Implementation of Generational Garbage Collector in SOM++ VM

Sebastian Klose and Manuel Blechschmidt

Hasso-Plattner-Institut, 14482 Potsdam, Germany
Sebastian.Klose@student.hpi.uni-potsdam.de,
Manuel.Blechschmidt@student.hpi.uni-potsdam.de

Abstract. This paper describes the contribution of Sebastian Klose and Manuel Blechschmidt for the SOM++ VM in the lecture Virtual Machines in 2009 on the Hasso-Plattner-Institut.
They tried to implement a generational garbage collector which is able to manage two heaps to speed up garbage collection based on the empirical proven thesis: "most objects die young"[1] but the didn't succeed. Garbage collection is a part of most modern process virtual machine implementation to free the developer from the task of memory management[2]. The efficient and effective implementation is critical for performance and memory consumption of the system.

1 Introduction

Every computer program needs a certain amount of memory to obtain it's assigned goal. In the beginning of computation this memory was directly accessed by its register name or address. There were problems with growing and shrinking memory and fragmentation. Then with newer programming languages and modern operating system some of these problems where solved with virtual memory systems [3].

Although the virtual memory systems made it easy to efficiently allocate memory the problem of freeing the memory again kept unsolved and the task of cleaning up the memory was given to the developer. Even if it sounds simple, freeing memory can be quite complicated. A lot of methods were introduced to help the programmer doing this job for example [4,5].

To solve this problem automatic memory allocation and deallocation strategies were developed. These methods are called "Garbage Collectors". In this paper we will describe the implementation of a new garbage collector for the SOM++ VM.

Section 1 gave a brief overview about the history and the problems which arise during memory management. Section 2 will give an overview of the basic concepts and requirements for a garbage collector. The next part 3 will explain how a generational garbage collector have been implemented for SOM++. Afterwards in section 4 we will discuss the performance gains. In Related Work 5 other ways of garbage collections are mentioned. 6 will introduce some task which could speed up memory management more. We will end with the Conclusion in section 7.

2 Garbage Collectors

A garbage collector is a part of a computer programing most of the time of a virtual machine running from time to time which tries to find unused memory regions and returns them to the superior system. The most Garbage Collectors are running when the memory is full. They will stop the system and then search for unused objects. Garbage Collector which stop all other executions when freeing memory are called stop-the-world Collectors.

The platform which we are referring to in this paper is a virtual machine implementation for Smalltalk called SOM++ so most of the memory is consumed by objects which are represented in memory. The other memory is consumed for example by the object table or the interpreter itself. These parts will manage the memory for their own. The garbage collector will work on the object representations created by the small talk code in the memory. This memory is called heap. When an object won't be used anymore the memory which it consumed can be reclaimed. The challenge for the implementation is to figure out with few resources as possible if this is the case. The implementation that we are describing will use an generational approach.

Most of the collectors start to run when the heap is full. This makes the run of a garbage collector unpredictable and most of the systems which have garbage collectors loose the ability for real time applications [6].

3 Implementation of Generational Garbage Collector in SOM++

The next paragraphs will described the SOM++ virtual machine and especially the memory manage part followed by a description of the new collector.

3.1 SOM++

SOM++ is a virtual machine for smalltalk which is completely written in C++. The smalltalk meta model is implemented as an own set of classes written in the smalltalk dialect supported by the platform. The system was developed at the HPI with the goal to have an experimental VM for research and teaching purposes. SOM++ is platform independent and can run on Windows, Linux or Mac OS X environments. It contains a complete virtual machine environments with the following features:

- Stackbased virtual machine
- Bytecode compiler
- Bytecode interpreter
- Own implementation of smalltalk meta model
- Automatic memory management (Mark-And-Sweep Garbage Collection)
- Testsuite
- Benchmarksuite

– Examples
– more stable and experimental features

It was primarily written by Arne Bergmann, Tobias Pape and Michael Haupt and is now extended by a groups of students for studying virtual machines.

3.2 Traditional Mark-And-sweep GC

The first implementation of SOM++ has a traditional and naive mark-and-sweep garbage collector[7]. It has one heap and all objects are allocated on that heap. The heap and the garbage collector are located in the Heap.cpp and the GarbageCollector.cpp files in the memory folder.

