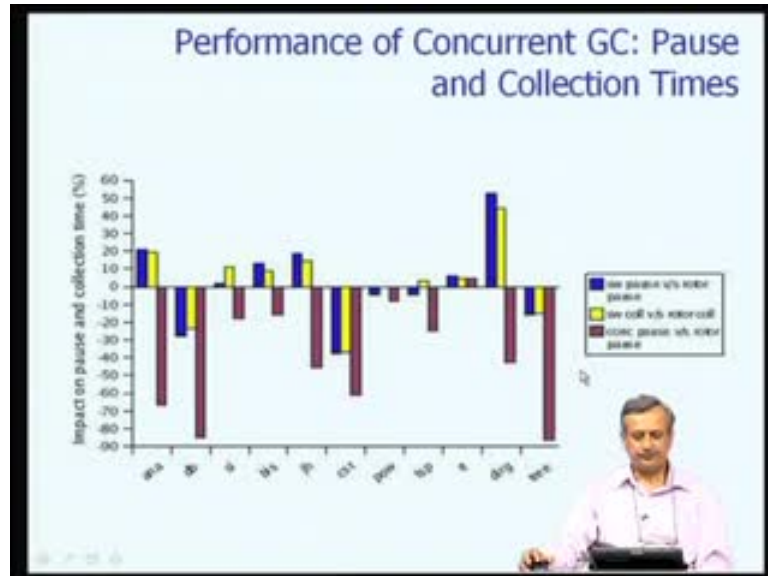
Within the old generation, we have actually two parts; one is the from half space as it is called and the other is the 2 half space. One of them is active at any point in allocation. When we transfer the objects which survive garbage collection here, they are transferred to the from space; let us say to begin with so, this is called as a minor collection. So, what we do on this particular young generation is called as a minor collection; and the living objects there are transferred to this from half space. Obviously the objects which we have transferred to this from half space, may not live for the entire duration of the program. They may become garbage at various times within the runtime of the program. Since if we do not do anything to delete some of the areas here you know nodes rather release some of the nodes here and move them to the memory pool; this phase will also, become full. When it big low goes below this particular threshold availability of memory goes below this threshold there is a garbage collection which is invoked on this from space. So, that is known as a major garbage collection.

The objects which actually are still living here are all now transferred to the to space. Objects which have died here and have become garbage are all going to be released. Now the trick is every time you have long living objects here they are transferred to the to collection and that is entire from space now becomes free. The same is true for this young generation as well. Since all the surviving objects are transferred to this from half space, this entire young generation becomes free after the garbage collection process. Now this entire thing is free and there are some objects which are living here. So, again you know transfer from this particular young generation keeps happening. So, again we do garbage collection once in a while keep transferring objects to half space.

After sometime the two half space may also become full and it may actually invoke garbage collector on the to half space. Now the roles are now reversed. The to half space becomes from space and the from space becomes the to space. So, all the long living

objects from to are now transferred to from and this entire to is free. So, from now on all the allocation for older objects happens on this to space. So, this is how the concurrent garbage collector really works.

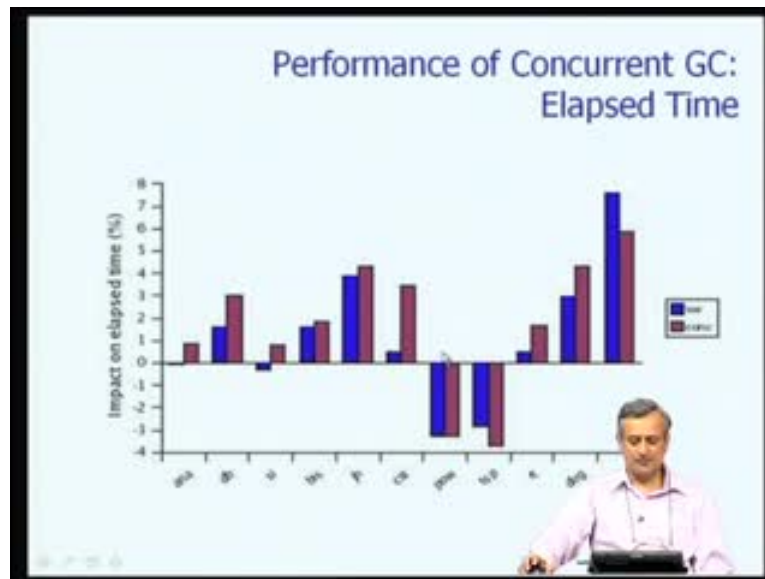
(Refer Slide Time: 37:50)



Let us look at the performance of the concurrent garbage collector. First let us look at the pause and collection times. So, this is the impact of pause impact on pause and collection time in percentage. What we have shown here, in blue is the the stop the world pause versus rotor pause. It is very clear that this blue in in general is about 20 percent here minus 30 percent here and so on and so forth. Then the yellow bar really shows the stop the world collection versus rotor collection; and then the brown bar shows concurrent pass versus the rotor pass.

