UNIVERSITY OF GLASGOW

Relaxed Garbage Collection

Report

Emilia Vulpe

29 August 2013

Introduction

Relaxed garbage collection is an automatic way for memory management. The way it works is when a program is run, every now and then it freezes and garbage collector looks through objects in memory that are dead and deallocates them to free more memory for new objects that are needed for the program. The problem with this approach is that the process of looking for dead objects in memory is slow and may increase significantly the running time of the program. What we introduce is a new policy for deletion of objects by implementing different heuristics for prediction of when objects won't be used by the program before they die and remove them from memory rather than perform the check for dead objects every time garbage collection takes place. However, this approach may lead to introduction of errors that will not let the program run normally and a way for backup and recovery must be supported as well.

Background

Garbage collection was invented by John McCarthy around 1959. The *garbage collector* attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program. The basic principles of garbage collection are:

1. Find data objects in a program that cannot be accessed in the future
2. Reclaim the resources used by those objects

Many computer languages require garbage collection, either as part of the language specification (e.g., Java, C#, and most scripting languages) or effectively for practical implementation (e.g., formal languages like lambda calculus); these are said to be **garbage collected languages**.

Garbage collection frees the programmer from manually dealing with memory deallocation.

However, it consumes computing resources in deciding which memory to free, even though the programmer may have already known this information.

In order to get information about the objects lifetimes we used a tool called Elephant Tracks. It is an open source project and the code for it is available at http://www.cs.tufts.edu/research/redline/elephantTracks/.

It makes records when objects are allocated in memory, when they are accessed and when they die.

We used this information for finding similarities and patterns which objects die at different stages of the program to develop a tool for disposing of objects that are not likely to be used any more.

Problem statement

As an automatic way for memory management, garbage collection has many advantages but one main disadvantage that our aim was to address is the fact that when the collection takes place, the program freezes until the garbage collector does its job and identifies and deallocates all dead objects from memory which can lead to significant increase of the program's execution time. The aim of the project is to find a different way for removing objects from memory based on previously identified patterns which objects are more likely not to be used after a certain point. The problem is divided in three sub-problems: parsing and analyzing information from a trace file describing every memory access, determining the heuristics for deletion of objects before they die and analyzing the different heuristics and selecting an optimal variant.

Implementation

The program I am writing is in Java and is an open source project available on https://github.com/emivulpe/AdvancedSmartGarbageCollector

The first stage was developing a parser that reads trough a trace file produced by the previously introduced program Elephant Tracks and extracts information about different events that happen to objects in memory such as allocation, updates, method calls, and death. In order to parse the trace files I created a class ETParser that reads through a file, treats every line as a single event and parses this line further extracting information about the particular event and storing it into an instance of another class I introduced- the Event class. It store information such as the type of the event, the object id of the involved object, its size.

The second stage was to create a simulated heap memory that works the way a real memory would when a program is run. I introduced the class Heap which later became a superclass of an enormous family of heaps. The main function this class had was receiving events and updating memory accordingly. For example, if a creation event was encountered, the object was allocated in memory, if an update event-updated, and so on.