When the vm starts up some objects are defined as the so called gc roots. These gc roots objects will never be removed from the heap and are used as the starting point for garbage collection. Typically these objects are global and important objects. In figure 1 you can see the system object and the Frame class which are 2 of the many roots.

Most of the roots are set in the Universe.cpp class. Some examples of roots can be found in the initialization method $Universe :: initialize()$.

A mark-and-sweep garbage collector works in 2 phases in the first phase all reachable objects from the gc roots are marked and then in the second step all objects which are not marked are deleted from memory.

The mark phases uses a recursive approach to go through the heap the sweep phase uses a linear algorithm. The algorithm itself is implemented in the GarbageCollector.cpp class. There are 3 functions which work in conjunction to freeing memory.

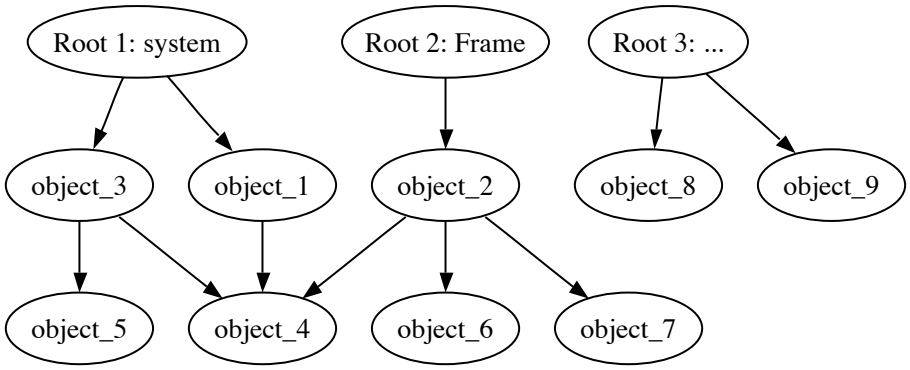

Fig. 1. Subset of objects reachable from gc root nodes.

GarbageCollector :: *Collect*() this function is called to start the garbage collection run. After setting some initial variables it calls *GarbageCollector* :: *markReachableObject*(). This procedure cares that all objects which are still reachable will be marked. This works by first taking all entries from the globals array and then calling $MarkReferences()$ for the first and for the second entry. Recursively all gcFields of reachable objects will be marked by assigning a 1 to the gc field. Afterwards the $MarkReferences()$ function on the current frame will be called which will recursively mark all stack frames and all referenced objects.

The next thing what the *Collect*() function does is going through the heap freeing all the unused objects and saving informations about the space in a double linked list called freeList of VMFreeObjects.

In the last step of the classical collector the free cells in the heap which are saved in the freeList are merged. This works by going sequentially through the free objects and check if the current entry of the freeList plus it's object size points directly to the beginning of the entry afterwards. If this is the case, the following entry will be removed and the current will be extended by the removed size and associated with the second one.

The pseudo code in 1 illustrated a normal implementation.

Algorithm 1 Simple mark and sweep implementation for a garbage collector

```
function startGarbageCollection(){
roots ← getAllRoots()
for root in roots do
  mark(root)
end for
sweep()
}
function mark(node o){
if NOT o.marked then
  o.marked = true
  for referencedByO in o.objects do
    mark(referencedByO)
  end for
end if
}
function sweep(){
for o in allObjectsInHeap do
  if o.marked then
    o.marked = false
  else
    delete o
  end if
end for
}
```

The implementation have to care for atomic allocations and when the memory becomes short the garbage collector has to run reliable and make create enough space for the allocation.

For atomic commits *StartUninterruptableAllocation*() and *EndUninterruptableAllocation*() are provided by the heap to tell him that he shouldn't start a collector run until the section is done. When *StartUninterruptableAllocation*() is called on the next object allocation the heap checks if at least 10% of the heap are still free. If no a collection run starts.

Besides errors in the implementation the mark-and-sweep approach has many other disadvantages. The memory becomes fragmented and it is hard to find new parts with at least specific size. The reference counting group found that the SOM++ VM is especially slow when it has to find free cells in the heap [8]. The one that we want to reduce and optimize in this paper is the allocation of new space by splitting the heap in two parts were the first part will be used a lot more often then the second one.

