Cost Effective Garbage Collection for Embedded Java

IT-Vest Master Thesis

Master of IT with Specialisation in Software construction

Aalborg University

June 2013

By Lars B. Sørensen

Study no. 20110454

Counsellor Anders P. Ravn

Handed in on the 6th. June 2013

Number of pages 41

Abstract

This thesis presents an implementation of a Garbage Collector for the Hardware near Virtual Machine. The HVM is a Java Virtual machine targeted at low-resource embedded devices. The GC is written in Java except for a few lines of C code in the run time system. Therefore the GC does not impose any restrictions to the HVM. In the design of the GC we have managed to incorporate the GC into the HVM with no overhead in regards to memory usage and processor time for applications that do not need a GC. For applications that use the GC there will be some overhead but we have made a soft real-time GC with low overhead. The design of the GC is a concurrent mark-sweep garbage collector with snapshot at the beginning and conservative root finding.

Table of Contents

1 Motivation and background

The purpose of this project is to build and document a **G**arbage **C**ollector (GC) for the **H**ardware near **V**irtual **M**achine (HVM) [2]. The HVM is a Java Virtual machine for low-resource embedded devices. It supports Java to C compilation as well as interpretation. The key feature of the HVM is that it translates a Java program into a self-contained unit of ANSI-C that does not depend on any operating system (POSIX or similar) or C-library. In order to keep the executable small, the HVM uses intelligent class linking so that only classes that might be used (a conservative estimate is made) are translated into C. Due to this the HVM can support the entire Java Development Kit (JDK) and still produce small executables since only what is needed is paid for. Lately support for the SCJ profile has been added to the HVM [20].

The main principles of the HVM are summarized in:

- Incremental
- Portable

In the design of the GC for the HVM the goal has been to support these principles.

1.1 Incremental

Due to its intelligent class linking the HVM is able to run on embedded systems with only a few kB of RAM and 32 kB of ROM and yet support the entire JDK [2]. This is what we understand with it being incremental: it is possible for the HVM to run on very small systems because it do not include more than the application needs and still the HVM can utilize the entire JDK when running on larger systems. In the design of the GC our goal is to support this principle. Not all applications need garbage collection. Just as the HVM do not load a lot of classes that are not needed an application that do not need garbage collection should not pay for it. As discussed later it is possible to avoid overhead for applications that do not need garbage collection. So the HVM without GC can still support safety critical systems using SCJ.

In our design of the GC, making no overhead comes at the cost of not being able to guarantee hard real-time predictability when using the GC, this will be discussed in chapter 3.2 and 5.7.

1.2 Portability

The other key point of the HVM is that it is portable. A large portion of the HVM as well as of the GC is therefore written in Java. This is translated with the SUN Oracle Java compiler to byte codes and they are then translated to C by the HVM. In order to interface with the target, some additional C code is necessary but only very few lines of C code is needed for the HVM (and GC) to run on a target. The C code can finally be translated to machine code if a C compiler exists for the target platform. It is very unlikely not to have a C compiler for a target platform.

1.3 Delimitation

The application scope for the HVM is embedded low-resource systems. We will therefore focus on single chip systems with a single RAM memory space. The efficiency advantages of parallelism on multi chip systems and the challenges with consistency in those systems will not be discussed. The questions of where referenced objects are placed in the memory on larger systems (is it in cash, RAM or paged out?) is very important for the

look up time of objects and thus for the efficiency of the GC in these systems but since we do not aim at system with cash or virtual memory it will not be discussed.

In this version of the GC, problems with fragmentation will not be solved. Due to this and other constraints we will not be able to give general hard real-time guaranties.

2 Garbage collection methods and concepts.

The principles of the HVM have been guidelines in the design of the GC. In order to understand the reasons for our choices, fundamental GC techniques with their pro and cons will first be described. This section builds on material from seminal papers by E.W. Dijkstra [11] and P.R. Wilson [1] and the book by R. Jones [12].

2.1 The purpose of a Garbage Collector.

Before venturing into the world of garbage collection techniques and theories it is useful to step back and look at the purpose of having a GC in the first place. In C, memory can be allocated dynamically with `void *malloc(size_t size)` where `size` is the size of the object to be created and the function returns a pointer to allocated space. When the object is no longer used the memory can be freed with `void free(void *p)` where `p` is a pointer to the object. The service that allocates memory and frees it again is called the "Allocator".

Without a GC the programmer is responsible for keep track of the allocated memory. If memory for objects not used any more is not freed and memory continuously is allocated for new objects, the system will eventually run out of memory; this is called a "memory leak". Another problem is "dangling pointers". This is when an object is freed while still in use. Next time it is accessed it might have been overwritten by the allocator and unpredictable behavior is the result.

The purpose of a GC is to avoid these problems by automatically ensuring that unused memory is freed and that used memory is never freed.

Another problem with the C allocator is fragmentation of the memory space. When allocating and freeing objects of different sizes, the memory may over time become fragmented meaning that there is plenty of free spaces but it consist of small chunks between allocated spaces so that requests for space to a large object cannot be satisfied. This is called "external fragmentation". To prevent this, C programmers can limit themselves to only allocate blocks of the size of the biggest object used. This will prevent external fragmentation but inside the blocks a lot of space may be wasted (called "internal fragmentation"). Often it is the task of the GC to prevent fragmentation by compacting the allocated memory. There are many algorithms that can do this. In its simplest form an atomic operation scans the memory and move all live objects together. The result is that the memory consists of a compact block of objects and a block of free space. In the process all pointers must be updated so that when an object has been moved all pointers to that object point to the new location. This is however difficult to do in C since integers can be cast to pointers so it is hard to tell which values are pointers and which are data values.

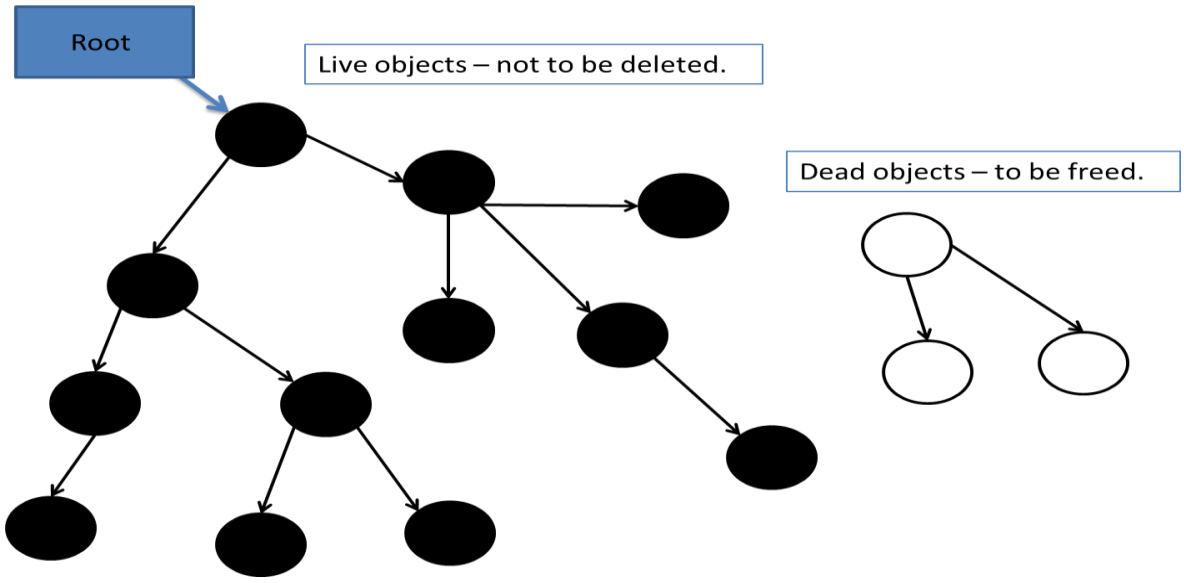
2.2 How Garbage Collectors work.

There are many different types of GCs but they all have in common that they must be able to discriminate between objects that are no longer used by the program (dead objects) and objects that are still in use (alive). Since it is not possible to predict all programming behavior a conservative estimate is made by all GCs. This means that no object may ever be reclaimed that might be access later while it is ok to keep some objects that are actually dead. Conceptually the work of a GC can be divided into two steps:

1. Distinguishing between dead and live object.
2. Reclaiming the memory of the dead objects so it can be reused.

2.2.1 How garbage collectors distinguish between dead and live objects.

References to objects occur in stacks and static variables and in the objects themselves. The former kinds of references are called the “Roots”. The area of the memory where the objects reside is called the “Heap”. The objects in the heap can also hold references to each other. Together roots and objects form a directed graph. By computing a transitive closure of a “points to” relation between objects and roots it can be determined which objects are reachable from the roots and which are not. This is done by following the references from the roots which can be done with a depth first or breadth first traversal. All objects reachable might be accessed later by the program and are considered to be alive while all unreachable objects are dead because they can never again be accessed by the program. See figure 1. for an example of an object graph where the black objects are reachable from the root and the white are unreachable and therefore dead.