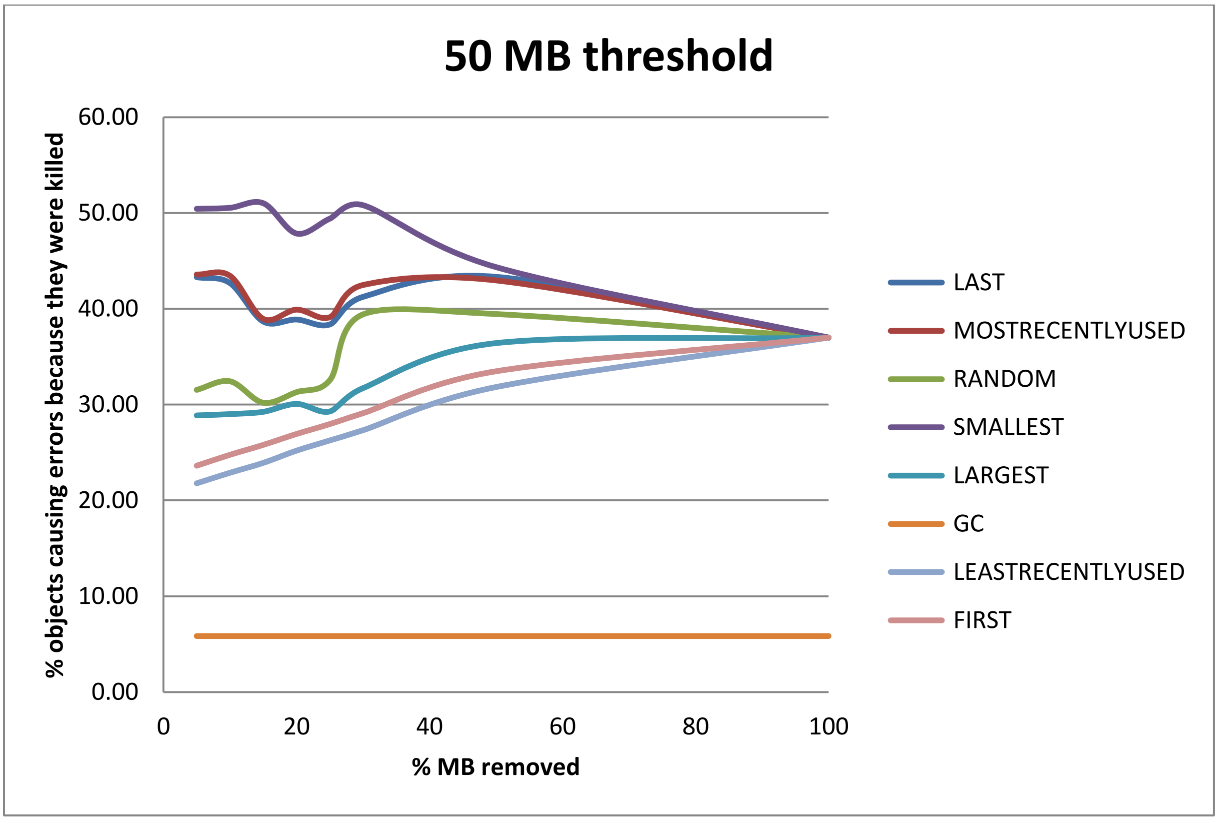
The third stage was to study the lifetimes of objects, in other words, to find patterns when objects die in relation to their size, allocation time and any other characteristic. I developed heuristics for determining which objects to remove from memory at which point of the program's execution thus not performing the usual way of garbage collection where the whole program is stopped until the collector identifies all dead objects and deallocates them from memory. Instead, I was working with different memory sizes and after reaching the threshold, I deallocated a specified % of this size based on a heuristic. The memory sizes I worked with were from 50 to 200 MB with incremental of 25 MB, the % to deallocate once the threshold is reached varied from 5 to 30 with incremental of 5% and also 50% and 100% and the developed heuristics are the following: remove the first allocated objects; remove the last allocated objects; remove the smallest objects; remove the largest objects; remove the most recently objects; remove the least recently objects; remove random objects; simulate a traditional garbage collection. I implemented the different heuristics by creating a super class SmartHeap extending the Heap class and a subclass of the SmartHeap for every heuristic. All heaps are implementing the EventHandler interface and override the handle() method that determines how the heap should behave if a creation or death event is encountered.