3.3 Generational GC

A generational gc[9] manages two heaps one nursery heap which is fixed in size and one mature heap which can grow if it exceeds it boundaries. All new space for objects will first be allocated in the nursery heap. When the nursery heap is full a garbage collector run is started. The garbage collector will use a mark-and-sweep technique to figure out which objects aren't used anymore. The mark-and-sweep can run in two modes. One mode only traverses the nursery heap and the other mode makes a full mark-and-sweep run and uses both heaps.

It expects that most of the objects in the nursery heap will only live a short time so most of the object won't get a mark. All the object which get a mark or are referenced by objects in the mature heap will be moved in the mature heap. When the mature heap becomes full during this action a full garbage collector run in started and afterwards a compacting algorithm will go through the mature heap and care that fragmentation won't happen. Then the remaining objects can be moved.

The garbage collector will work according to the following algorithm:

- Go through nursery heap and mark all objects which are used with a 1
- Move all objects which have a mark or are references by mature objects to the mature heap (update references in object table)
- If the mature heap is full do a complete collector run
  - Mark-and-sweep over both heaps
  - Sweep and compact on mature heap
  - Copy left objects from nursery heap to mature (update references in object table)
- Clean nursery heap

It is a challenge to set a good size for the nursery and the mature heap. It is a good idea to set the nursery heap to a size which fits in the Level-2 cache of modern processors.

Because of the moving character of the GC it is necessary to change the pointers after a moving operation. For implementing this behavior there are different methods. In our implementation we are using an object table which was implemented by another group [8].

During the implementation of the generational GC a lot of bugs in conjunction with atomic allocations of multiple objects. We would guess that we found about 5 places where that happened. Further the object table could become inconsistent if the garbage collector run at the wrong time. Errors like this make the implementation of a garbage collector quite complex because the implementor of the garbage collector has to trust that the other vm developers are allocating their objects in the correct order and enclose them with uninterruptible calls.

If this is not the case debugging can be quite complicated because the vm will crash when one of the unnecessary cleaned objects is accessed. Unfortunately this access and the freeing are not connected and it is pretty hard to find and fix the missing reference. This is also the main reason why the outcomes of the implementation are unsatisfying.

Object Table 2 shows how an object table maps the variables which appears in the smalltalk code to the data which is located on the heap.

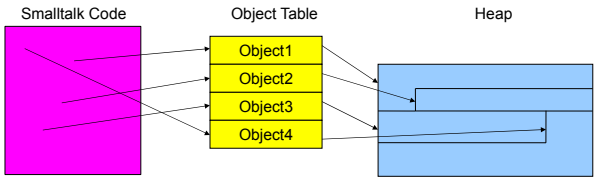

Fig. 2. Illustration of pointer into object table and from object table into heap.

The implementation of the object table is a singleton and provides an interface where an object can be added ($ObjectTable :: AddObject(obj)$) and removed ($ObjectTable :: RemoveObject(index)$). Further a method was added to update a reference pointing to an object ($ObjectTable :: updateObjectReference(index, newObj)$). The internal data structure which holds the entries of the table is a vector from the c++ standard library. The memory for the object table is separated from the heap and grows for itself.

All the time when a new VMObject is constructed it will also create an instance of a VMPointer. This VMPointer is the connection from the object to the object table. It holds the index of the object in the object table. The pointer will be saved in the self_pointer variable of VMObject.

When during a garbage collector run an object is removed from the object table. The object will be replaced with an int which holds the last free position

in the object table. These free entries do point to each other. This is illustrated in figure 3

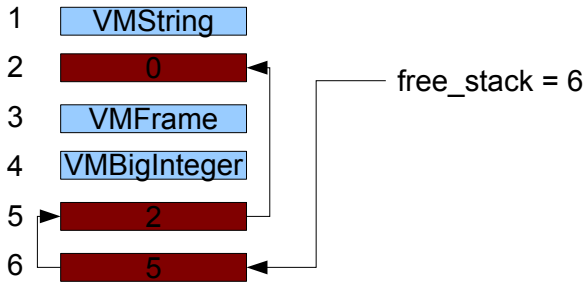

Fig. 3. Free spaces in the objecttable are filled with ints pointing to the next free space.