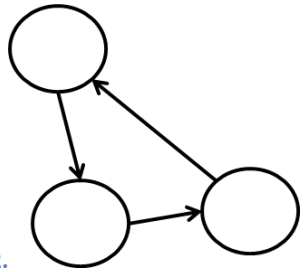


Marking objects 1

2.3 Types of Garbage Collectors.

2.3.1 Reference counting.

A simple type of GC is reference counting. The idea is that every object in the heap counts how many references there are to it. The run time system must every time a reference is changed increment or decrement the reference counter in the target object. When an object dies (the counter reaches zero) all of it children must have their reference counter decremented and if this causes some of them to die their children must be decremented too etc. All the objects that die during this traversal can be reclaimed by the allocator. This work can be done in one step or some of it can be done later by putting dead children into a list for later scanning. The big problem with reference counting is that it cannot free cyclic data structures. In a cyclic structure all objects hold a reference to the next and so the reference counting GC will think they are alive even if there are no references to the data structure from other objects or roots. See figure 2. for an example of a cyclic data structure that reference counting will fail to reclaim.



A cyclic reference 2.

2.3.2 Mark-Sweep

A mark-sweep GC works in distinct steps:

- An idle phase (not marking or sweeping)
- A marking phase
- A sweeping phase

Initially the system is in the idle phase since all memory available for the heap is free and can be allocated. The marking phase starts when the GC detects that the system is running low on memory and hopefully can free some that is not used any more. Starting from the roots the object graph is traversed and all objects meet during the traversal are marked. When done traversing the graph, all live objects are now marked and all dead (unreachable) objects are not marked.

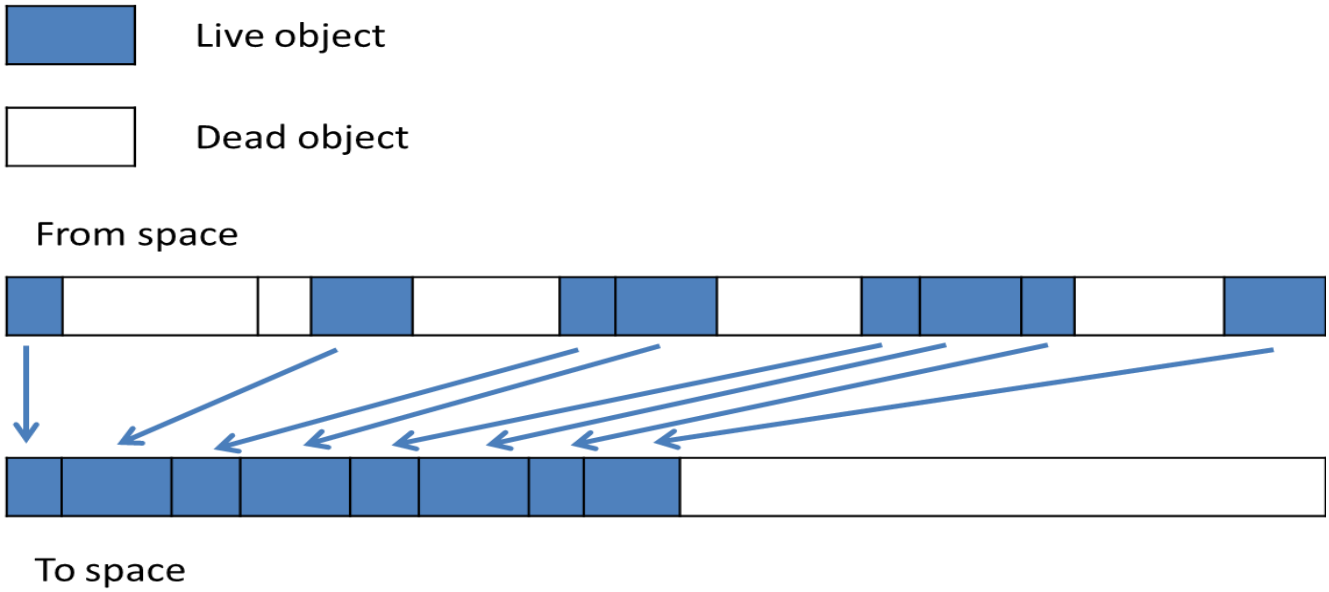
The next step is the sweeping phase: all objects in the memory are scanned and all those not marked can be freed and reused by the allocator.

2.3.3 Mark-Compact

The mark-sweep algorithm does not help against fragmentation of the memory. To deal with this problem a compacting phase can be added. In the compacting phase the live objects are moved in order to squeeze out the space from dead objects between them, giving the allocator a large contiguous area to allocate from.

2.3.4 Copying collector

Another way to collect garbage and fight fragmentation is to use a copying collector. Actually it does not collect garbage rather it collects live objects from one part of the memory and move them to another part of the memory so that all that's left behind is garbage. A simple version is "semi space". The idea is that the memory is divided into two parts a "From space" and a "To space". Instead of marking the live objects during the marking phase the live objects are moved from the "Form space" to the "To space". The space of the dead objects is squeezed out leaving a large contiguous area of memory for allocation. See figure 3. for an example of this.



Moving collector 3

All objects are now compacted in the "To" space and new objects are allocated in the free space here. When the "To space" is about to run out of memory the old "From space" becomes the new "To space" and the old "To space" becomes the new "From space" and a new collection is done. During the process of moving the objects, all references both in roots and in objects must be updated in order to point to the new location in the "To space".

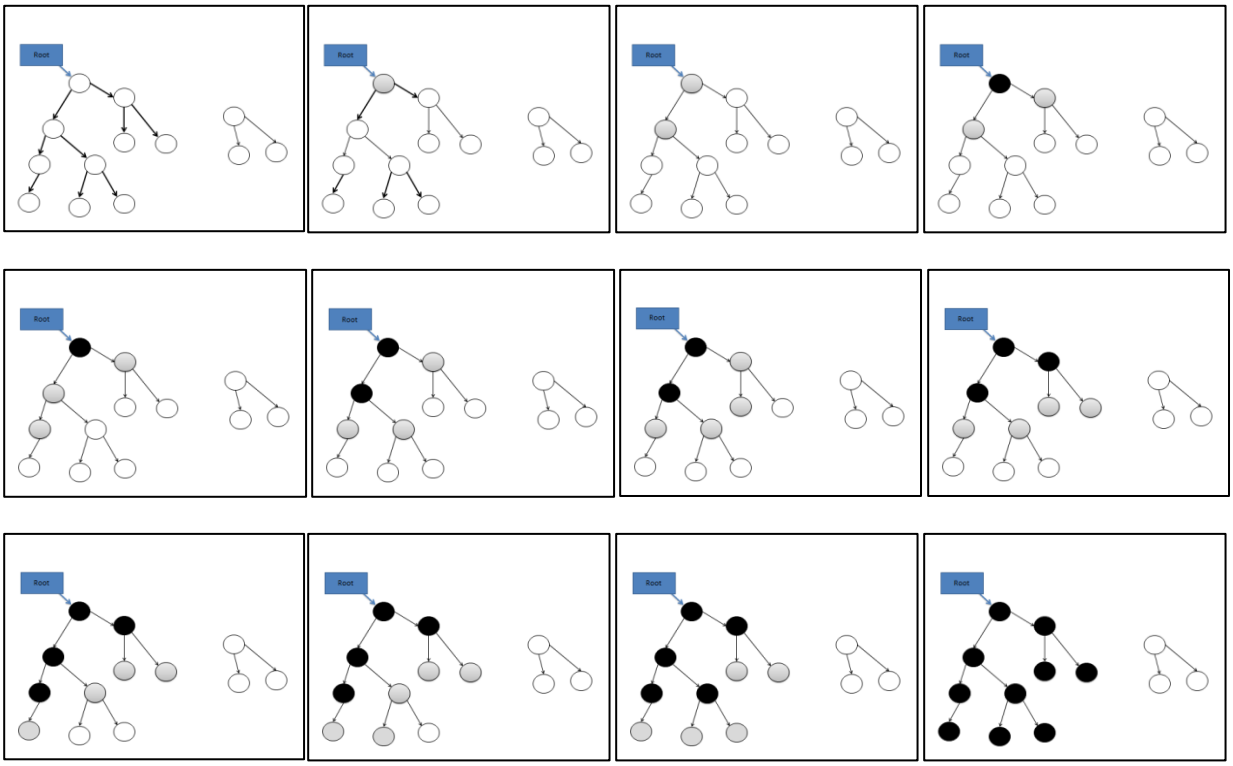
2.4 Concurrency and incrementality.

2.4.1 Mark-sweep, mark-compact and copying collectors.

Mark-sweep, mark-compact and copying collectors all work in distinct phases with at least an idle phase and a collection/reclaiming phase. A simple way to implement the algorithms is "Stop the world". During the collection/reclaiming phase only the garbage collector runs and all other threads are stopped. The problem is that the pause imposed by the GC can be long since the entire heap has to be scanned and the space of all dead objects freed. This is unacceptable for hard real-time systems and very annoying for soft real-time systems. To avoid this long pause it must be possible to preempt the GC and let other threads run during the

collection phase. This requires some synchronization since the application threads called “Mutators” can change the object graph will the GC thread called the “Collector” is marking or moving objects. These synchronization mechanisms are called “Barriers”. In the case of mark-sweep only the mutators change the object graph so there is no danger when the mutators read references. But when the mutators change the graph it is necessary to inform the collector. This is done with “Write barriers”. In the case of copying collectors both mutators and the collector change the graph and therefore “Read barriers” are needed to ensure that the right reference is read when an object has been moved.