Let us take the last one that is concurrent pass versus the original rotor pass. So, you can see that the concurrent pass time has really reduced to more than, in some cases it has reduced up to 85 percent and in some cases it has reduced by about 15 percent or even 5 percent. In general all of them have reduced, so, that is the indication when the bars are on the negative side. So, the concurrent garbage collector, it really improves the pass time; it reduces the pass time. So, thereby the program does not stall for long time. Whereas, in the stop the world garbage collector, program stops for quite a bit of duration, that is why most of the bars are on the positive side of the axis. Whereas, the concurrent garbage collector reduces the pass time, so, the bars are all on the negative side.

(Refer Slide Time: 40:04)



What about the impact on the elapsed time. Here again the blue bars correspond to stop the world garbage collector; and the brown bars correspond to the concurrent garbage collector. So, impact on the elapsed time in percentage is shown on this side. In many cases, the concurrent garbage collector requires you know a little more time than the other one. The elapsed time impact is a little high on this compared to the stop the world collector because the concurrent garbage collector keeps running all the time.

So, we are assuming that there is a thread running corresponding to concurrent garbage collector, but there is a single processor that we have assumed. So, it being shared by the main program and also, by the garbage collector. So, eventually if the concurrent garbage collector thread runs for a long time, it will actually stop the main program from running. In some of the cases the concurrent garbage collector has a little more overhead than the stop the world collector for example, (41:28) here here here here. And in some other cases in this for example, here the it is much lesser, so, stop the world has more effect on the elapsed time than the concurrent garbage collector. Here also, the other way; so, these are equal and this is slightly better.

(Refer Slide Time: 41:51)

**Performance of Clustering:  
Heap and Mature Object Spaces**

Average reduction in heap requirement is 12.6%

Program	Young gen/ Old gen-no clust. (MB)	Young gen/ Old gen-with clust. (MB)	Max clust. size (MB)
_211_anagram	2/8	0.7/1.4	3.8
_209_db	1/10	1/2	2.5
_208_cst	1/40	0.7/1.4	12.7
raytrace	0.8/1.6	0.8/1.6	3.6
treeadd	4/8	0.19/0.38	1.4

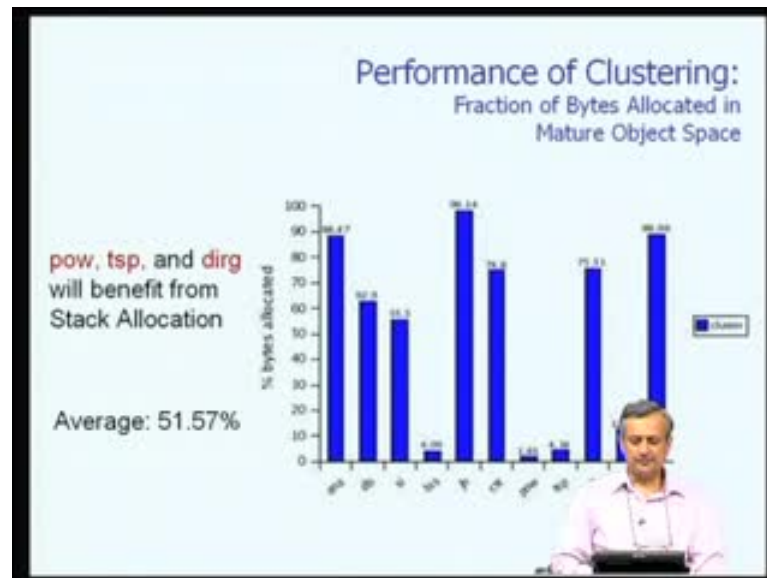
The moral of the previous two graphs is the pause time of the concurrent garbage collector is improved that is reduced whereas the elapsed time for a concurrent garbage collector increases the program completion time a little bit. What about the performance of clustering itself. So, what we studied was for the general garbage collector that we have implemented. Now let us introduce clustering and then stack allocation and see what happens. Heap and mature object space, so, we have done this for analysis for about five programs.