We don't know if a self growing vector is the fastest implementation. Memory allocation could slowdown the vm until the object table has reached it's biggest size but we guess that this doesn't matter. If we could guess how many objects fit in the heap we could set the object table size according to this information. In some tests we figured out that most of the objects are between 20 and 50 bytes in size. There are only a few ones which are a lot bigger. We could divide the heap size by 40 to get a good starting size for the object table. When the heap is 1 mb, 26.214 is a good starting size for the object table.

During the implementation it could happen that the garbage collector runs when an object was not yet added to the object table. This caused the garbage collector to free the first entry of the table which holds an null pointer. The next time the null pointer is accessed the system will crash.

Heap The Heap was divided into two Heaps. The nursery heap and the mature heap. According to the names the smaller one was implemented in the Nursery-Heap.cpp file and the bigger one in the MatureHeap.cpp file.

The interface used before which was defined in the Heap.h file stayed intact and was reused in the implementation. Therefore it shouldn't be necessary to make any further changes on source code besides the memory folder. The default Heap Size for the nursery heap was set to 1mb and the mature heap to 100mb.

Moving Garbage Collectors can be divided into two classes Moving and Non-Moving. A moving gc will copy the objects which are on the heap during the garbage collector run. Exactly this is the case when the generational collector moves objects from nursery to mature and when compacting the mature heap.

The moving is implemented in the *GarbageCollector* :: *Collect*() and in the Compaction in *GarbageCollector* :: *compact*().

When moving an object the memory is copied to the new location and afterwards the pointer is updated.

Compacting Like we already described we are using an compaction algorithm to always keep a perfect aligned mature heap. In our case perfect aligned means that all objects which are alive means reachable by the gc roots are on the beginning of the heap and afterwards everything is free.

After a full marking run is completed the compaction algorithm should start. This is happening in *GarbageCollector* :: *compact*()

The compaction for the mature heap goes sequentially from the beginning of the heap through the objects and moves all alive objects as close as possible to the previous alive object. This can be seen in illustration [**?**].

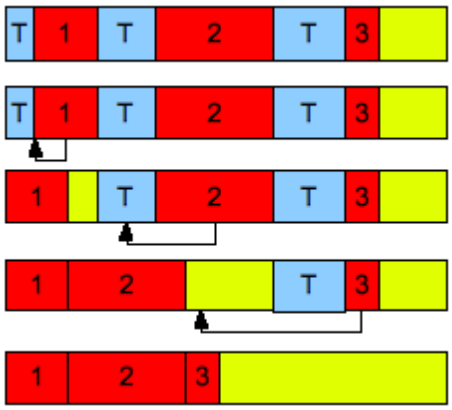

Fig. 4. Illustration of compaction run.

References from Mature to Nursery Heap One thing which was sometimes mentioned but never explained is the problem when a mature object references an object in the nursery heap because we get the moving generational garbage collector never to run we decided that we will implement a non moving garbage collector with this feature from the trunk. The problem itself is shown in figure 5. Our solution is based on the remembered sets approach shown in [13]. Basically all the time when a new reference assignment is done. We check if this assignment goes from an old to a new object. If this is the case. We save that there is that assignment. Later we can use these counted references in the garbage collector run to save nursery objects which are still used.

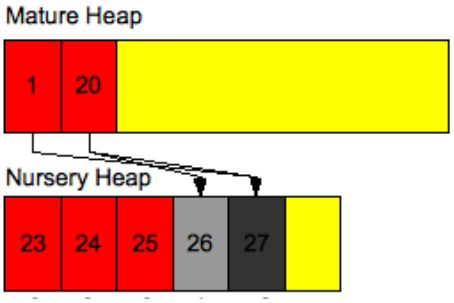

Fig. 5. Mature objects having references in nursery heap.

4 Benchmarks

Unfortunately it was not possible to implement all the ideas given in this paper in the SOM++ VM. After implementing a moving gc which uses the object table and maintained two heaps which always crashed. Even after about 20 hours of debugging and finding some problems in conjunction where the garbage collector ran. So sometimes wrong entries of the object table were deleted because the object didn't received their index yet other times objects which weren't referenced yet were removed from the heap. Sometimes even pointers to wrong objects appeared this was never fixed.

To have a running solution we implemented a non moving garbage collector which has two lists one nursery list and one mature list. This kind of garbage collector worked for some benchmarks but was about 10 times slower then before.

