Region Based Allocation in Java

Henrik Enqvist and Johan Lilius
Turku Centre for Computer Science
Laboratory of Embedded Systems
Lemminkäisenkatu, 14 A,FIN-20520 Turku, Finland
{Henrik.Enqvist, Johan.Lilius}@abo.fi

June 18, 2002

Abstract

De-allocation of memory in Java is carried out by a garbage collector. Modern garbage collectors are fast and effectively reduces fragmentation. It is, however, not always possible to use sophisticated algorithms, e.g. if the data-structures are too memory consuming. This is the case in embedded systems, the limited amount of memory forces the memory usage and the memory footprint of the virtual machine to be modest. This work will suggest a method for reducing the work of the garbage collector and decreasing memory fragmentation through *region based allocation.*

1 Introduction

In this work we will discuss three types of allocation mechanisms: *heap allocation*, *stack allocation* and *region allocation*. The Java language uses heap allocation to allocate objects. The aim of this work is to suggest improvements of the memory allocation scheme in Java by extending the existing allocation scheme with *nested regions*.

Using *heap allocation* objects can be allocated in arbitrary position on heap. An *allocation algorithm* manages a *free list*, which holds pointers to the free memory blocks. Fragmentation and allocation speed depends on the algorithm. In C heap allocation is performed with **malloc** and **free**. In Java allocation is done with **new** and the memory is freed by a garbage collector. The garbage collector is alone responsible for identifying unreferenced objects and reclaiming free memory. This utility relieves the programmer from the burden of de-allocating objects. There are, however, disadvantages when using garbage collectors. The activity of finding unreferenced objects introduces an overhead to the virtual machine. Garbage collected systems bind also more memory than an explicit allocation / de-allocation scheme. In [SKS00] Shaham shows that the excessive use of memory in some common applications is rather high.

Stack allocation is carried out in a first in / first out manner. Stack allocation is fast since allocation and de-allocation only require adding or subtracting the size of the block from the stack pointer. An *escape analysis* [Bla98, CGS+99] can determine if objects are local to a method, such objects are called *stackable* and can safely be allocated on the stack frame of the method instead of on the heap. Several implementations [GS00, Bla99] show that stack allocating objects in Java greatly reduced the overall time spent allocating memory.

In a *region based allocation* scheme objects are allocated inside regions instead of the on a stack or on the heap. A region is a portion of memory that can be allocated and de-allocated during the execution of the program. Memory occupied by objects allocated inside a region will be freed when

the region is de-allocated. The main advantage using regions is that numerous requests for memory blocks are replaced by one allocation of a large memory region. Thus, the amount of allocation and de-allocation operations will be reduced. Since the execution time for allocation is bound to the number of allocations rather than to the size of the objects, the reduced number of allocations will lead to a performance gain. A positive effect on memory fragmentation will also be noticed as small objects can be gathered in chunks instead of being spread around in the heap.

Many studies have been made on region based allocation, especially for functional languages such as SML[TT97, AFL95]. In [GA98] Gay presents a mechanism for region based allocation in C. The 'Real-time Specification for Java' [BBD+00] defines a mechanism called *MemoryAreas*. The *MemoryArea* class has four subclasses *HeapMemory*, *ImmortalMemory*, *ImmortalPhysicalMemory* and *ScopedMemory*, where subclasses of *ScopedMemory* can act as memory regions that are can be allocated and de-allocated, these regions are not affected by the garbage collector.

2 Region Allocation

2.1 Region vs. Stack Allocation

Regions can be used for stack allocating objects local to methods. Instead of allocating stackable objects on the stack frame these objects are allocated in a region whose lifetime is the same as the lifetime of the stack frame (the lifetime of the method). Such regions are easily created by allocating the region at method invocation and freeing it at method return. Stackable objects can now be allocated in the region instead of the stack frame. The first heap in figure 1 shows how a stack frame and two stackable objects would be allocated with ordinary heap allocation. The second heap shows the same allocations using when the objects are allocated on the stack frame. The third heap shows how region allocating would handle the same situation. Stack frames for methods can either be allocated on a thread specific stack or on the heap. The figure assumes that the stack frame is allocated on the heap.

If only one object will be allocated on the region, the region scheme will be slower than normal heap allocation. At least two objects or more have to be allocated in the region before region allocation will be advantageous. An optimization to this scheme would be to allocate also the stack frame inside the region. Stack allocation will be the faster than heap allocation regardless of the number of objects.