We are comparing the young generation versus the old generation with no clustering. Young generation was given 2 megabytes rather required 2 megabytes and the old generation required 8 megabytes for this particular program it was 1 and 10 for the second one, 1 and 40 for the third one, 0.8 and 1.6 for the fourth, and 4 and 8 for the last. The way we have to read this is the maximum amount of space that was required by the old generation or the young generation, to run the program without any core dumps and things of that kind. So, with clustering there is this phenomenal improvement. So, the young generation now required only 0.7 MB and the old generation instead of 8, required only 1.4 MB. And it is not as if this is magic, it is just that a large number of objects in the old space old generation were long living and now they have moved to the cluster space that is the mature object space.

The maximum cluster size that we have detected is also, about 3.8 MB. So, even if we add 3.8 and 1.4, we still are lower than 8 MB instead of 10 MB we require 2 MB in the second program. And then the cluster size was 2.5, so, still we have gained a lot whereas, here 1 and 40.7 and 1.4 and 12.7. So, the gain is phenomenal 0.8 and 1.6 did not yield

anything, in fact it may reverse because the cluster size is 3.6 and it adds to the old generation space. 4 and 8 became 0.19, 0.38 and 1.4; so, there is a lot of gain. In general, the average reduction in heap requirement is 12.6 percent, otherwise program specifically they can be even more than that. So, by introducing the mature object space and clustering, we have reduced the sizes of the young and old generation that we require. So, the heap size requirement has come down drastically.

(Refer Slide Time: 45:08)



So, this is regarding the memory space requirement what about the performance of clustering. So, fraction of bytes allocated in the mature object space. So, for example, the if you look at the total amount of space memory space 88.47 percent for ana was allocated on the mature object space. Whereas, for pow 1.85 percent was allocated on the mature object space. In general for a very large number of programs, the cluster allocation methodology has helped a lot. So, the average benefit is really 51.57 percent because of cluster allocation.

What about these programs which have very little benefit, basically they benefit from stack allocation. In other words, if we do escape analysis determine that the objects die within the method; then allocating those objects in the stack instead of in the heap gives a large amount of benefit; so, that is what we have found. pow tsp and dirg will benefit from stack allocation after an escape analysis.

(Refer Slide Time: 46:34)

### Performance of Clustering: Inter-region References

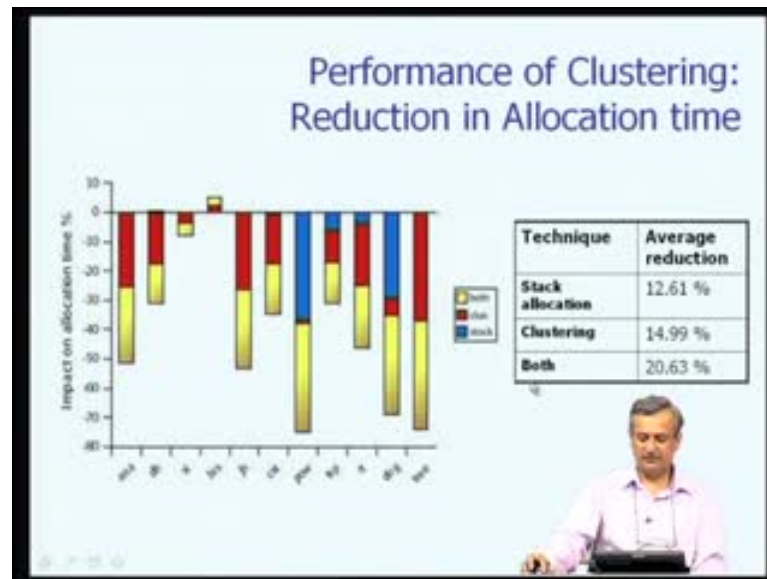
Program	Total No. of cluster to heap references	Total inter-region pointers without/with clustering	% Reduction in no. of inter-region pointers	% Garbage in cluster
_209_db	1	6916/14	99.79	11.2
_210_si	2	44731/39324	12.08	19.38
_208_cst	6	403912/169319	58.08	33.3
raytrace	3	163798/293	99.82	99.5

The inter region references from young to old, old to young etcetera. Also, reduce drastically, so, for these programs total number of cluster to heap references 1, 2, 6 and 3 total inter region pointers without or with clustering. They reduce from 6916 to 14, 44731 to 39000, 40300 to 169000, 163000 to 293. So, percentage reduction in very high in the number of inter region pointers. These inter region pointers really affect, and make the updating of these older generations very slow, so, that is where the problem is. And if something is editing into cluster from the old or the young etcetera, there is synchronization etcetera needed. So, these slow down, the updating of the objects and that is where the inter region references become very important. So, because of clustering the inter regions pointers reduced a lot.