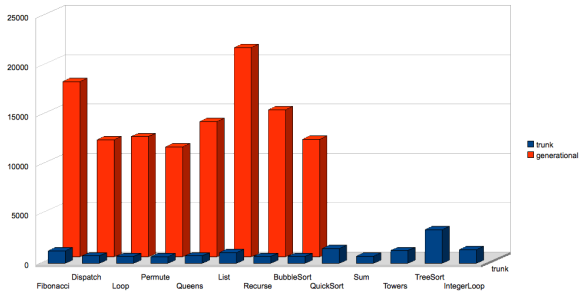

Fig. 6. Comparison between non-moving generational garbage collector and mark-and-sweep in trunk.

5 Related Work

Another group in the lecture implemented a Reference Counting Garbage Collector [8]. In comparison to a mark and sweep approach the Reference Counting Garbage Collector can't break automatically circular structures which are not pre defined. The big advantage of a the counting gc is that it does not have to stop the world when it is running. The group also implemented immediate signed integers which made the benchmarks look pretty well. The performance of the the collector itself compared to the trunk mark-and-sweep approach is a little bit better. Especially at allocating new space when this is necessary a lot.

The current trend in computation to parallel processors, large memory and real time applications poses new challenges to the garbage collectors even after 50 years it is still necessary to write new ones. The Java 6 VM brings the new G1 (garbage first) [12] collector as an experimental new collector. It can run in parallelism and mets soft real-time goals. Time will show how good the new collector will perform in the real world.

6 Future Work

The most important thing is to make the moving garbage collector version work. When this is done the remembering sets code can be merged to gain a real performance benefit.

7 Conclusion

Implementing a mark-and-sweep garbage collector is difficult because memory corruptions are hard to find and race conditions appear often. The whole code has to be designed in a transactional manner where parts are allocated in conjunction on the heap.

References

1. William D. Clinger, Lars T. Hansen: **Generational Garbage Collection and the Radioactive Decay Model** ACM SIGPLAN Notices Volume 32, Issue 5 (May 1997) 97 – 108
2. John McCarthy.: **Recursive functions of symbolic expressions and their computation by machine, Part I** Communications of the ACM Volume 3, Issue 4 (April 1960) 184 – 195
3. Jack Belzer, Albert G. Holzman, and Allen Kent: **Virtual memory systems** Encyclopedia of computer science and technology, eds. (1981), 14, CRC Press, pp. 32
4. David R. Barach, David H. Taenzer, and Robert E. Wells: **A technique for finding storage allocation errors in C-language programs** ACM SIGPLAN Notices Volume 17 Issue (May 1982) 16 – 24
5. David L. Heine, Monica S. Lam : **A practical flow-sensitive and context-sensitive C and C++ memory leak detector** ACM SIGPLAN Notices Volume 38 , Issue 5 (May 2003) 168 – 181

6. Donald E. Knuth: **The Art of Computer Programming, volume I: Fundamental Algorithms**, Chapter 2. Addison-Wesley, second edition, (1973)
7. Bruno R. Preiss: **Data Structures and Algorithms with Object-Oriented Design Patterns in Java** Mark-and-Sweep Garbage Collection, John Wiley & Sons, Inc (1998) 424 – 425
8. Martin Czuchra, Daniel Hefenbrock: **Object Table and Reference Counting in SOM++** Lecture Virtual Machines at Hasso-Plattner-Institut (2009)
9. Richard Jones, Rafael Lins: **Garbage Collection – Algorithms for Automatic Dynamic Memory Management** John Wiley & Sons, Inc (1999)
10. Jacques Cohen, Alexandru Nicolau: **Comparison of Compacting Algorithms for Garbage Collection** ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 5 , Issue 4 (October 1983) 532 – 553
11. David Ungar, Frank Jackson: **An Adaptive Tenwing Policy for Generation Scavengers** ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 14 , Issue 1 (January 1992) 1 – 27
12. David Detlefs, Christine Flood, Steve Heller, Tony Printezis – Sun Microsystems, Inc. **Garbage-First Garbage Collection** ISMM04, October 2425, 2004, Vancouver, British Columbia, Canada. ACM 1581139454/04/0010.
13. David Ungar: **Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm.** ACM SIGPLAN Notices, Volume 19, Issue 5 (1984) 157–167.