Garbage Collection supporting automatic JIT parallelization in JVM

Author: Erik Österlund
Date: 2012-06-26
Subject: Computer Science
Level: Master
Course code: 4DV01E
**Abstract**

With increasing clock-rates in CPUs coming to an end, a need for parallelization has emerged. This thesis proposes a dynamic purity analysis of objects, detecting independent execution paths that may be run in parallel. The analysis relies in speculative guesses and may be rolled back when proven wrong. It piggybags on an efficient replicating garbage collector integrated to JVM.

The efficiency of the algorithms are shown in benchmark, and are comparable to the speed of state of the art garbage collectors in hotspot’s JVM.

With this dynamic purity analysis now accessible in Java programs, the potential for automatic JIT-parallelization of pure methods is possible.

**Keywords:** garbage collection, dynamic analysis, automatic parallelization
9. FUTURE WORK ................................................................. 22
10. REFERENCES ................................................................. 23
List of Figures

FIGURE 3.1 STRONGLY CONNECTED COMPONENTS ................................................................. 3
FIGURE 3.2 CONDENSED GRAPH ......................................................................................... 4
FIGURE 3.3 HEAP CELLS AND THEIR PROPERTIES ............................................................ 5
FIGURE 3.4 MUTABLE, IMMUTABLE AND PURE OBJECTS ............................................... 6
FIGURE 3.5 GENERATIONAL HEAP ..................................................................................... 7
FIGURE 5.1 REPLICATING GARBAGE COLLECTOR .......................................................... 12

List of Tables

TABLE 7.1 BENCHMARK RESULTS ................................................................................... 20
1. Background

In this chapter a small introduction covering why parallelization is getting increasingly important is provided, and how automatic parallelization fits the mainstream market.

1.1. Introduction

We are entering an era with (among others) two problems: 1) Clockrates can no longer be increased as it used to be the last couple of decades on the PC market. Hence computational performance is no longer increasing as it used to. 2) Mobile devices are growing in popularity and in the range of tasks they can perform, but there are many problems in increasing performance in these devices while keeping the temperatures and energy consumption low as battery technology doesn’t seem to progress in the speed of light. These problems are an effect of Moore’s law coming to an end (Moore, 2005).

A common relief for these problems is that of parallelization and increasing the number of computational units. It allows us to increase computational performance without increasing clockrates. And by keeping clockrates down, we are able to be more energy efficient and to put unused computational units to sleep. Already today we see PCs with 16 CPU cores and mobile tablet devices with 4 CPU cores, and perhaps we can expect these numbers to grow if we can make software take advantage of these cores.

This approach requires software to be able to use the increasing number of computational units. In order to help software in general to take advantage of the new hardware, a Garbage Collector (GC) capable of finding subgraphs of objects that are temporarily independent from the rest of the system, referred to as pure objects was created by me. Methods invoked on these objects may be automatically parallelized as they can be asynchronously executed.

1.2. Report structure

The remainder of this report is structured in the following way. Chapter 2 goes through what the problem is and what the goal is with this thesis. Chapter 3 introduces some terminology and required knowledge to be able to read and understand this report. Chapter 4 mentions some related work in this field, Chapter 5 goes through the solution to the problem and Chapter 7 evaluates the results. Chapter 8 concludes this thesis and Chapter 9 mentions some future directions for research.
2. Problem and Goal

This chapter describes the problems to be solved and what the end goals are. In Chapter 8 the results and whether the goals were fulfilled is discussed.

2.1. Problem

I previously made a garbage collector capable of finding pure objects for free, with potential for just-in-time (JIT) parallelization (Österlund & Löwe, 2012). It specifically targets object oriented languages with accurate pointers. The main problem is that it was not integrated with a real virtual machine (VM) for such an object oriented (OOP) language. Hence the first step that has to be taken which is addressed in this thesis, is to integrate it into such a VM. Java will be targeted as it is one of the most widely used object oriented languages with accurate pointers, supporting moving garbage collectors.

2.2. Goal

The goal with this degree project is to get an already functional GC framework written by me to become integrated into a VM and its runtime system.

The integrated solution has to be efficient in a parallel environment, hence performance will be prioritized over everything else, like for instance portability and understandability. To fulfill this requirement, thread-local allocation buffers, thread-local mutation buffers and parallel garbage collection must be supported. In case memory overhead or computational overhead must be prioritized, memory will in general be sacrificed. All information needed to do JIT-parallelization must be accessible in the JVM so that the analysis collected may be used in practice. However, the mechanism for doing the actual JIT-parallelization is outside the scope of this thesis.
3. Required knowledge

In this chapter, some notions and basic knowledge vis-à-vis garbage collection and algorithms are introduced.