They will remain within the mature object space or within the young space etcetera. And the flip side of it, there was garbage within the cluster which and since clusters are not garbage collected they remain forever So, 11.2 percent of the cluster in the first program was garbage, 19 percent in the second, 33 percent in the third and in ray trace actually 100 percent was really garbage. And therefore, it did not give any benefit; if you recall ray trace did not give a much benefit in other cases also, but this is a special program.



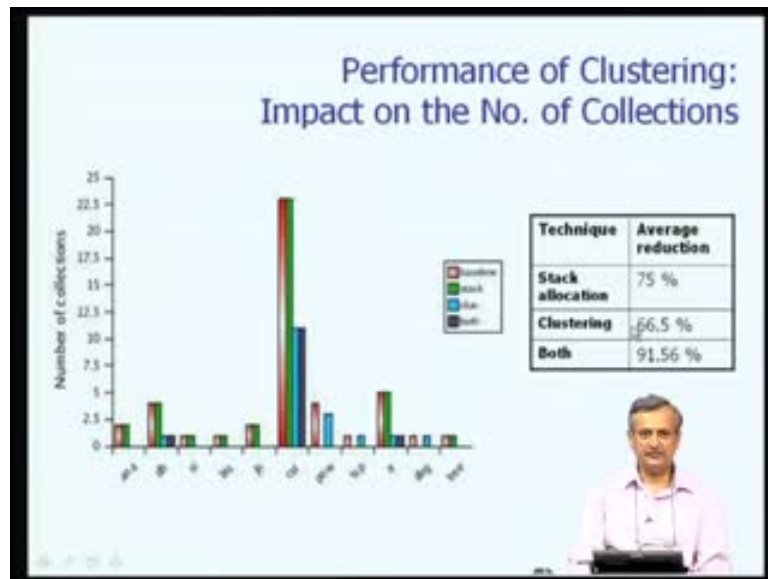
(Refer Slide Time: 48:21)



Performance of clustering reduction in allocation time: let us look at this the yellow part. It really says that by using only clustering, we are able to reduce so much. By using both clustering and stack allocation, we are really able to reduce so much. For example, let us take one with all three. So, with just stack allocation we are able to reduce so much, and then with just cluster we are able to reduce so much. And if we use both then we actually reduce so much. So, allocation time goes down drastically, if we really use both stack allocation and clustering; because stack allocation does not require allocation call at all and only clustering requires allocation call.

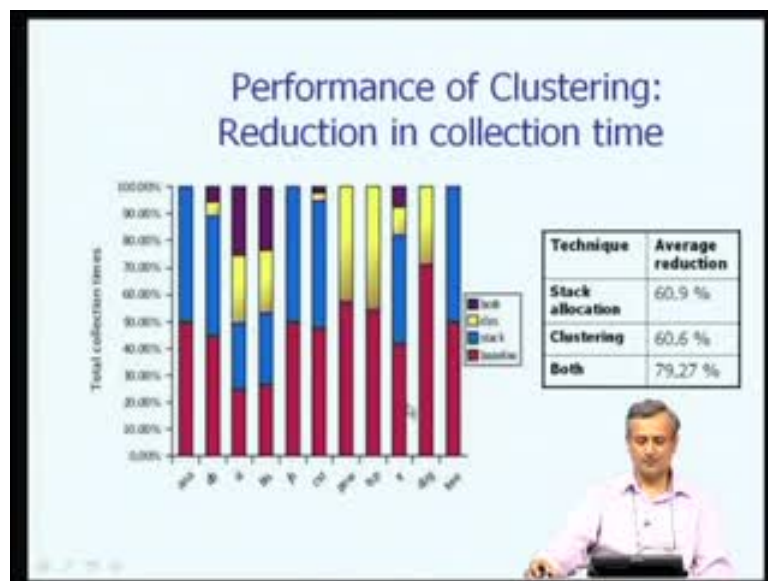
If we combine stack allocation and clustering, then the allocation time reduction happens quite a bit. So, due to just stack allocation there is a 12 percent reduction; just clustering there is a 14 percent reduction and with both there is a general average reduction of 20 percent in allocation time.