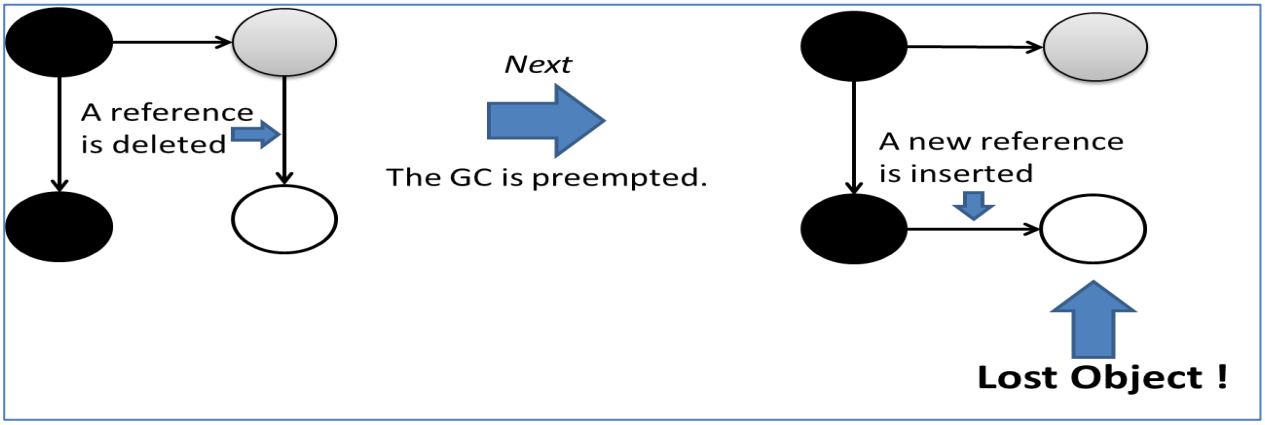
Regardless of which algorithm is used the “Tri color abstraction” [11] is a useful way of viewing the objects graph during the collection. The general idea is that objects are black, grey or white. Initially all objects are white and during the scan of the object graph the objects are colored black when all their children are found. While the collector scans the object graph it uses grey for objects that are reached and therefore alive but where all children are not yet scanned. When all children of a grey object have been scanned the grey object is shaded black. It can be visualized as a grey wave front moving through the graph from the roots. It doesn’t matter whether depth first or breadth first is used as long as the traversal will keep the nodes in front of the wave white and after black. The example in figure 4. uses breadth first:



Traversing the objects graph 4

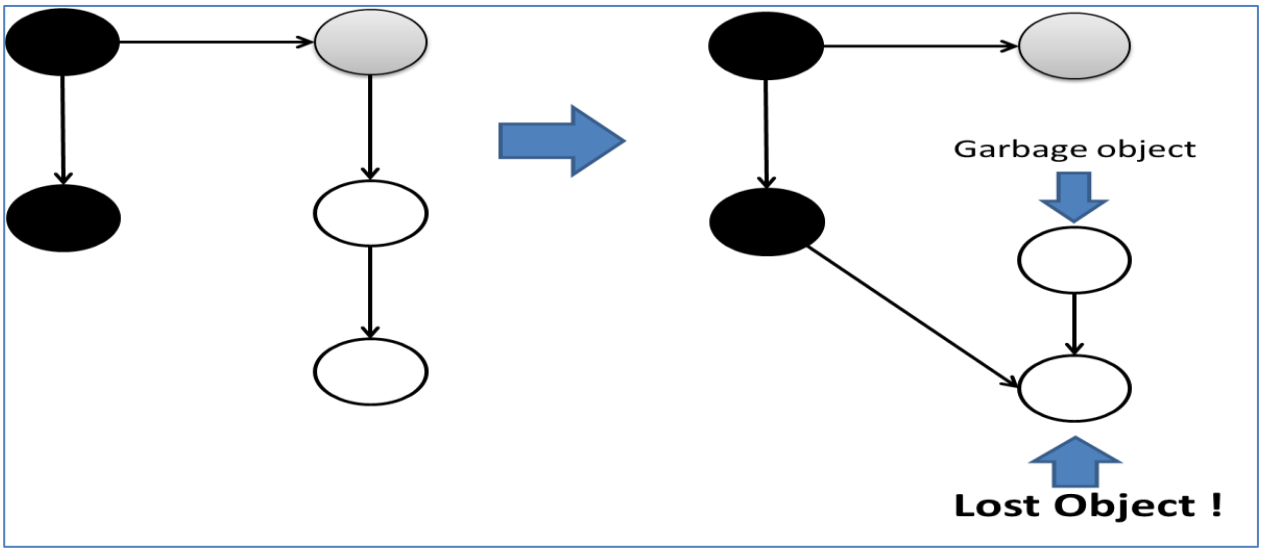
Eventually all reachable objects will be colored black and the dead ones will remain white.

As long as the mutators changes a reference only involving black and grey nodes no objects will be lost since they are all known by the collector. On the other side of the front the same applies: if a reference involving white or grey nodes is changed only white objects that have lost all their references will not be reached which is perfectly ok since they are garbage and should be reclaimed. The problem comes when a reference between a black and a white node is changed. This may cause the lost object problem: First a reference between a grey and a white node is deleted. This is not a problem: if there are no more references to the white node it should be reclaimed.



Lost object 5

But if the collector gets preempted by a mutator that writes a reference to the white node into a black node then the white node is still alive but will be lost (see above in figure 5.). The reason is that the black node will never be scanned again. The problem can be transitive if a chain to a white node is broken:



Lost object through chain 6

This defines an invariant: there must never be a reference from a black to a white node.

We can however break the invariant as long as the white object is saved somehow which is called “Grey protection”: When a reference from a white or grey node to a white node is deleted the child is saved in a list for later scan. That makes the child grey and therefore it and all of its children will be scanned later. Now the object is not lost if a black node has a reference to it.

Or we can ensure that there actually never exists a reference from a black node to a white which is called “Black protection”: When a reference to a white node is written into a black node the black node is reverted to grey (so that the white child will be found later when the node is rescanned).

2.4.2 Reference counting.

Reference counting is in its nature a more incremental algorithm since every time a reference is changed some work must be done to update the counters. Also every time an object dies it can be reclaimed right away. This allows a great deal of concurrency since the collector is constantly working in-between the work of the mutators.

3 Proposed GC for the HVM

As stated in chapter one the HVM has as one of its two principles to be incremental meaning that any given application should only pay for what it needs. This implies that there should be no overhead for applications not using a GC.

The other principle is that it should be portable and therefore most of the GC should be written as a normal Java program.

Another important aspect is that a lot of the applications that run on embedded systems are at least soft real time so we want the pauses imposed by the GC to be short and that it shall be possible for high priority tasks to preempt the GC. The GC for the HVM must therefore be concurrent.

Based on this we will in the following describe and discus our proposed design for a GC for the HVM.

3.1 Root finding

An important consideration is how to find the roots. In some systems it is known which cells on the stacks contains references and in others systems it is not. Obviously the work is harder if we can’t tell which cells are references and which are just data. Since Java 1.5 we can’t be sure which cells contain references so unless we compile with java 1.4 we can’t look at the stack and identify references right away. The possibility of compiling with Java 1.4 and get knowledge about the references will be discussed in chapter 7.1.1.

One solution is to make the run time system responsible for identifying all references on the stacks. This will add some overhead since all operations involving references on the stacks must be dealt with. In its simplest form this just means that the stack cells containing references are marked somehow, in more advanced forms root lists are constantly updated. When a garbage collection starts the root set in the last form is available right

away where in the first form the stacks has to be scanned for the cells containing references (just as for systems where the position of references are known).

Another approach for systems where the position of references is not known is “Conservative root finding”. Here the run time system does not do anything when the GC is not running. When the garbage collection starts all cells in the stacks must be checked to see if they contain a reference. The result is a conservative estimate of the roots; everything that looks like a root is considered to be a root. In order to check if a value could be a reference the following check is done:

- Is the value in the address range of the heap?
- Is it pointing to something that could be an object?
- Is there an allocated object starting at that place?

If all is true then the value is pointing to an object and could be a reference. However the value might not be a reference, but just a simple value that by chance points to an object. Since we cannot tell if the object is alive it has to be kept since this is a conservative approach (we may keep dead objects but never free live ones). So the consequence of conservative root finding is that we might keep objects alive that should have been reclaimed (dead objects that are not reclaimed are called “Floating garbage”). Since stacks grow and shrink the value might be gone by the next collection and then the dead object will be reclaimed. There is however a small risk for a memory leak: false references on the stacks can lead to more and more floating memory kept alive but this requires that the false references are never pop from the stacks since the garbage will then be collected.

Despite these disadvantages we have decided on conservative root finding. The reason is that applications that do not need garbage collection should not pay the price of having extra instructions added to all operations changing references on the stack.

3.2 Copying collector, mark-sweep or reference counting.

During the traversal of the object graph, a copying collector moves the objects it meets to the “To space”. Likewise, a compacting collector after the traversal moves all live objects to form a block of allocated space. If using conservative root finding a stack value that is not a reference but pointing to a recently dead object will cause the object to be moved and thus kept (creating floating garbage). This is acceptable; we cannot avoid creating some floating garbage when using conservative root finding. When the object has been moved the reference is updated to point to the new location. But the value was not a reference; it was just some data that had the same value as the address of an object and if we change the value of the data we corrupt the program.

The choice of conservative root finding therefore prevents us from directly updating references on the stacks. This problem can be solved using tables for looking up objects. This will add overhead to all operations involving references on the stacks since they must look up the address of heap objects in a table instead of accessing them directly. This is not acceptable with the principles of the HVM since this overhead is put on all applications including those that do not need a GC.