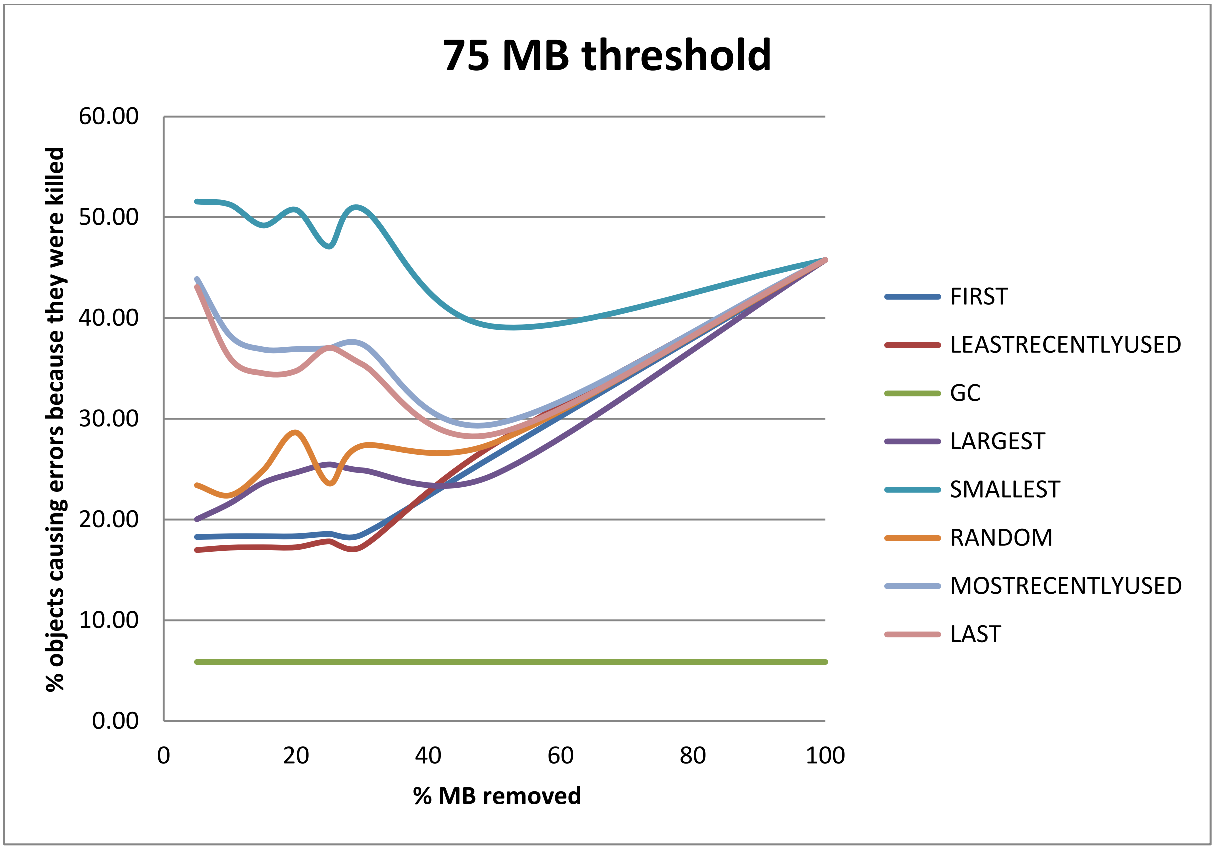
Crucial information we needed for determining how good each heuristic is was to count percentage of objects that were accessed after we removed them from memory which leads to an error and prevents the garbage collected program from its normal execution probably leading to a crash. In order to do that I introduced a class called CountDead and every object that was first removed from memory and after that accessed but not found contributed to a counter which was later used for calculation what percentage of the objects caused such an error. Our aim was to increase the percentage of objects removed from memory while decreasing the percentage of objects causing error after that.