(Refer Slide Time: 49:56)



Impact on the number of collections, so, again if you look at only stack allocation there is a 75 percent reduction in allocation, this is a large number because we really do not do garbage collection if it is stack allocation. Clustering gives 66.5 percent improvement on it is own, again this is because we do not do garbage collection on the cluster. And if we deploy both, then the number of collections comes down drastically by 90 percent or more. So, in some cases like this, so, the number of collections in very large because our techniques may not be very helpful in such cases. Whereas, in other cases the number of collections has come down drastically.

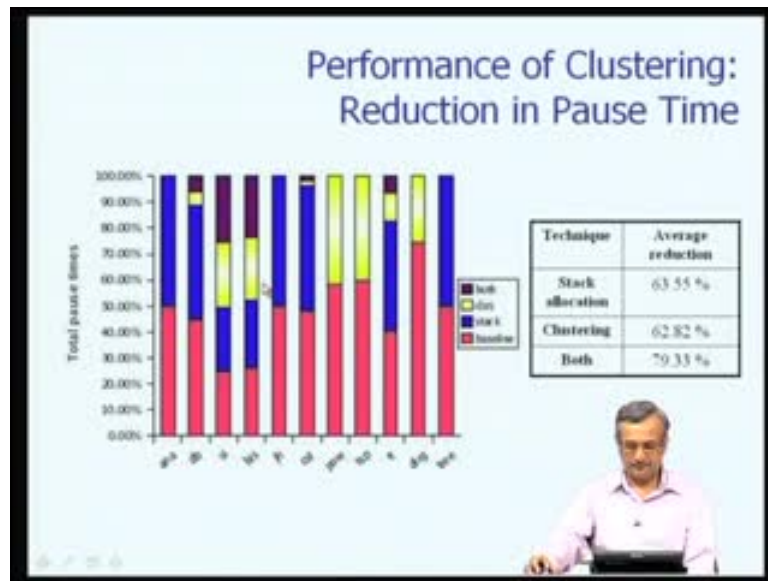
(Refer Slide Time: 50:45)



Reduction in collection time, again in the stack allocation alone reduces the collection time by about 60 percent. Clustering reduces it again by 60 percent and both reduce it by

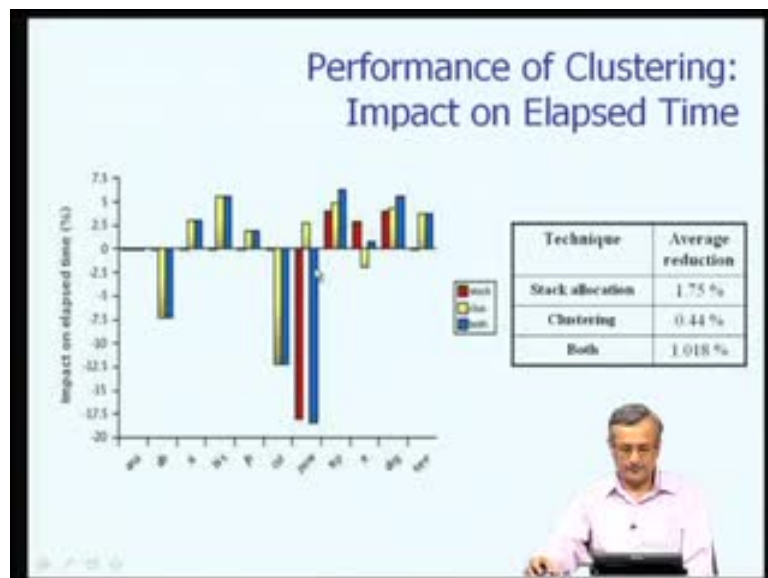
about 79 percent. So, the way we have to look at it, this is 100 percent collection time. So, what is the contribution of the base line, it is so much and what is the contribution of the stack, that is so much; and together this 100 percent reduction happens. We have to look at this blue and the violet and yellow. So, base line had so much; and then we have the stack contributing so much correction time; and this cluster contributes so much and if you have everything, then it becomes 100 percent. It is best if we really use both the techniques of stack allocation and clustering in order to reduce the collection time.