3.1. Strongly Connected Components

A strongly connected component from graph theory is a maximum set of nodes such that each node in the set can transitively reach every other node in the set.

Figure 3.1 shows the SCCs of an example graph, and Figure 3.2 shows the same graph condensed to its SCCs. Note that the first graph is a directed graph, and the condensed graph is a directed acyclic graph (DAG).

![Strongly connected components](image)

Figure 3.1 Strongly connected components
3.1.1. Roots
Roots are pointers from outside the heap into the heap. For instance, pointers in the stack or global memory into the heap are considered roots. Figure 3.3 demonstrates this graphically.

3.1.2. Cells, oops and objects
A memory cell is a region of memory in the heap allocated by a memory management system. In the hotspot JVM, this is referred to as oops and in object oriented languages these are objects. The terms will be used almost interchangeably, depending on which perspective they are seen from. If seen from garbage collectors in general, the term cell will be used. From the hotspot JVM’s perspective they will be called cells and from the mutators’ perspective, they will be called objects.

3.2. Cell liveliness properties
A cell is referred to as a live cell iff it can be transitively reached from the roots. It is referred to as a dead cell iff it can not be transitively reached from the roots. It is the purpose of a garbage collector to find dead cells and recycle their memory. Figure 3.3 demonstrates this graphically.
3.3. Dijkstra’s color coding

Dijkstra colors cells in 3 different colors; white cells, grey cells and black cells (Dijkstra, Lamport, Martin, Scholten, & Steffens, 1976). A cell is considered white if it has not yet been found by a tracing garbage collector. Grey cells have been found by the garbage collector, but their children have not. Dijkstra’s definition of a black cell is a cell that has been found by the garbage collector and whose immediate children have also been found by the garbage collector. However, for our purposes, we will extend this definition to say all transitively reachable cells have been discovered.

3.4. Pure Objects

A pure object is an immutable object whose transitively reachable children are all immutable, i.e. a pure object is an immutable cell in a strongly connected component of only immutable cells. Error! Reference source not found. Figure 3.4 demonstrates these concepts graphically.
3.4.1. Pure Methods
Pure methods are methods where all arguments (including the implicit ‘this’ argument), are all pure. We observe that pure methods may not change global state, and hence optimizations such as parallelization of invocations to such methods are possible.

3.4.2. Generations
Generations are partitionings of the heap into smaller areas, just like a semispace. The major difference between a generation and a semispace is that a generation can be independently collected, ignoring the rest of the heap. To make this possible, something called remembered sets link generations together, remembering the pointers between them, referred to as generational pointers. The inter-generational pointers in the remembered set are then regarded as roots when collecting a generation. Typically the heap has generations for cells of different age. Since most objects die young, it is useful to have a young and an old generation. The old cells tend to survive much longer, and collecting them over and over can be wasteful.

Garbage collectors that use generations are referred to as generational garbage collectors. Figure 3.5 demonstrate in graphical form how a generational heap can look.

Figure 3.4 Mutable, immutable and pure objects
3.5. Safepoints

A *safepoint* is a synchronization point where only one thread is allowed to execute. Safepoints are used when the garbage collector needs to sample roots or synchronize semispaces and make sure both mutators and collectors have the same view of the world. In the interpreter of a virtual machine, safepoints can be checked for between the execution of instructions. As for JIT-compilers, code is emitted for safepoints at certain points so that it is always possible to reach a safepoint quickly. This means they are emitted in loops so a loop may not stall the synchronization process.

When a safepoint is requested, it is signalled that a safepoint is needed. The other threads will eventually run into a safepoint, and then suspend execution. Everything will stall until all other threads are suspended. The critical code is then run, and the threads are resumed.

It is crucial that the time to reach a safepoint is small so that one single thread is not preventing all other threads from executing. Likewise, it is crucial that the task to be done in this critical section is very quick so that threads may execute as much as possible.
3.6. Semispace
A semispace is a partitioning of the heap into smaller regions. This is similar to the concept of a generation. However, the difference is that pointers between semispaces are not tracked. Typically, only one semispace is visible at a time. In a copying garbage collector, there are two semispaces; from-space and to-space. Only to-space is visible to mutators.