Even if it is possible to unambiguously find the roots on the stacks without adding overhead on applications that do not need garbage collection, moving collectors pose other challenges in relation to not putting overhead on applications that do not need GC. One problem is that the run time system and the allocator must be tightly connected with the GC and synchronize tightly with operations involving moving objects and updating references. It might be possible to implement a version of the GC with a moving collector that do not put overhead on applications that do not need GC, but that is beyond the scope of this project since it will require major changes in the HVM run time system and we do not currently have the resources. The possibility of implementing such a version will be discussed in chapter 7.

This leaves us with mark-sweep or reference counting. The problem with mark-sweep and reference counting is that they don't prevent fragmentation. On the positive side they require less memory than copying collectors because they do not require an extra equally sized memory spaces to copy to. This is an advantage in respect to the principles of the HVM: applications that do not need garbage collection (and thus don't have to worry about fragmentation) do not have to pay a price in memory for fighting fragmentation.

3.3 Concurrent reference counting

As mentioned in the previous chapter reference counting is incremental in its nature but due to the fact that it cannot reclaim cyclic structures and that there always is an overhead when updating the reference counters regardless of whether or not a GC is needed this option will not be considered.

3.4 Concurrent mark-sweep

A large number of concurrent mark-sweep algorithms exists [1]. They differ in whether they use grey protection or black protection and how they implement the protection. In general the decision is a tradeoff between how fast the object graph can be scanned and how much floating garbage is made.

Of the different versions of mark-sweep we have chosen an algorithm called "Snapshot at the beginning" [1]. The idea is that all objects alive when the collection starts are considered to be alive when the collection ends. This is enforced with grey protection and a "New barrier" that marks all newly allocated objects black. All objects that dies during a collection becomes floating garbage due to this. The advantage of snapshot at the beginning is that it never rescans any part of the graph. All objects have their references scanned exactly once. The worst case execution time is therefore bound by the time it takes to scan all references (vertices) in the objects (nodes) that were alive (reachable from the roots) when the collection started. The number of changes made to the graph during the traversal has no influence on the scanning time (except the time it takes to execute the write barrier and new barrier).

3.5 Stack scanning

The stacks play a special role in the scanning of the objects graph. In the HVM write barriers on the stacks are expensive and we would therefore like to avoid them. One way to avoid them is by looking at the stacks as objects. If all references on a stack has been found we can think of it as a black object. In our implementation we do not worry about black objects since we are using grey protection and therefore have a barrier that only works on grey and white objects. But this implies that all stacks must be scanned atomically in the beginning of

the collection since grey or white stacks would require a write barrier on them. One way to do that is to give the GC thread max priority while it scans the stacks. This gives a potentially long pause in the beginning of the collection which would be unacceptable in a hard real-time system but since we are only aiming at soft real-time in this version it is acceptable.

3.5.1 Scanning of static references

The roots stored in references in static fields are also treated as being one object and since we don't have a write barrier on them they must also be scanned atomically together with the stacks.

3.6 Scheduling of the GC

The final important decision is when should the GC work? One algorithm is work based scheduling. The idea is that the threads that need to allocate memory must pay for garbage collection so that every time memory is allocated a proportional portion is collected/freed. The advantage of this algorithm is that it allows WCET. The disadvantage is that the threads that need to allocate memory might be high priority threads and the threads that don't need to allocate memory might be low priority and yet the high priority thread has to pay for garbage collection. The other disadvantage is every time some memory is allocated an overhead is put to it even if the system never uses up all the available memory and thus don't need garbage collection.

We have therefore decided on time based scheduling. The idea is that the system switches between periods where the GC is idle and periods where it is running. When the GC is not running the only overhead put to the system is a check on whether or not the GC is running. It must be checked so that if the GC is running then a write barrier can be enforced if a reference in an object is changed; otherwise nothing has to be done in relation to the write barrier. In relation to the new barrier this could also only be called during a collection.

We have implemented the GC to run as normal Java thread that can be started when memory is running low (some headroom for memory allocated during the collection and the GC itself is necessary). The GC can be started by any thread that finds the memory low or it can be started by a GC controller thread that periodically checks the amount of free memory. During the collection the controller can increase the priority of the GC thread if free memory becomes critically. The reason for making the controller a separate thread is that in the HVM a thread cannot change its own priority. Therefore a separate thread is needed to raise the priority of the GC thread when memory becomes low.

3.7 Allocation and deallocation

In this version of the HVM with GC fragmentation of the memory will not be addressed. For allocation and release a simple allocator that works like `malloc` and `free` has been implemented in the HVM. When a collection has been done the GC will produce a list of free object and pass it on to the allocator that can then free the objects. To keep track on the allocated part of the memory the GC uses a bitmap that will be described in detail in the next chapter.

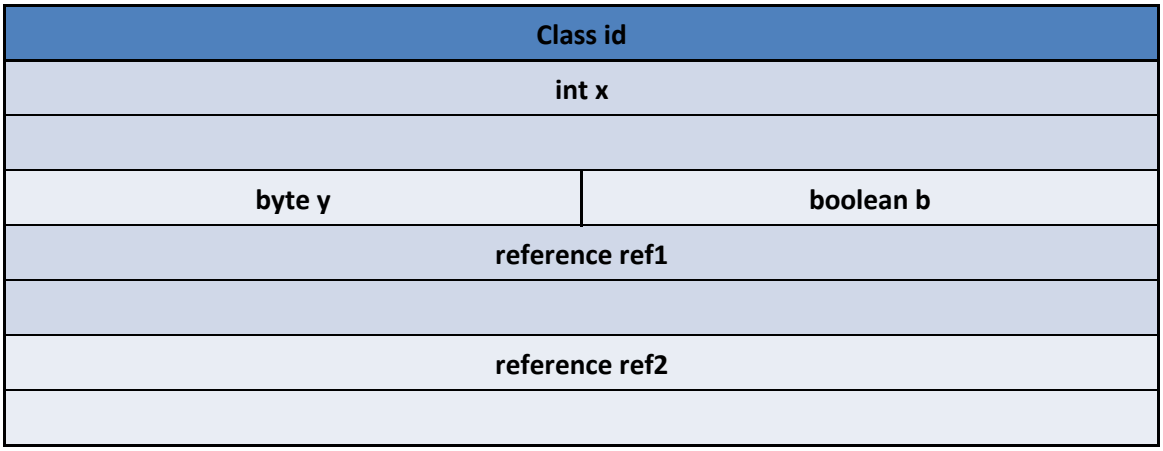
3.8 Summarizing the GC design

Based on the principals for the HVM and weighting the pro and cons of the various solutions we propose a "*concurrent mark-sweep garbage collector with snapshot at the beginning and conservative root finding"* for a GC for the HVM.

4 Fundamental HVM data structures

4.1 Objects

As described in the previous chapters, garbage collection is about finding and distinguishing between dead and live objects so that dead objects can be reclaimed. It is therefore useful to take a look at objects in the HVM. An object in the HVM is a consecutive area of memory. It starts with a 16 bit class ID and then follows the fields of the class. The fields are packed in the object to minimize RAM requirements. But Boolean fields occupy one byte. An integer is 32 bits and so are references (the HVM can only address up to 2 GB). In figure 7 an example of an object can be seen where one row corresponds to 16 bits so this object is 128 bits or 16 bytes.



Object layout for the HVM 7

From a GC point of view there are two kinds of information relating to objects that it needs to know: where are objects located in memory? and where are references located in objects? This will be described in the following.

4.2 The heap and the bitmap

To keep track of the location of objects, the GC use a bitmap that maps all possible addresses of objects in the heap to positions in the bitmap. The heap is divided into blocks of 8 bytes each (objects are allocated at 8 byte boundaries). The reason for the small size is that we want to prevent internal fragmentation and in the HVM a Java object with only one integer field is 6 bytes. Larger objects will therefore occupy several blocks. The allocator place an object in consecutive blocks. In the bitmap the GC marks each block with one of 4 different values. It follows that two bits per block are needed in the bitmap. Three of the values correspond to the tricolor abstraction; black, white and grey. The last we call *hatched.* The hatch mark signifies blocks that are part of an object (but not the first block) or blocks that are not allocated. In the idle phase new objects are

marked white by the new barrier. When in the collection phase, new objects are marked black by the new barrier. During traversal the grey and black markings are used according to the mark-sweep algorithm. When done, the bit map will only contain black objects (that are possibly alive), white objects that are dead and hatched blocks that are part of an object or unallocated. The GC can now scan the bitmap for white objects and add them to a free list that can be passed to the allocator. The allocator can now reclaim the dead white objects.

An alternative approach to a bitmap is marking objects directly in the memory by using some bits from each object (called bit stealing). Since we are using conservative root finding we can however not be sure it is an objects we are writing into. This can be solved by asking the allocator if an object is allocated at that place but that is not implemented in the current HVM allocator. An advantage of using a bitmap for markings is that it puts fewer requirements on the run time system and the allocator. This gives a more detached system with a narrow interface between the allocator and run time system on one side and the GC on the other. Also the bits used in the objects for marking will be wasted for applications that do not need the GC.

4.3 Stacks and static fields

In the HVM all stacks are located in the heap. They consist of two parts: a Java part and a C part that grow towards each other – the Java stack pointer starts at the beginning of the stack area and grows toward higher addresses while the C stack pointer starts at the end of the stack area and grows towards lower addresses. The HVM supports translation of selected methods into C (whiteout interpretation) such methods place their data on the C stack, while interpreted methods use the Java stack.

When the GC wants to get the contents from the stacks currently active, it will call a method in the HVM scheduler that returns an iterator over all live stack cells. This iterator will return all stack cells, also those that do not contain references. Scanning of the stacks cells is the only place where conservatism applies.