At this point in the reasoning stack allocation is a better scheme than region allocation. We will, however, observe the fact that the size of the stack frame must be calculated when class is loaded to the virtual machine. Hence, room for all stackable objects must be preserved in the stack frame. It is not sure that all objects will be allocated, and the exact size of some objects may not be know at compile time. This may result in some waste of memory. If the stack frame is allocated on the heap the size could be altered by moving the frame to a larger block or adding pointers to new memory chunks. To have several small stack frames changing their sizes is not desired. Instead of using several stack frames we try to minimize the memory spill and execution overhead by allocation stackable objects of several methods on a common resizable region. The possible overhead of resizing will now only affect one memory block. The fourth heap in figure 1 shows an example of two methods stack allocating objects. The fifth heap allocates all objects in one region. Once again the schemed could be improved by allocating the stack frames inside the region.

2.2 Nested Regions

Needed is a mechanism to allocate and de-allocate regions, we also need some way to define that several methods should use the same region. Table 1 show the operations that will achieve this.

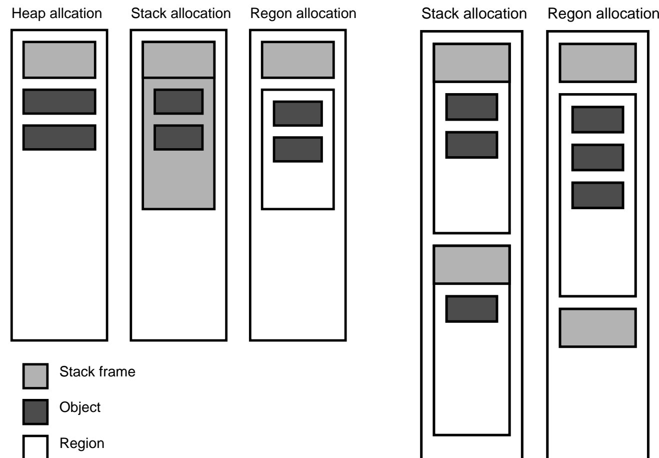

Figure 1: Heap, stack and region allocating several objects

Operation	Description
createRegion(id)	Creates a region, either by allocating space from the heap or by instantiating it as a sub-region in another active region.
freeRegion(id)	Frees the memory occupied by the region. All objects created inside the region will be destroyed.
addSubRegion(parentid,subid)	Defines that the region with identifier *subid* will be a sub-region of the region with identifier *parentid*.
newInRegion(id) object	Allocates the object in the region with identifier *id*.

Table 1: Operations

Each region has a unique identifier and each time operations are performed on the region this identifier must be given as a parameter. When a region is created the virtual machine must first ensure that the given identifier is unused. In such a case, no new region is created, instead the existing region will be used for further allocations. A region can be defined to hold a set of *sub-regions*, such a region is called a *parent-region*. A region may be both a parent-region to a region and a sub-region to another region. Cyclic parent and sub-region relations are not allowed. When appending a sub-region to a parent-region an identifier is added to the set of sub-region identifiers with the **addSubRegion** operation. Sub-regions only allocate their own memory if there does not exist a parent-region. This mechanism allows us to easily allocate stackable objects of several methods in one region. Figure 2 shows an example, using pseudo code, of how this can be performed. This method allows the lifetime of the initial regions to be modest. Short-lived regions can now use memory allocated by other regions.

2.3 Implementation of Regions

To be able to use regions we must make extensions to the virtual machine. The **newInRegion** operation is implement as an opcode **rnew**. The class files are extended with *attributes*, which define which **new** opcodes that should be replaced with **rnew** opcodes [Mun01]. If we would have directly replaced then **new** opcodes in the class files virtual machine that do not recognize the

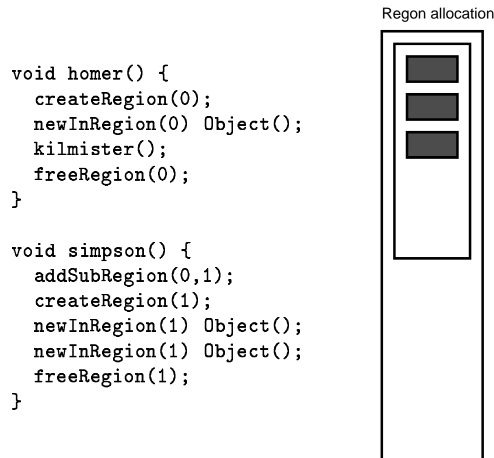

Figure 2: Sub-parent regions

rnew opcode would not be able to run the classes. The other operations presented in table 1 are also implemented with attributes, this is done to gain speed.