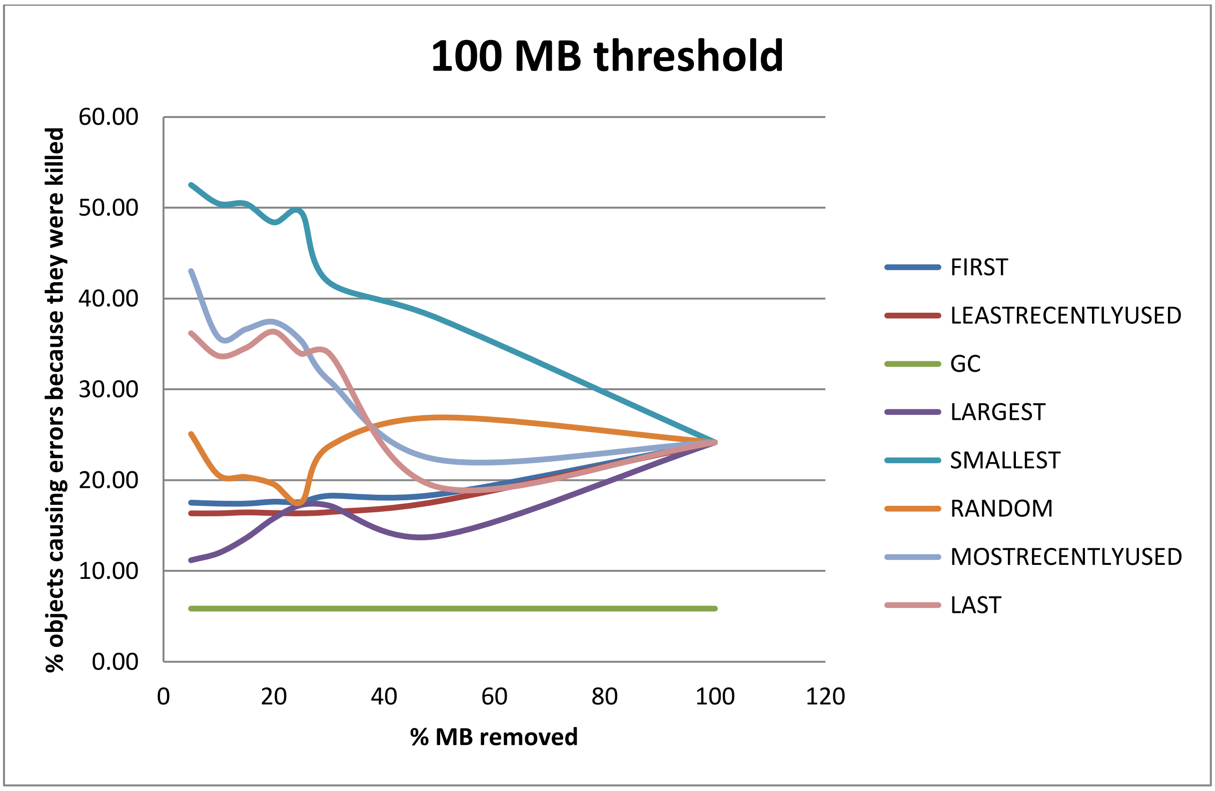
The last stage was to gather all steps described above and develop a functional unit in which the user specifies their needs without having to know details about how the program works. In order to do that in a simple enough way for the user, I created a command line parser for the user to specify their needs and according to them, the program decides which trace file to process, which SmartHeap subclass to run with what memory size, what percentage to deallocate after reaching the threshold, where to store the results and so on. After parsing the command line arguments an instance of ParameterSettings is created and the results are stored into an instance of Results class after the test is run. For running multiple experiments without specifying different needs every time I created a BatchParser file and it outputs the results for every test to a user defined .csv file for further analysis. Multiple threading was implemented for the BatchParser for parallelizing the processes.