3.7. Mutators and collectors
Mutators are the threads of an actual application, in contrast to the collector thread that collects garbage. It is useful to differ between the two because they see the world from different perspectives and have different concerns. The mutators know nothing about the memory management and expects the collector to do all of that.

3.8. Barriers
There are two types of barriers in a garbage collectors, and typically each type of garbage collector only uses one of them. The first type of barrier is the read barrier, where read operations from the heap are protected by arbitrary code in an arbitrary way. A typical implementation is to use read-protected pages of memory.

A write barrier, similarly, protects the heap from various types of write operations. Most garbage collectors require write barriers only for pointer store operations to heap cells.

3.9. Moving garbage collection
A moving garbage collector is a garbage collector that may move cells in memory while executing. Some garbage collectors refer to this as scavanging. The purpose of this is typically to improve caching performance by moving surviving cells closer physically in memory.

3.10. Accurate and conservative garbage collection
In the garbage collection community, accurate garbage collection is garbage collection that requires the runtimesystem and/or compiler to help identify exactly where pointers are located to cells. Typically, all moving garbage collectors need accurate pointers. A conservative garbage collector on the contrary do not require accurate pointers.
4. Related work

In this chapter some critical, closely related work is studied to widen the views into the subject of garbage collection.

4.1. Mark & Sweep garbage collection

Mark & Sweep garbage collectors such as Boehm-Demers-Weiser (Boehm & Weiser, 1988) collect garbage in two phases. The first phase is the marking phase. In the marking phase the heap is traversed from roots, and all cells the garbage collector finds are marked as live. The second phase, sweeping, linearly scans through the heap and reclaims memory for any dead cells encountered. Cells are considered dead if they are not marked.

A variant of Mark & Sweep collection is mark compact collection. It adds one more phase; compacting the heap. With this approach, live cells are moved physically close to each other in memory. This leads to improved memory locality and hence cache performance for mutators.

The marking phase is a tracing phase, proportional to the number of live cells in the heap. The sweeping phase, however is proportional to the size of the heap, including both live and dead cells in memory. Hence, the lower the heap residency, the less efficient the mark & sweep approach becomes. Empirical studies show that typically the heap residency is low in the heap of object oriented programs; cells die young.

However, while from a computer science point of view, O(heap) seems worse than O(live cells), in practice, it is the other way around. Because the sweeping phase is a linear scan of the heap, the memory access pattern is easy for caches to understand and with prefetching the sweeping phase is quicker than the tracing marking phase where memory access patterns are more random, depending on memory locality.

4.2. Copying garbage collection

Copying garbage collectors such as the one described in (Cheney, 1970) typically divide the heap into two semi-spaces: from-space and to-space. Only one semi-space is visible to mutators at any point in time. The garbage collector collects garbage by traversing live cells from the roots, copying them over from from-space to to-space. When all live cells have been copied, sides are flipped; from-space becomes to-space and vice versa. This way, all live cells have been preserved and been moved physically close in memory, improving memory locality and hence caching performance.

Another major benefit with the copying garbage collector is its linear memory access patterns for allocating memory and copying live cells to to-space. With prefetching, the caches will perform well and memory allocation is very efficient.

A concurrent variant was made by (Baker, 1978). In fact, when talking about copying garbage collectors, it is Baker’s algorithm that we refer to.

In order to collect garbage concurrently while mutators are executing, only to-space is kept visible to mutators. To protect the mutators from seeing from-space, from-space is protected with a read-barrier, typically by having operating system traps (read protection on the pages). When a read barrier is invoked, the mutator is suspended, copies over live cells in the page it read from, and then resumes execution.

The collector concurrently copies cells from from-space to to-space using BFS traversal. A pointer points to the lowest cell seen by the collector. Then it continues forward, copying the children to the end of from-space. This effectively uses the heap as a stack for the BFS traversal.

The major con with this approach is that the read barrier is expensive compared to the write barrier of concurrent mark and sweep collectors.
4.3. Reference counting garbage collection

Reference counting garbage collectors do not have a tracing phase, meaning that they never traverse the heap from the roots like the other two major approaches to garbage collection. Instead, it releases and retains cells on store operations. Each cell hence has a counter, representing how many cells reference this cell. When a pointer store operation is issued, the currently referenced cell is released, decreasing the retain count by one, and the newly referenced cell is retained, increasing the retain count by one. When a cell reaches a retain count of 0, it is dead and its memory is recycled by the garbage collector.