3 Region Clustering Based on Escape Analysis

3.1 The Analysis Flow

The input for the analysis is Java class files. The classes are analyzed with an escape analysis to find the stackable objects. These objects will then be allocated inside regions. Each method is initially assigned one region. This gives us modified class files that allocate stackable objects in regions. After this the modified class files are executed using a *test-bench*. The execution provides us with a *log file*, this file contains allocation and method invocation information that is used to build the *region clustering graph*. The size and the lifetime of parent- and sub-regions will then be optimized using the information in the log file. Finally new modified class files using nested region allocation will be created.

3.2 The Region Clustering Graph

The region clustering graph is an extended call graph with arcs labeled as active or inactive. In contrast to a call graph, arcs are also connected from a method to all methods below it in the call-chain. The graph is not computed statically by analyzing class files, instead it is created from method invocation information from executions of the program. In this way only the methods actually getting called end up in the graph. For this reason we create a test-bench that logs all method calls. The test-bench will be further discussed in section 3.3.

It is desired to define which regions of all possible sub-regions should be sub-regions and which should not. Therefore an arc is said to be active or inactive. Only active arcs are used when defining which objects to allocate in regions and which regions should be sub-regions. Figure 3 shows how active and inactive arcs affect parent and sub-region relations. Active arcs are show as solid lines and inactive as dashed lines. The **createRegion**, **newInRegion** and **freeRegion** calls are added to show how the attributes would affect the code.

```
void all() {
  createRegion(1);
  Object one = newInRegion(1) Object();
  your();
  base();
  freeRegion(1);
}

void your() {
  addSubregion(1, 2);
  createRegion(2);
  Object two = newInRegion(2) Object();
  belong();
  freeRegion(2);
}

void base() {
  addSubRegion(1, 3);
  createRegion(3);
  Object three = newInRegion(3) Object();
  Object four = newInRegion(3) Object();
  belong();
  freeRegion(3);
}

void belong() {
  addSubRegion(2, 4);
  createRegion(4);
  Object five = newInRegion(4) Object();
  freeRegion(4);
}
```

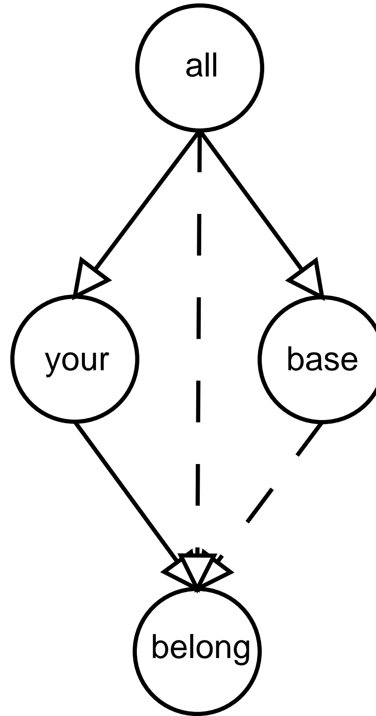

Figure 3: Region example.

3.3 Size and Lifetime Prediction

To use regions as profitable as possible they should be as large as possible without binding too much unused memory. The binding of unused memory will be avoided by clustering the regions in such a way that the size and the lifetime of the regions approaches desired values. We use allocated bytes as a measure for lifetime, observe that we don't refer to the current state of heap, instead we talk about the amount of allocation requests to the memory manager. Calculating elapsed time in allocated bytes instead of elapsed CPU cycles or seconds better fits our purpose, because the objective of our analysis is - memory. Thus, the lifetime of a region is the number of allocated bytes between the **createRegion** and the **freeRegion** call.