The static fields are placed in one continuous area of the memory even though they belong to different classes. Just as with the stacks the GC calls a method in the HVM to obtain an iterator over all static references. There is no conservatism in the scanning of the static references: The HVM has been extended to generate a list of offsets into the static field memory indicating where the references originating from static class fields are placed.

4.4 Threads and processes

Currently only round robin scheduling is implemented in the HVM. A priority based scheduler is however necessary for optimizing the GC work. When an atomic operation is needed the only way to ensure the atomicity is to disable the scheduler. This is not part of the GC and this work.

5 Detailed description of the GC for HVM

5.1 Initialization:

If an application needs garbage collection it must call `GarbageCollector.start()` in main.

If the GC is initialized the run time system starts a controller GC thread at some high priority level. The controller thread must run periodic to check the amount of used memory.
Interface to the HVM: `int getMemoryLeftInProcent()`
If memory gets below a certain level the controller starts a collection - otherwise it does nothing.

The collector runs in another thread which is initially stopped and starts out with low priority. The controller thread can increase the priority of the collector thread if memory levels become critical.

The collector task is released with a semaphore and goes back to pending when the collection is done.

5.2 Barriers: interface to the GC

Whenever a reference is changed or a new object is allocated the HVM run time system calls the GC write barrier or new barrier. These barriers are implemented as static Java methods. The HVM run time system thus makes a callback from C space into Java space during interpretation. The barrier methods are always compiled (and not interpreted) to increase efficiency.

If the GC is not used, the code in the HVM run time system making the callback is not compiled into the executable. Therefore this overhead in the HVM run time system is not added to applications that do not use garbage collection.

5.2.1 Write barrier

Interface GC: `void writeBarrier(int source, int oldRef)`

The write barrier will only do something if a collection is running. This is done by checking a global flag set by the GC controller thread.

When a reference is overwritten in an object the write barrier will be invoked if the object (the `source` node) is white or grey and if the reference overwritten was to a white child (`oldRef`). If true, the child (`oldRef`) is added to the root list unless it was `null`. This is done in order to grey protect the white child.

5.2.2 New object barrier

Interface GC: `void newBarrier(int ref)`

Every time a new object is created the GC will marked it black while running a collection. This is because “snapshot at the beginning” requires that all objects created during a collection are shaded black. Excepted from this are objects created by the GC itself; they are marked white since they are not needed after the collection (the Heap and BitMap objects are created statically in order not to be reclaimed).

When not running a collection new objects are shaded white so that the bitmap always reflects the current state of the heap.

5.3 Start a collection

First the roots are scanned using conservative root finding and saved in a list called `liveSet`. This is done in an atomic process. The `liveSet` list is an instance of the class `LiveSet` that has a special feature: it marks the references that are added to it grey in the bitmap. This is done because the object the root points to is now known but not scanned yet. Therefore the possible children of the object are not known yet.

Atomic Start:

Get all references in static variables:
Interface HVM: `ReferenceIterator getStaticRef();`
save the references in `liveSet`.

Get all possible references from stacks (Except the GC stack):
Interface HVM: `ReferenceIterator getRefFromStacks();`
and add them to in `liveSet`.

Scan all reference `ref` from `r`:

1. Is `ref` within range of the heap?

   Interface bitmap: `boolean isWithinRangeOfHeap(int ref)`

2. Is `ref` pointing to something that could be an object?
   Interface HVM: `boolean isRefAnObj(int ref)`

3. Is `ref` pointing to an allocated block?
   Interface bitmap: `boolean isRefAllocated(int ref)`
   (Point 3 is not implemented in the the HVM allocator but if an allocator is used that does supports this check it should be used to minimize the conservatism).

If all are true add `ref` to the `liveSet` since it might be a reference.

Atomic End.

Below is the source code for the function `setRoots()` that finds the roots. Since we are using conservative root scanning on the stacks we do some checks in order to rule out values that can't be references. In the case of the static roots all we need to check is if the reference is a null pointer (there is no reason to add a null pointer to the roots).

```
private void setRoots() {
    liveSet.clear();
    staticRoots = heap.getStaticRef();
    while (staticRoots.hasNext()) {
        int nextRoot = staticRoots.next();
        if (nextRoot != 0) {
            monitor.addStaticRoot(nextRoot);
            liveSet.push(nextRoot);
        }
    }

    ReferenceIterator stackRoots = heap.getRefFromStacks();
    while (stackRoots.hasNext()) {
        int possibleRef = stackRoots.next();
        if (possibleRef != 0) {
            if (bitMap.isWithinRangeOfHeap(possibleRef)) {
                if (heap.isRefAnObj(possibleRef)) {
                    if (heap.isRefAllocated(possibleRef)) {
                        liveSet.push(possibleRef);
                        monitor.addStackRoot(possibleRef);
                    }
                }
            }
        }
    }
}
```

Source code for finding roots 8.

5.4 Running a collection

When all possible roots are found, the object graph is traversed and all live objects in the bitmap are marked black.

Algorithm for traversing graph:

While `liveSet` is not empty:

> Take out a `ref` from `liveSet`
>
> Get all children from `ref` and add them to an iterator:
>
> `ReferenceIterator children = heap.getRefFromObj(parent);`
>
> while `children` is not empty
>
> > Get next element `child` from iterator
> >
> > If `child` is not null and `child` is not black and child is within range of the heap:
> >
> > > add `child` to `liveSet`
>
> Mark `parent` black (in bit map)

When `liveSet` is empty the collector thread will make a free list and go back, waiting for the next collection (pending on the release semaphore).

When traversing the graph we use the `getRefFromObj` method to obtain the references in the object. Since we are using conservative root scanning we can't be sure that it is an object we are looking at and therefore we check if the references in the object points to something within the heap. We also check if the references are null pointers or if they point at already marked objects. Below the source code for object traversal can be seen:

```
private void traverse(LiveSet nodes) {
    ReferenceIterator children;
    int parent;

    while (!nodes.isEmpty()) {
        parent = nodes.pop();
        children = heap.getRefFromObj(parent);
        while (children.hasNext()) {
            int child = children.next();
            if (child != 0) {
                if (bitMap.isWithinRangeOfHeap(child)) {
                    monitor.visitChild(parent, child);
                    if (!bitMap.isRefBlack(child)) {
                        nodes.push(child);
                    }
                }
            }
        }
        bitMap.shadeRefBlack(parent);
    }
}
```

Source code for the traverse method 9.

During a traversal the references stored in objects must be found. For each class the references are located at specific offsets. The HVM tool chain has been extended to generate a list of offsets for each class. The lists indicate where the references for a particular instance of a class can be found. Using the class ID the run time system can find these offsets by looking in an offset table and put them into a list:

Interface heap: `public ReferenceIterator getRefFromObj(int ref);`

Source code for scanning an object:

```
@Override
public ReferenceIterator getRefFromObj(int ref) {
    oi.setAddress(ref);
    short classId = oi.classId;
    if (classId >= 0) {
        if (classId < ClassInfo.getNumberOfClasses()) {
            ClassInfo cInfo = ClassInfo.getClassInfo(classId);
            ReferenceIterator referenceOffsets = cInfo.getReferenceOffsets(ref);
            if (referenceOffsets != null) {
                objectReferenceIterator.initObjectReferenceIterator(referenceOffsets, oi);
                return objectReferenceIterator;
            }
        }
    }
    return emptyIterator;
}
```

Source code for scanning an object 10.

If the object in question is an array then a special action is required. All arrays in the HVM start with a class ID (a short) and a count for the size of the array (another short). By looking at the ID it is possible to tell if it is an array of references. If that is the case all values in the array (which are references) can be put into the iterator.

5.5 Collect garbage (making a free list):

When the collection is done the bit map is scanned for free objects that are white. All the black objects are alive and can be ignored (though they are shaded white in order to prepare for the next collection).

The white objects are added to a free list and they are shaded hatched in the bitmap since they will be reclaimed by the allocator.

Now the allocator can do a full scan of the iterator to free all memory or do it in small chunks (lazy scan).

5.6 Run time system interface

To ensure portability the GC is written in Java. However one of the key points in Java is that the addresses of objects are hidden. To overcome this problem hardware objects [13] are used to get the addresses from the run time system (which is written in C and therefore can access the addresses directly). All requests for addresses go through the interface Heap or methods in the GC that are called by the run time system with addresses as arguments. Currently the HVM only supports 32 bit addressing.

5.7 Cost of the proposed GC

As described in this chapter we have managed to implement a GC for the HVM where applications that do not need garbage collection does not pay for overhead in memory and processor time. The reason is that only five changes have been made to the HVM:

- On object access the HVM calls the write barrier callback function in the GC.
- On objects creation the HVM calls the new barrier callback function in the GC.
- A function is implemented in the HVM for obtaining a reference offset list for class fields.
- A function is implemented in the HVM for obtaining a reference offset list for static class data.
- A function is implemented in the HVM for checking if a value points to something that could be an object.

and none of this are compiled into the application if `GarbageCollector.start()` is not called.

In terms of memory usage the HVM allocator and object model has not been changed so extra memory is not used for applications that do not use the GC.

For applications that do use the GC our solution uses less memory than a moving collector does and there is less overhead in the run time system since objects are not moved but only marked.

6 Related work

Many environments exist for embedded Java. Some of the interesting candidates are JamVM, CACAO, GCJ, FijiVM, JamaicaVM and KESO. The first five are targeted at POSIX like operating systems where KESO and the HVM are targeted at barebone environments. For a detailed description of the differences between the HVM and these systems see [2] page 33-39. In this work the focus will be on what the different systems offer in regards to garbage collection.