(Refer Slide Time: 51:56)



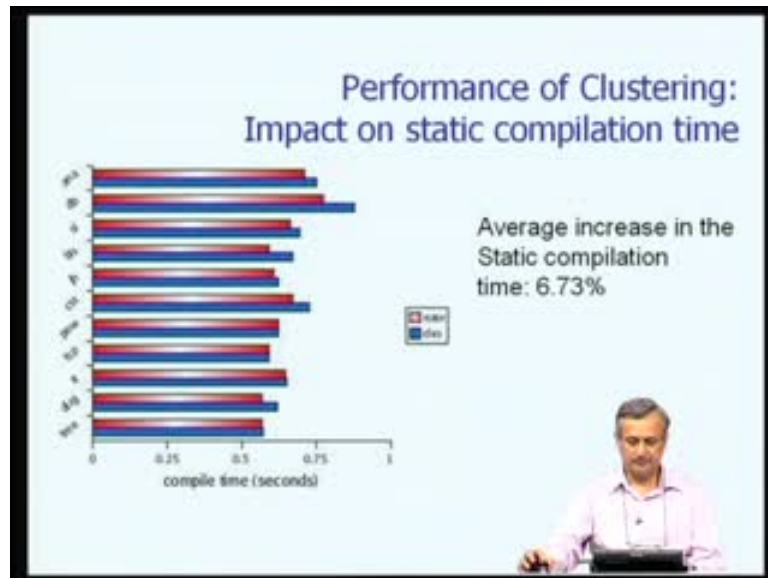
Pause time again reduces considerably; stack allocation itself reduces 63 percent. Clustering reduces it by 62 percent and both of them together reduce it by about 79 percent on the average.

(Refer Slide Time: 52:12)



The impact on the elapsed time, stack allocation the increase is average reduction in elapse time is about 1.75 percent because of clustering it is about 0.44 percent. And if we deploy both, then the average reduction is 1 percent. In other words there is not much impact on the elapse time; just because of clustering, so, it is the pass time and the number of collections etcetera that really reduce.

(Refer Slide Time: 52:45)



Static compilation time because we changed the program by introducing new one etcetera. There is some increase in the static compilation time of 6 to 7 percent but this is not really something that we need to worry about because unless they are in 20 to 30 percent range, you do not have to worry about compilation time increase.

(Refer Slide Time: 53:06)

### Limitations of the Clustering Algorithm

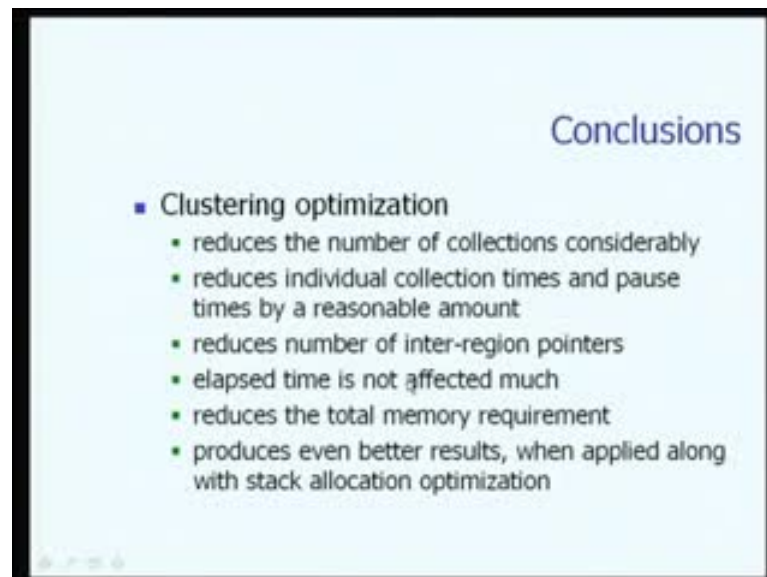
- Analysis static and hence conservative
- Cannot handle dynamically growing data structures
- Cannot handle allocation site homogeneity

```
Void X (...) {  
  if(condn)  
    a.f=Y();  
  else  
    b.f=Y();  
}
```

```
Void Y() {  
  return new clas;  
}
```

There are limitations on the clustering algorithm. For example, analysis is static, and hence very conservative. It cannot handle dynamically growing data structures they must be size limited otherwise growing ones cannot be handled. And it cannot handle allocation site homogeneity, for example, if there is only one of them is really allocated a dot f or b dot f, but it cannot really take care of such difficulties.

(Refer Slide Time: 53:40)



So, concluding clustering optimization reduces the number of collections considerably. It reduces the number of the individual collection times and pause times by a reasonable amount. It reduces the number of inter region pointers, elapse time is not affected much, so, there is neither reduction or increase. Reduces the total memory requirement, and produces even better results when applied along with stack optimization.

So, that is the end of my lecture, thank you very much.