The size and the lifetime are calculated by running the test-bench. The test-bench is in practice a normal execution of the program. When the test-bench is run a log is created which contains region creation and object allocation information. The size of regions could also be calculated at compile time, but this would require us to make a lot of conservative assumptions. For example: as a result from Java's dynamic binding of method calls the exact type of an object is not known at compile time, therefore must a worst-case assumption be made. The size must be assumed to be the largest of all possible types for the object, thus the memory consumption will be overestimated. Neither will the size predicted by the test-bench be correct, but it will give a more accurate assumption for the size of an object. Since regions are allowed to grow, underestimating memory usage is not hazardous. A real execution may, however, call other methods and the lifetime of objects will differ from case to case. If it is not possible to create a test-bench that imitate a real execute it can not be guaranteed that the optimization of the region clustering graph will be correct.

Program	Average execution time excluding garbage collection		
	Heap allocation	Stack allocation	Region Allocation
Test 1	5.88	4.76	4.78
Test 2	3.29	2.91	2.72

Test 3		
Method	GC:s	Average block size (bytes)
Heap allocation	56	65.55
Stack allocation	20	142.50
Region allocation	14	202.80
Test 4		
Method	GC:s	Average block size (bytes)
Heap allocation	85	81.63
Stack allocation	42	185.47
Region allocation	42	196.62

Table 2: Tests

3.4 Optimization

This work uses a stochastic method for optimizing the sizes and the lifetimes. The algorithm used was a variant of a *pure random search* algorithm. A pure random search algorithm strives to find the best solution by picking a set of random samples and choosing the best among these. The goal of this work was not to find the best algorithm, an algorithm based on *linear programming* would have assured us that the optimal solution would have been found. The random search method was used because it was easy to adapt to the problem.

Because allocating long lived objects in regions may be harmful we must manually give an upper bound for the lifetime of regions. For the region size we give a limit in between which the region size must reside. We then strive to find a configuration of region size and active arcs that minimizes amount of unused memory in regions. All region size and arc configurations are calculated by simulating the memory behavior using information in the log file.

4 Simulation Results

The first test was a simple program that only allocated a number of objects. The second test had six different methods allocating objects which all could be allocated in one region. We see that stack and region allocation reduced the execution time with about 10-20%. In the second test we see that region allocation will be 7% faster than stack allocation. The speed improvements also depends the implementation on the virtual machine, some machines may spend more or less time allocation objects.

Test three and four are the same program but with altered objects sizes. The tests show how the amount of garbage collection and fragmentation will be decreased with regions. The program ran two threads, which randomly allocated objects both in regions and on the heap. We measure the fragmentation of the heap in terms of block size for free blocks after garbage collection. Region allocation showed for both tests the largest average block size. Both tests were run under configurations where regions were optimized for the third test, for this reason, the fourth test did not show as good result as the third test.

5 Conclusion

The target of this work has been embedded systems. Our approach can, however, be used in any environment using Java and may even show better results than in this case. Memory usage is currently the keyword in embedded environments. The constraints on memory force the execution size of both the application and the virtual machine to be modest. Optimization of byte code outside the embedded target allows for more complex analysis. In our approach a part of the memory management is moved outside the virtual machine. The results are highly application dependent. Not all applications suffer from memory fragmentation and the speed improvements gained due to faster allocation will only be visible in allocation intensive applications.

References

[AFL95] Alexander Aiken, Manuel Fähndrich, and Ralp Levien. Better static memory management: Improving region-based analysis of higher-order languages. *ACM SIGPLAN Notices*, 30(6), 1995.

[BBD+00] G. Bollella, B. Brosgol, P. Dribble, et al. *The RealTime Specification for Java*. Addison-Wesley, 2000.

[Bla98] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *ACM SIGPLAN POPL '98*, 1998.

[Bla99] Bruno Blanchet. Escape analysis for object oriented languages. application to java. In *ACM SIGPLAN OOPSLA '99*, 1999.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Shreedhar, and Sam Midkiff. Escape analysis for java. In *ACM SIGPLAN OOPSLA '99*, 1999.

[GA98] David Gay and Alex Aiken. Memory management with explicit regions. In *ACM SIGPLAN ISMM '98*, 1998.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object bases programs. Microsoft Research, 2000.

[Mun01] Jonas Munsin. Compile time garbage collection using escape analysis. Master's thesis, Åbo Akademi University, 2001.

[SKS00] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. On the effectiveness of gc in java. In *ACM SIBPLAN ISMM '00*, 2000.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.