KESO has two kinds of GC [5] [6]. Both are mark-sweep. One is a stop the world GC while the other is concurrent. The concurrent GC use a snapshot at the beginning write barrier. There is no defragmentation and therefore KESO with GC can't be used in safety critical systems. The overall GC algorithm is the same as for the HVM except that KESO can be preempted between scanning of the stacks since it has a write barrier on the static references and therefore only needs to scan the individual stack atomic. Also KESO keeps track of the references on the stacks by grouping the references in the stacks frames and linking the groups together. How exactly the lists are updated is not clear but it could involve recordings of all changes of the references on the stacks. The cost of unambiguously getting the stack roots is that it hurts performance. This is the conclusion in the paper by F. Henderson [14] page 3, and this is the algorithm KESO builds on [5] page 795. The possibility of implementing similar optimizations in the HVM but without the cost in overhead will be discussed in chapter 7.

In contrast to KESO and the HVM, Jamaica [7] [8] [15] and FijiVM [9] [10] offers hard real-time guaranties as well as means to avoid fragmentation. Both systems are based on a concurrent mark-sweep algorithm that allows root scanning to be done none atomic. In the case of root scanning FijiVM uses a strategy of letting the threads scan their own stack. This is described in more detail in chapter 7. Jamaica on the other hand keeps a copy of all roots in the heap for easy access when a collection is started. The main difference between the two algorithms is when the application is paying for root scanning. In the case of Jamaica there is always an overhead when dealing with references but acquiring the roots when doing a collection is fast and simple. In contrast FijiVM does not have an overhead of always updating a list of roots but must scan all stacks when a collection starts (Though references on the stack must be marked in order to identify them). If the HVM is going to support hard real-time it could use one of these algorithms to give hard real-time guaranties (see chapter 7 for a detailed discussion).

Jamaica and FijiVM both prevents external fragmentation by splitting up the memory in a number of equal sized blocks. Objects larger than the block size and arrays must then be distributed over several blocks. This applies overhead when allocating and accessing large objects and arrays but saves the system from moving objects with the involved overhead of updating pointers and enforcing read barriers as well as the extra memory used to have a *from* and a *to* space. This however comes at the cost of some internal fragmentation in the blocks. Both systems use linked lists of blocks to tie larger objects together. In regards to arrays Jamaica use a tree structure instead of linked lists to speed up access while FijiVM claims even better look up times by using arraylets which is an array of pointers (called a spine) to access the array elements [10] page 147 .

JamVM, GCJ and CACAO all uses nonmoving mark-sweep collectors that do not fight fragmentation [16][17][18][19]. In the case of CACAO the reason for using a nonmoving collector is explicitly that moving large objects (the size of Gbytes) is giving too much overhead [17].

As it can be seen the GC for the HVM together with systems described in this chapter uses many of the same techniques. Since all systems are faced with the same challenges in regards to embedded Java this is not surprising. In the following chapter the possibility of using these techniques to extend the HVM to support hard real-time guaranties and avoiding fragmentation will be discussed. But the discussion will not be limited to the techniques used by these systems, alternative techniques will be compared against them.

7 Discussion

7.1 Future improvements

For safety critical applications it is a requirement for using a GC that it can give hard real-time guaranties as well as fight fragmentation (like Jamaica and Fiji). It would therefore be interesting to take a look at what would be required in order to fulfill these requirements taken into consideration that we still want the HVM to be able to run applications not using the GC without any overhead due to having the option of a GC. If possible the user should choose between three different types of the HVM:

- No GC (the default setting)
- Soft real-time GC: In main call `GarbageCollector.start();`
- Hard real-time: In main call `RealTimeGarbageCollector.start();`

7.1.1 Possibilities of making a GC that supports hard real-time systems

Compared to our GC a hard real-time GC would have to deal with the following problems:

1. Roots must be found non conservatively.
2. Stacks and static fields scanning must be incremental.
3. Fragmentation must be dealt with.

One way to solve the first problem could be to mark the roots in the stacks or group the references in the stacks and keep lists over them as KESO does. Both approaches add overhead to applications using the hard real-time GC. For applications not using the hard real-time GC this overhead can be eliminated by not compiling the code responsible for keeping track on the references. Another approach could be to compile with Java 1.4. Up to and including this version of the SUN Java compiler methods always had their references at the same offsets in the stack frames. Using this approach the roots can be found unambiguously at compile time. This static information can be embedded into the application thus avoiding the overhead to a barrier on all operations involving references on the stacks.

The next point is more demanding. There are several ways to deal with the problem of reducing the pause introduced by root scanning. One solution is to use barriers on the stacks and static references. This will however put overhead to the application because normal stack activity will trigger many barrier calls [3] page 13. Another approach is to only have a barrier on static references and then let the stacks scan themselves as described by W. Puffitsch [3] page 25 and used by FijiVM. Scanning of the individual stacks must still be atomic but since the threads do it themselves it is atomic since no thread can change the stack of another thread in Java. The scanning call can be implemented in the scheduler so that when a collection is required the GC sets a flag and when a thread go into waiting state the scheduler will call the stack scan method. An advantage of this algorithm is that when a thread goes into waiting state its stack will often be shallow and thus fast to scan.

The last point is fighting fragmentation. Again there are many different strategies to choose from but basically the way to avoid fragmentation can either be to use moving collectors or the strategy of Jamaica and FijiVM.

The advantage of moving collectors is that allocation becomes fast but during the collection objects must be moved, read barriers enforced and references updated. Outside the collection phase the nonmoving collectors that fight fragmentation must do more work when allocating and accessing objects but during collection they have less overhead since they only have to enforce a simple write barrier. In regards to memory consumption the moving collectors need two memory areas while the nonmoving collectors with fixed block size will experience some internal fragmentation.

Regardless of which solution is chosen it would require a new data structure and a new allocator to be built for the HVM. It might still be possible to do this without adding overhead to applications not using the hared real-time GC. All parts of the hard real-time GC that are written in Java will not be included due to the intelligent class linker and the code in the HVM that deals with the hard real-time GC could only be compiled when needed.

7.1.2 Interesting fields to explore in the current version of the GC

In the current soft real-time version of the GC there are a couple fields we would like to explore:

- Scheduling of the GC
- Hybrid type – reference counting and mark-sweep.

The scheduling of the GC is very important. If the GC waits to long with starting a collection it will have to run with high priority at the end of the collection in order to finish before the memory runs out and thus it will bother other threads in that phase. If it runs too often it will put unnecessary overhead to the application. Since many embedded systems work in predictable patterns it might be possible to calculate when the right time is to start a collection for at given application. This should be done by the GC itself perhaps using statistics from the last collections.

Another interesting improvement to the current GC could be to add a reference counting GC. As described previously reference counting is both efficient and incremental. It does however lack the ability to reclaim

cyclic structures. So a possible scheme is to have the reference counting GC running all the time and then only start the mark-sweep GC if memory runs low.

7.2 Fragmentation in the HVM with GC compared to systems programed in C

In the previous chapter the HVM was compared to other Java VM's that can be used in embedded systems. Though it is hard to find conclusive data there can be no doubt that a big portion if not the majority of embedded systems are programmed with C. It is therefore interesting to compare the differences between programming an embedded system using the HVM with GC and using standard C with `malloc` and `free`.

Since the C library functions `malloc` and `free` does not solve fragmentation problems the programmer must do something to avoid fragmentation. In C there are a number of ways to do this. One solution is not to use `free`. Instead all needed memory for a number of objects of a particular size is initially allocated in one block. The program must then itself allocate objects of this size in that block and again free it from the block when the object is dead. It is thus the program that is responsible for keeping track of used and unused memory in each block. Another approach is that the programmer can restrict the application to only allocated blocks of the same size. In C the size of an allocated block is set explicit with `malloc` so for all objects a block the size of the biggest object in the program is allocated.

The same ad hoc solutions can be used with Java and the HVM. The first corresponds to allocate all memory in the beginning of the program and then never to use `new` again. The second one requires that all objects are of the same size since Java automatically allocates memory that fits an object when using `new`. Objects smaller than the biggest one must therefore have dummy fields added to get the right size.

Besides adding overhead to the program both solutions suffer from internal fragmentation. In the first case one block may be filled up while there is plenty of space in other blocks. In the last case a piece of memory is wasted in all objects smaller than the biggest ones (equal to the difference in size between the biggest object and the object in question).

For a programmer of the HVM with GC and a C programmer the fragmentation problem is the same, but what the HVM with GC offers is a guarantee against memory leaks introduced by the programmer (though it may still happen due to the conservative root finding, see chapter 3.1) and a guarantee against dangling pointers. None of these guarantees are given by C with `malloc` and `free`.

8 Test

The HVM has an extensive test system that automatically tests the HVM. Some GC tests have been added to the test system as well as tests that check the run time system interface.