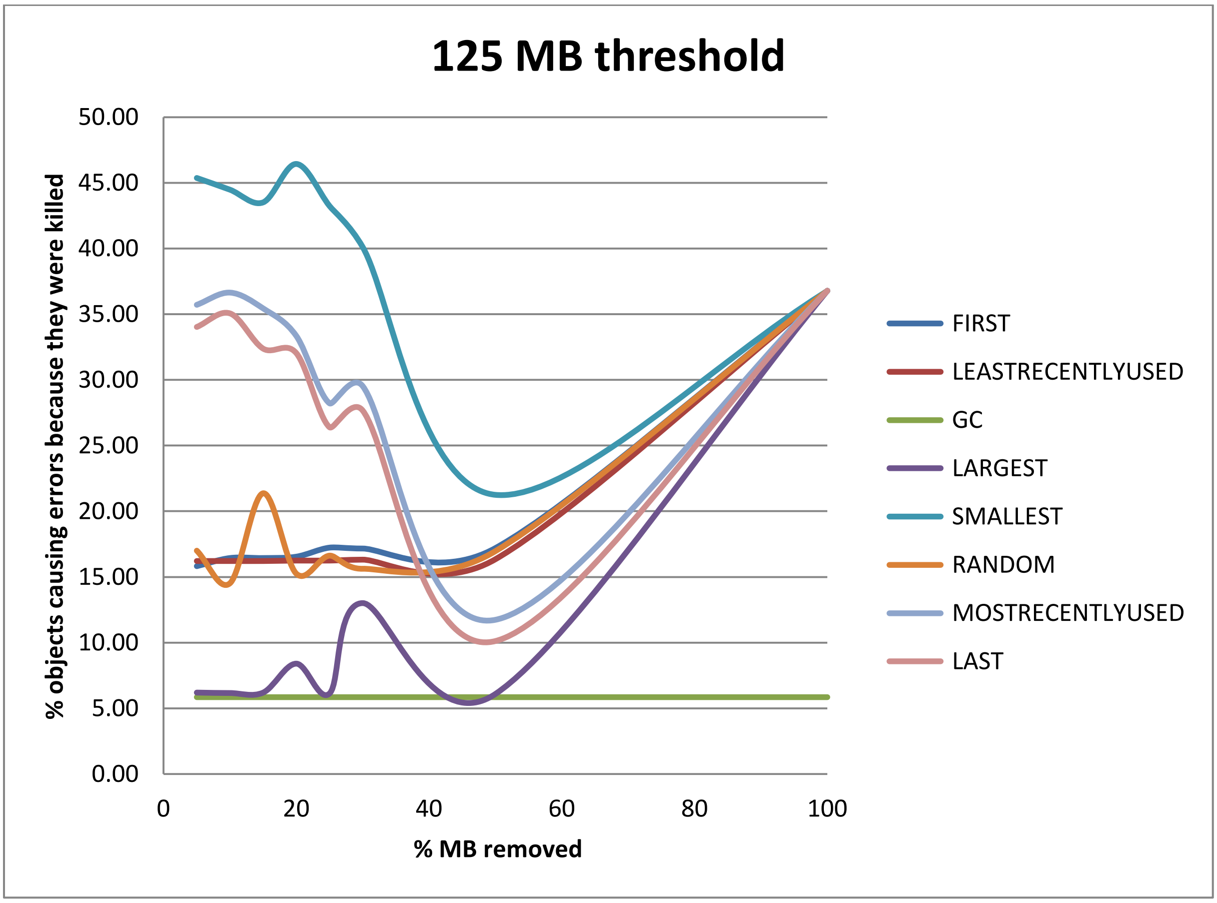
Finally, a javadoc for other developers was provided and a detailed user manual for beginners.

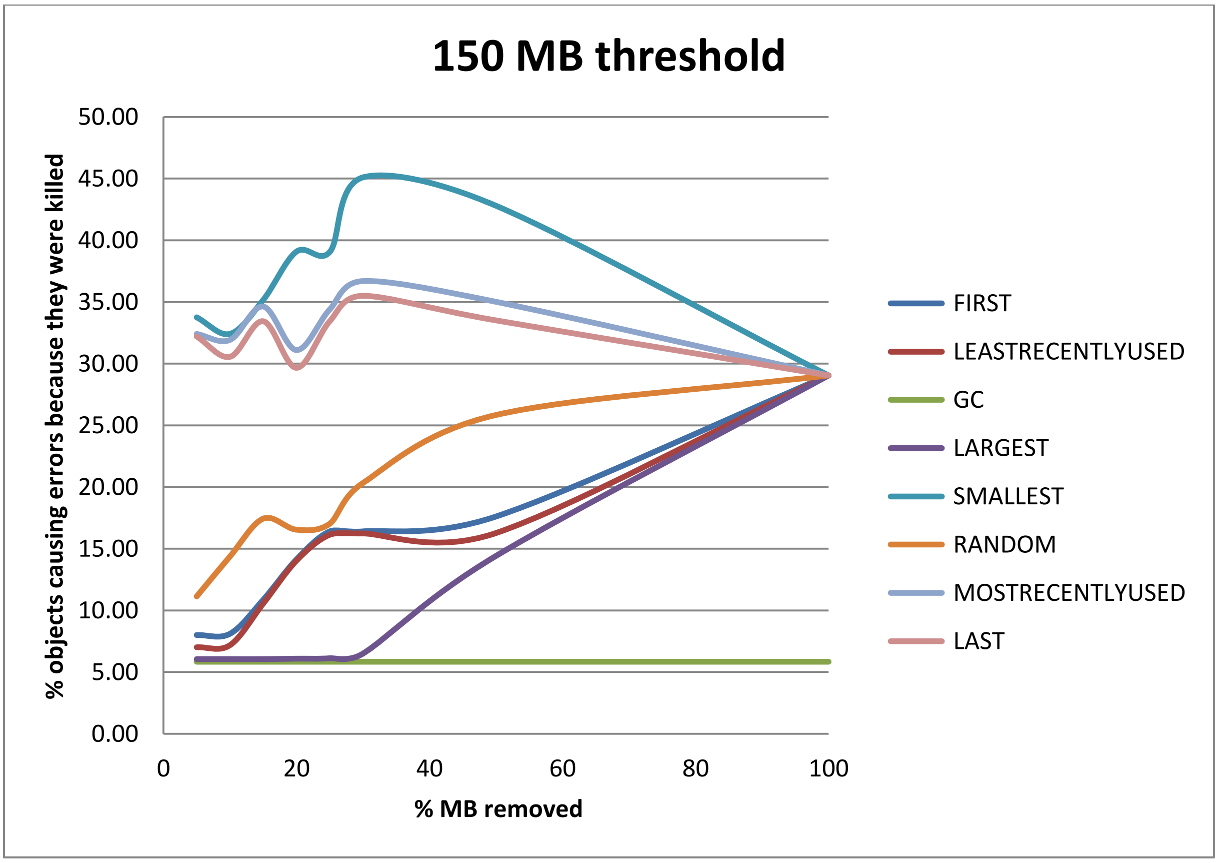
Evaluation

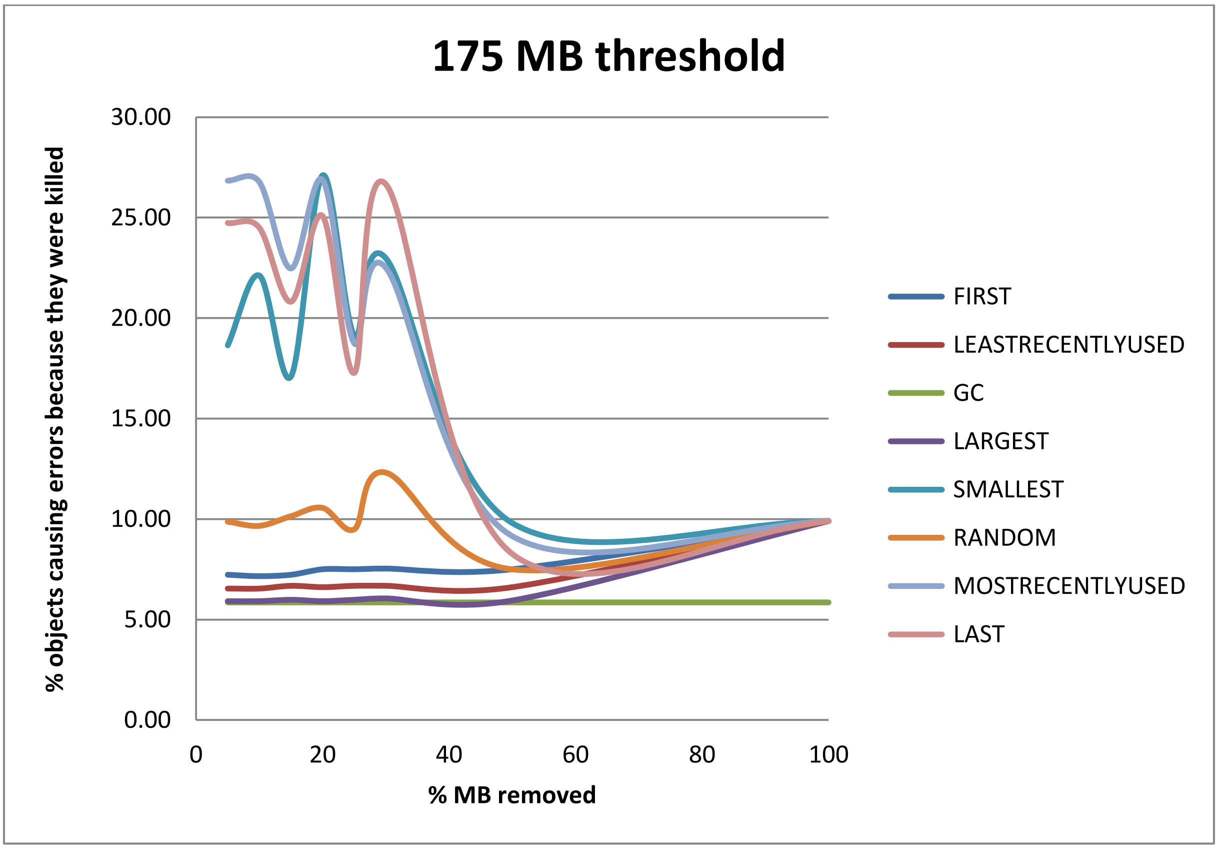


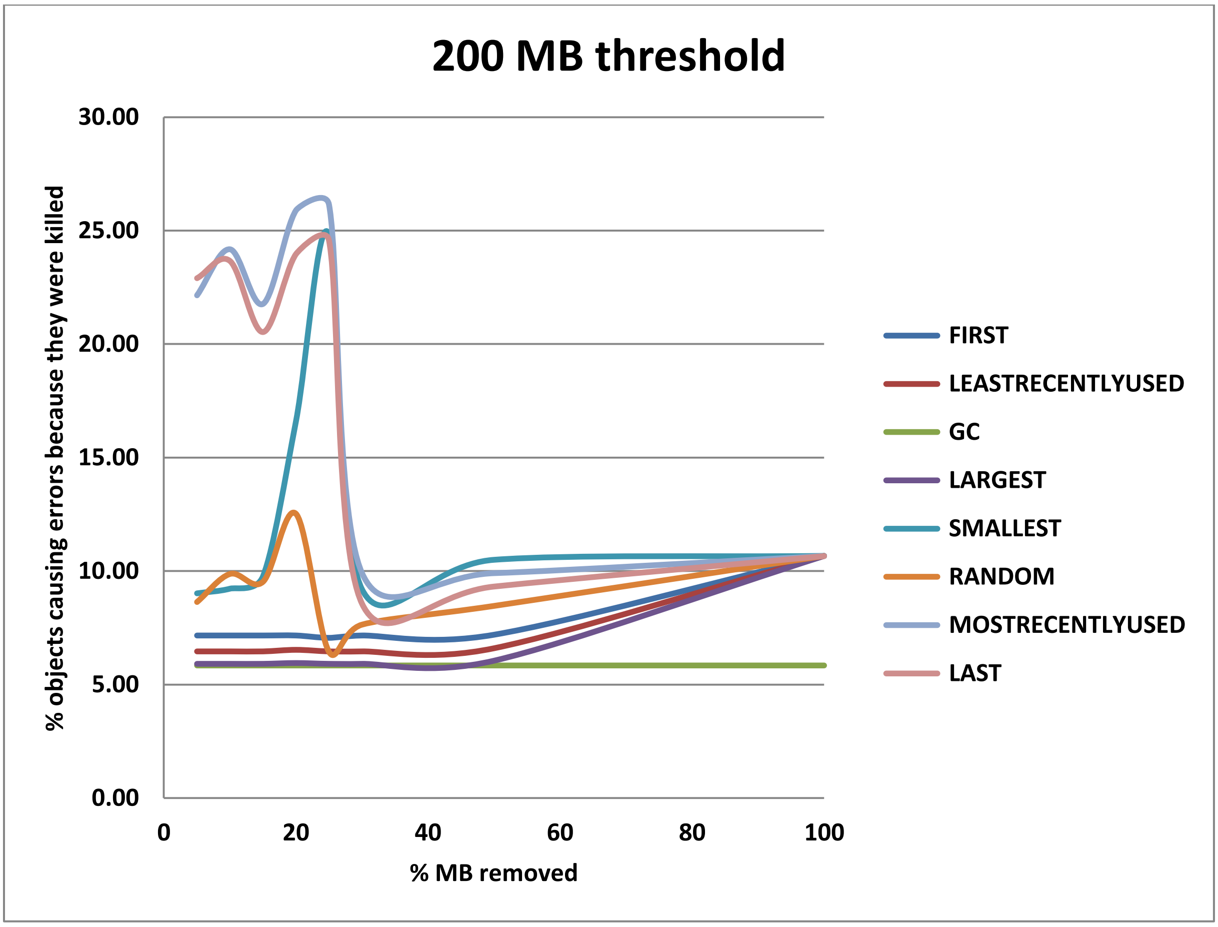

75 MB threshold
% objects causing errors because they were killed
60.00
50.00
40.00
30.00
20.00
10.00
0.00
0
20
40
60
80
100
% MB removed
FIRST
LEASTRECENTLYUSED
GC
LARGEST
SMALLEST
RANDOM
MOSTRECENTLYUSED
LAST


100 MB threshold
% objects causing errors because they were killed
60.00
50.00
40.00
30.00
20.00
10.00
0.00
0
20
40
60
80
100
120
% MB removed
FIRST
LEASTRECENTLYUSED
GC
LARGEST
SMALLEST
RANDOM
MOSTRECENTLYUSED
LAST


125 MB threshold
% objects causing errors because they were killed
50.00
45.00
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
0
20
40
60
80
100
% MB removed
FIRST
LEASTRECENTLYUSED
GC
LARGEST
SMALLEST
RANDOM
MOSTRECENTLYUSED
LAST


150 MB threshold
% objects causing errors because they were killed
50.00
45.00
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
0
20
40
60
80
100
% MB removed
FIRST
LEASTRECENTLYUSED
GC
LARGEST
SMALLEST
RANDOM
MOSTRECENTLYUSED
LAST


175 MB threshold
% objects causing errors because they were killed
30.00
25.00
20.00
15.00
10.00
5.00
0.00
0
20
40
60
80
100
% MB removed
FIRST
LEASTRECENTLYUSED
GC
LARGEST
SMALLEST
RANDOM
MOSTRECENTLYUSED
LAST


200 MB threshold
% objects causing errors because they were killed
30.00
25.00
20.00
15.00
10.00
5.00
0.00
0
20
40
60
80
100
% MB removed
FIRST
LEASTRECENTLYUSED
GC
LARGEST
SMALLEST
RANDOM