The intention of the GS tests is

1. To check that the system can do a collection
2. To check how much memory the GC itself allocates
3. To check that the memory allocated by the GC is freed after the collection
4. To check that live objects are not freed and that dead objects are freed

The intention of tests of the run time system interface to the GC is

1. To check that stack scanning finds the references on the stacks
2. To check that static field scanning finds the references in static fields
3. To check that references in objects are found

One example of a test is given here. The intention of this test is to check the first three points of GC tests. This is a simple test whit just one mutator. In main a mutator thread is made and started. The `run` method of the thread request a garbage collection so all that is reclaimed are objects made during initialization and objects made by the GC itself. The test is using a special monitor class that counts the number of objects freed during each collection and then inset the result into an array.

```
public void run() {
    GCMonitor monitor = new MyMonitor();
    GarbageCollector.registerMonitor(monitor);
    GarbageCollector.start();
    GarbageCollector.requestCollection();
    GarbageCollector.waitForNextCollection();
    array[0] = monitor.getFreeedObjects();
    monitor.reset();

    GarbageCollector.requestCollection();
    GarbageCollector.waitForNextCollection();
    array[1] = monitor.getFreeedObjects();
    monitor.reset();

    GarbageCollector.requestCollection();
    GarbageCollector.waitForNextCollection();
    array[2] = monitor.getFreeedObjects();
    monitor.reset();

    GarbageCollector.requestCollection();
    GarbageCollector.waitForNextCollection();
    array[3] = monitor.getFreeedObjects();
    monitor.reset();
```

Test code 11.

In `main` the result of running the mutator is verified. If all checks pass `main` will set `args` to `null` otherwise it will not change the value of `args` which indicates to the test system that the test failed.

The check here is that the value of `array[2]` in main contains the number of objects freed during the third collection. Since the GC did some work it can't be 0 and since there is no difference between the third and fourth collection the same number of objects (made by the GC) should be reclaimed. If the two check pass `args` will be set to `null` indicating success. The test is added to the HVM test suit and will be made during all full system tests.

```
public static void main(String[] args)  {
    int[] array = new int[4];

    Thread m = new Thread(new Mutator(array));

    m.start();

    try {
        m.join();
    } catch (InterruptedException e) {
    }
    for (int i = 0; i < array.length; i++) {
        devices.Console.println("array[" + i + "] = " + array[i]);
    }

    if (array[2] != array[3]) {
        return;
    }

    if (array[2] == 0) {
        return;
    }

    args = null;
}
```

Test code 12.

The test suits implemented tests the run time system interface to the GC. But it only tests some of the functionality of the GC itself. What remains are tests that check the write barrier and that the automatic scheduling of the GC works so that the system does not run out of memory.

Another range of tests that shuld be done are preformence tests. It would be interesting to see how much overhead there is when using the GC and how long the blocking pauses are in differente senarios.

9 Conclusion

As stated in chapter 1.1 we want a GC for the HVM with a small overhead both in terms of use of memory and load of the processor. Due to this we propose a concurrent mark-sweep GC with snapshot at the beginning and conservative root finding. The reason for our choices are that this solution gives a small overhead to the application both in regards to the memory used and extra work for the processor as described in chapter 3, while it does not impose a long pause like a stop the world GC.

But our solution is not easily analyzable with respect to memory consumption, because it does not prevent fragmentation and it does not give a guarantee against memory leaks. It is also only soft real-time since it has a potentially long blocking period while scanning roots. For many scenarios however we think the proposed GC will be useful. In a low-resource embedded system with few and small stacks and with few references in the static fields the pause imposed by the GC for finding roots might be acceptable. Also in a system like that, the same behavior is often repeated by the program. This means that the same type of objects will be allocated and freed again and again and this will not fragment the memory since the allocator will always try to fit new objects into holes the size of the new object.

The other important goal was to support the HVM being portable. The GC is therefore written in Java as a regular Java application. Only a few calls to C functions in the HVM are needed and they go through Hardware Objects. The addition of a GC to the HVM is therefore in line with the fundamental principles of the HVM.

10 Acknowledgments

I am deeply grateful to Stephan E. Korsholm for giving me the opportunity of making a GC for the HVM. He has been very patient in answering my many questions about the HVM and he has also been very helpful in writing the HVM interface to the GC and some of the tests. Without his help I could not have made this thesis.

I also very grateful to my counsellor A. P. Ravn for the helping me write this thesis. His critique and advises has been an invaluable help in shaping this paper.

Finally I would like to express my gratitude to my manager Jens Cramer head of the ICT engineering department at VIA for giving me the time to make this master thesis.

11 Literature

[1] Wilson, P. R. *Uniprocessor garbage collection techniques*.
Page 1-42. Memory Management. Springer. Berlin Heidelberg. (1992)

[2] . Korsholm, Stephan E. *Java for Cost Effective Embedded Real-Time Software*.
Ph.D. Dissertation, AAU. (August 2012)

[3] Puffitsch, W. *Hard Real-Time Garbage Collection on Chip Multi-Processors.*
Ph.D. Dissertation. Vienna University of Technology. (2012)

[4] Martin Schoeberl. *Scheduling of Hard Real-Time Garbage Collection*.
Page 176-213. Real-Time Systems, Volume 45, Issue 3. (August 2010)

[5] Michael Stilkerich et al. *Tailor-made JVMs for statically configured embedded systems.*
Page 789–812. Concurrency and Computation: Practice and Experience, 24 (2012)

[6] Isabella Thomm et al. *KESO: an open-source multi-JVM for deeply embedded systems.*
Pages 109-119. JTRES '10 Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. (2010)

[7] Fridtjof Siebert. *Hard Real-Time Garbage Collection in the Jamaica Virtual Machine.*
Pages: 96 – 102. Proceedings Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99). (1999)

[8] Fridtjof Siebert. *Concurrent, parallel, real-time garbage-collection.*
Pages 11-20. ACM Sigplan Notices Vol. 45, No. 8. (2010)

[9] Filip Pizlo et al.t al. *Real time Java on resource-constrained platforms with Fiji VM.*
Pages 110-119. JTRES '09 Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. (2009)

[10] Filip Pizlo et al. *Schism: fragmentation-tolerant real-time garbage collection*.
Pages 146-159. PLDI '10 Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation. (2010)

[11] Dijkstra, Edsger W., et al. *On-the-fly garbage collection: an exercise in cooperation*.
Pages 966-975. Communications of the ACM 21.11. (1978)

[12] Jones, Richard et al. *The Garbage Collection Handbook.* CRC Press, New York.(2012)

[13] Schoeberl, Martin, et al. *Hardware objects for Java.*
Page 445-452. Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on. IEEE. (2008)

[14] Henderson, Fergus. *Accurate garbage collection in an uncooperative environment*.
ACM SIGPLAN Notices. Vol. 38. No. 2 supplement. ACM, (2002)

[15] Siebert, Fridtjof. *Eliminating external fragmentation in a non-moving garbage collector for Java.*
Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems. ACM, (2000).

[16] CACAO VM http://www.cacaojvm.org/ last visited June 2013

[17] Krall, Andreas, and Philipp Tomsich. *Garbage collection for large memory Java applications*.
High-Performance Computing and Networking. Springer Berlin Heidelberg,(1999).

[18] Krall, Andreas, and Reinhard Grafl. *CACAO - A 64-bit JavaVM Just-in-Time Compiler*.
Page 1017-1030.Concurrency Practice and Experience 9.11 (1997):

[19] JAMVM http://jamvm.sourceforge.net/ visited June 2013

[20] Søndergaard, Hans, Stephan E. Korsholm, and Anders P. Ravn. *Safety-critical Java for low-end embedded platforms*. Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems. ACM, (2012).

12 Appendix: Garbage Collector source code.

12.1 GarbageCollector.java

```
package gc;

import icecaptools.IcecapCompileMe;
import reflect.ClassInfo;
import reflect.ObjectInfo;
import thread.Semaphore;
import thread.Thread;
import util.LiveSet;
import util.ReferenceIterator;
import vm.HVMHeap;
import vm.Heap;

public class GarbageCollector implements Runnable {
	public boolean GCstopped;

	private static Semaphore gcFinished;
	private static GCMonitor monitor;
	private static Thread gc;
	private static ReferenceIterator staticRoots;
	private static LiveSet liveSet;
	private static BitMap bitMap;
	private static boolean GCIsRunning;
	private static Heap heap;
	private Semaphore startGC;
	private Semaphore endGC;

	static {
		init();
	}

	@IcecapCompileMe
	private static void init() {
		GCIsRunning = false;
		heap = HVMHeap.getHeap();
		bitMap = new BitMap(heap.getBlockSize(), heap.getStartAddress(),
heap.getHeapSize());
		heap.setBitMap(bitMap);

		Object hack = new Object();
		newBarrier(ObjectInfo.getAddress(hack));
		writeBarrier(0, 0);

		short count = ClassInfo.getNumberOfClasses();
		do {
			count--;
			ClassInfo.getClassInfo(count);
		} while (count > 0);

	}
```

```
public GarbageCollector(Semaphore startGC, Semaphore endGC) {
	this.startGC = startGC;
	this.endGC = endGC;
	GCstopped = false;
}

@IcecapCompileMe
private static void writeBarrier(int source, int oldRef) {
	if (GCIsRunning) {
		if (Thread.currentThread() != gc) {
			if (!bitMap.isRefBlack(source)) {
				if (oldRef != 0) {
					if (bitMap.isRefWhite(oldRef)) {
						liveSet.push(oldRef);
					}
				}
			}
		}
	}
}

@IcecapCompileMe
private static void newBarrier(int ref) {
	if (GCIsRunning) {
		if (Thread.currentThread() == gc) {
			bitMap.shadeRefWhite(ref);
		} else {
			bitMap.shadeRefBlack(ref);
		}
	} else {
		if (bitMap != null) {
			bitMap.shadeRefWhite(ref);
		}
	}
}

private void setRoots() {
	liveSet.clear();
	staticRoots = heap.getStaticRef();
	while (staticRoots.hasNext()) {
		int nextRoot = staticRoots.next();
		if (nextRoot != 0) {
			monitor.addStaticRoot(nextRoot);
			liveSet.push(nextRoot);
		}
	}

	ReferenceIterator stackRoots = heap.getRefFromStacks();
	while (stackRoots.hasNext()) {
		int possibleRef = stackRoots.next();
		if (possibleRef != 0) {
			if (bitMap.isWithinRangeOfHeap(possibleRef)) {
```

```
                        if (heap.isRefAnObj(possibleRef)) {
                              if (heap.isRefAllocated(possibleRef)) {
                                    liveSet.push(possibleRef);
                                    monitor.addStackRoot(possibleRef);
                              }
                        }
                  }
            }
      }
}

private void traverse(LiveSet nodes) {
      ReferenceIterator children;
      int parent;

      while (!nodes.isEmpty()) {
            parent = nodes.pop();
            children = heap.getRefFromObj(parent);
            while (children.hasNext()) {
                  int child = children.next();
                  if (child != 0) {
                        if (bitMap.isWithinRangeOfHeap(child)) {
                              monitor.visitChild(parent, child);
                              if (!bitMap.isRefBlack(child)) {
                                    nodes.push(child);
                              }
                        }
                  }
            }
            bitMap.shadeRefBlack(parent);
      }
}

private void makeFreeList() {
      ReferenceIterator freeList = bitMap.getFreeList();
      while (freeList.hasNext()) {
            monitor.freeObject(freeList.next());
      }
}

public void run() {
      while (!GCstopped) {

            Thread.print("GC wait for next cycle");
            // run when signaled from controller thread
            startGC.acquire();
            GCIsRunning = true;

            Thread.getScheduler().disable();
            setRoots();
            Thread.getScheduler().enable();

            // start incremental traversal of object graph
```

```
                traverse(liveSet);

                GCIsRunning = false;
                // stops the write and new barrier -

                // free unused object
                makeFreeList();

                // signal controller that GC is finished
                endGC.release();
                gcFinished.release();
            }
        }

        public static void start() {
            Semaphore startGC = new Semaphore((byte) 0);
            Semaphore endGC = new Semaphore((byte) 0);
            gcFinished = new Semaphore((byte) 0);

            Thread controller = new Thread(new GarbargeCollectorController(startGC,
endGC));
            gc = new Thread(new GarbageCollector(startGC, endGC));

            liveSet = new LiveSet(bitMap);

            devices.Console.println("Bitmap size = " + bitMap.getSize());
            controller.start();
            gc.start();

        }

        public static void registerMonitor(GCMonitor m) {
            monitor = m;
        }

        public static void requestCollection() {
            GarbargeCollectorController.requestCollection();
        }

        public static void waitForNextCollection() {
            gcFinished.acquire();
        }
}
```

12.2 BitMap.java

```
package gc;

import util.ReferenceIterator;
import util.ReferenceList;

public class BitMap {

	public static final byte WHITE = 3;
	public static final byte GREY = 1;
	public static final byte BLACK = 2;
	public static final byte HATCHED = 0;

	private int blockSize;
	// In int/words

	private int startAdr;
	// In int/words

	private int heapSize;
	// In int/words

	private int bitMapSize;

	private byte[] bitMap;

	public BitMap(int blockSize, int startAdr, int heapSize) {
		this.blockSize = blockSize;
		this.startAdr = startAdr;
		if (heapSize > 64000){
			heapSize = 64000;
		}
		this.heapSize = heapSize;
		bitMapSize = ((heapSize / blockSize) / 4) + 1;
		bitMap = new byte[bitMapSize];
	}

	public void clear() {
		for (int i = 0; i < bitMap.length; i++) {
			bitMap[i] = 0;
		}
	}

	public boolean isRefWhite(int ref) {
		int index = getIndex(ref);
		int offSet = getOffset(ref);
		if (((bitMap[index] & (1 << (offSet << 1))) != 0) && ((bitMap[index] & (1 <<
((offSet << 1) + 1))) != 0)) {
			return true;
		}
		return false;
	}
```

```
	public boolean isRefGrey(int ref) {
		int index = getIndex(ref);
		int offSet = getOffset(ref);

		if (((bitMap[index] & (1 << (offSet << 1))) != 0) && ((bitMap[index] & (1 <<
((offSet << 1) + 1))) == 0)) {
			return true;
		}
		return false;
	}

	public boolean isRefBlack(int ref) {
		int index = getIndex(ref);
		int offSet = getOffset(ref);

		if (((bitMap[index] & (1 << (offSet << 1))) == 0) && ((bitMap[index] & (1 <<
((offSet << 1) + 1))) != 0)) {
			return true;
		}
		return false;
	}

	public boolean isRefHatched(int ref) {
		int index = getIndex(ref);
		int offSet = getOffset(ref);

		if (((bitMap[index] & (1 << (offSet << 1))) == 0) && ((bitMap[index] & (1 <<
((offSet << 1) + 1))) == 0)) {
			return true;
		}
		return false;
	}

	public void shadeRefGrey(int ref) {
		int index = getIndex(ref);
		int offSet = getOffset(ref);

		bitMap[index] = (byte) (bitMap[index] | (1 << (offSet << 1)));
		bitMap[index] = (byte) (bitMap[index] & ~(1 << ((offSet << 1) + 1)));
	}

	private int getOffset(int ref) {
		return ((ref - startAdr) / blockSize) & 0x3;
	}

	private int getIndex(int ref) {
		return ((ref - startAdr) / blockSize) >> 2;
	}

	public void shadeRefWhite(int ref) {
		int index = getIndex(ref);
		int offSet = getOffset(ref);
```

```
		bitMap[index] = (byte) (bitMap[index] | (1 << (offSet << 1)));
		bitMap[index] = (byte) (bitMap[index] | (1 << ((offSet << 1) + 1)));
	}

	public void shadeRefBlack(int ref) {
		int index = getIndex(ref);
		int offSet = getOffset(ref);

		bitMap[index] = (byte) (bitMap[index] & (~(1 << (offSet << 1))));
		bitMap[index] = (byte) (bitMap[index] | (1 << ((offSet << 1) + 1)));
	}

	public ReferenceIterator getFreeList() {
		ReferenceList freeList = new ReferenceList();
		int i = 0;
		while (i < bitMap.length) {
			byte nextByte = bitMap[i];
			if (nextByte != 0) {
				int ref = ((i << 2) * blockSize) + startAdr;
				for (int j = 0; j < 4; j++) {
					if (isRefWhite(ref)){
						freeList.add(ref);
						shadeRefHatched(ref);
					}
					else if (isRefBlack(ref)){
						shadeRefWhite(ref);
					}
					ref += blockSize;
				}
			}
			i++;
		}
		return freeList.iterator();
	}

	private void shadeRefHatched(int ref) {
		int index = getIndex(ref);
		int offSet = getOffset(ref);

		bitMap[index] = (byte) (bitMap[index] & ~(1 << (offSet << 1)));
		bitMap[index] = (byte) (bitMap[index] & ~(1 << ((offSet << 1) + 1)));

	}

	public boolean isWithinRangeOfHeap(int possibleRef) {
		return (possibleRef >= startAdr) && (possibleRef < startAdr + heapSize);
	}

	public int getSize() {
		return this.bitMapSize;
	}
}
```

12.3 TestGCSimple.java

```
package test.icecapvm.minitests;

import icecaptools.IcecapCompileMe;
import gc.GCMonitor;
import gc.GarbageCollector;
import thread.Thread;

public class TestGCSimple {

    private static class MyMonitor extends DefaultGCMonitor {
        @Override
        public void visitChild(int parent, int child) {
            super.visitChild(parent, child);
        }

        @Override
        public void addStaticRoot(int ref) {
            // devices.Console.println("static: " + ref);
        }

        @Override
        @IcecapCompileMe
        public void addStackRoot(int ref) {
            // devices.Console.println("stack: " + ref);
        }

        @Override
        public void freeObject(int ref) {
            super.freeObject(ref);
            // printRef(ref, "free: ");
        }
    }

    private static class Mutator implements Runnable {

        int[] array;

        public Mutator(int[] array) {
            this.array = array;
        }

        @Override
        public void run() {
            GCMonitor monitor = new MyMonitor();
            GarbageCollector.registerMonitor(monitor);
            GarbageCollector.start();
            GarbageCollector.requestCollection();
            GarbageCollector.waitForNextCollection();
            array[0] = monitor.getFreeedObjects();
            monitor.reset();
```

```
            GarbageCollector.requestCollection();
            GarbageCollector.waitForNextCollection();
            array[1] = monitor.getFreeedObjects();
            monitor.reset();

            GarbageCollector.requestCollection();
            GarbageCollector.waitForNextCollection();
            array[2] = monitor.getFreeedObjects();
            monitor.reset();

            GarbageCollector.requestCollection();
            GarbageCollector.waitForNextCollection();
            array[3] = monitor.getFreeedObjects();
            monitor.reset();


        }
    }

    /**
     * @param args
     * @throws InterruptedException
     */
    public static void main(String[] args)  {
        int[] array = new int[4];

        Thread m = new Thread(new Mutator(array));

        m.start();

        try {
            m.join();
        } catch (InterruptedException e) {
        }
        for (int i = 0; i < array.length; i++) {
            devices.Console.println("array[" + i + "] = " + array[i]);
        }

        if (array[2] != array[3]) {
            return;
        }

        if (array[2] == 0) {
            return;
        }

        args = null;
    }
}
```