MOSTRECENTLYUSED
LAST

Above can be seen 7 graphs representing different memory sizes and testing the same parameters and heuristics every time. The x axis represents the percentage of memory size that is removed once the thresholds are reached whereas the y axis shows the percentage of objects that cause error because they were accessed after we artificially deleted them from memory. From all graphs, it can be clearly seen that the shapes of the most recently used objects and the last allocated objects tend to be very similar and the percentage of objects were accessed after their removal is very high compared to other heuristics which means that a big proportion of objects live for a short period of time. Another clearly shown pattern is that for all the graphs the heuristic that removes smallest items first shows highest percentage of objects causing errors.. Another two heuristics also show similar behavior to each other: the one that remove the first allocated objects and the one that removes the least recently used objects and they both show low percentage of objects causing errors which draws to a conclusion that a proportion of objects live a short live and are accessed only shortly after their allocation in memory and they are not used at later stages of the program. There are 2 more heuristics tested but they are control ones: the traditional garbage collection that causes constant percentage of errors and if it wasn't for the Elephant Tracks flaws, it must be 0 and a heuristic that removes random objects and it can't show any relevant results.

All these results show a couple of patterns. A stable proportion of objects die shortly after their "birth" and are never accessed later. The results also show that the proportion between the heap memory size and the percentage of objects to deallocate also play an important role for determining errors. It can be seen that optimal performance can be achieved if we have a big enough memory in the range between 150 and 200 megabytes and we can remove up to 50% of largest, least recently used or first allocated objects without introducing a lot errors.

Conclusion

Relaxed garbage collection is a research area with a bright future. My work proves that patterns, which objects can be removed from memory before their actual death, were found and analyzed so the basis of the study was founded. Using the discovery that a big part of the objects are not accessed after a certain point close to their allocation can lead to a better heuristic that addresses all these common characteristics and predicts even more accurate which objects can be removed without introducing further errors.

As a future work to be done remains recovery and back up mechanisms for restoring artificially "killed" or in other words removed from memory objects. Further tests need to be done and analyzed and further heuristics are to be developed.