The major drawback with this approach is its inability to find cycles. If cell a references cell b and cell b references cell a, then both will have a retain count of 1, even though they are not reachable from the roots. To deal with this, they are either accepted as memory leaks, or a hybrid garbage collector is used, using reference counting most of the time, but switching to, for instance, mark and sweep occasionally, to get rid of this memory leak eventually.

Note that Apple’s latest invention, which they refer to as automatic reference counting (Apple), is a variant of a reference counting garbage collector, trying to use language semantics to deal with cycles by letting the programmers denote which references are strong or weak.
5. Solution

In this chapter the conceptual solution is first covered, describing what we need. It is divided into two major subsections; garbage collection and purity analysis respectively. A subsection concerning how to use the analysis for parallelization is provided in the end.

5.1. Garbage collection

The solution to the whole problem is dynamic purity analysis piggybagging on an efficient garbage collector. For this, we need to carefully pick a suitable garbage collection algorithm.

5.1.1. Replicating garbage collection

A replicating garbage collector such as the one described in (Nettles & O'Toole, 1993) is a variant of a copying garbage collector. There a major difference from a regular copying collector. In an ordinary copying collector, from-space is invisible for mutators; they see only to-space. To make this possible, a read-barrier is implemented to copy cells to to-space, point to the new object and then resume mutator execution. For a replicating garbage collector, it is the other way around. Only cells in from-space are visible to the mutators. The collector then replicates the cells in from-space to to-space by tracing through the roots. Instead a write barrier is used to synchronize the two semispaces.

This means that for a normal copying collector, both collector and mutators copy cells to-from-space which they both can see. An implication is that collector and mutators hence always are in synch, but at a high cost. If any cell is read in from-space, the world needs to be stopped until it can be copied to to-space.

In a replicating collector, however, the collector and the mutators are n synch only when flipping sides at safepoints. Between those occasions, the mutator mutates cells in from-space, and logs these mutations in a mutation log which is read by the collector to keep to-space in synch.

A replicating garbage collector has a number of phases. When garbage collection is started the following steps are taken:
1. Safepoint is entered
2. Roots are sampled
3. Safepoint is exited
4. Heap is traversed from roots, replicating live cells from from-space to to-space
5. Safepoint is entered
6. Roots are sampled
7. Heap is traversed from roots, replicating live cells from from-space to to-space
8. Semispaces are flipped; from-space become to-space and vice versa
9. Safepoint is exited

Step 1 – 4 can be repeated any number of times. Sampling the roots more often leads to less waste of memory when sides are flipped as some cells were live when garbage collection started, but mutators changed the topology of the heap while garbage collection executed in parallel, rendering some previously live cells dead. The con is increased downtime for all mutator threads. The last traversal must always occur while in a safepoint so semi-spaces are synchronized when sides are flipped. Figure 5.1 depicts an idea how a replicating garbage collector looks graphically.
Figure 5.1 Replicating garbage collector

5.1.2. Traversal strategies

A requirement for this garbage collector for the analysis to work is, as we will see later, that Depth-First-Search (DFS) traversal is used when collecting garbage. Normal copying collectors typically use Breadth-First-Search (BFS) traversal as that allows them to use the heap as a natural stack for the BFS traversal. Mark & sweep typically uses a marking stack for the tracing phase.

An alternative to stack based DFS traversal is pointer reversal (Schnorr & Waite, 1967). The replicating garbage collector is the only garbage collector which is not a stop-the-world collector but still allows this. Normally it would be very difficult to use pointer reversal for the garbage collection in the same semi-space as the mutators run is as the pointers would be broken for the mutators. With a replicating garbage collector this is not a problem.

The benefit with stack based traversal is slightly improved performance. The con is that stack overflows have to be handled. In a deep graph, the DFS stack can reach its limits. This is often handled by rotating the stack, restarting at the bottom, forgetting where it started. The algorithm is then called iteratively, collecting some garbage each iteration. However, for the analysis to work, we require strict DFS traversal and can not afford breaking as violations to DFS order is a violation of the purity analysis algorithm.

5.1.3. Parallel collection

To collect garbage in parallel to the normal execution of the mutators, a few things have to be considered. The images of from-space and to-space have to be synchronized. The mutation log is used for this. Whenever a store operation is issued to a heap cell, we say it has mutated. This is logged in a mutation log. The mutation log can be a buffer manifesting an array of pointers to cells that mutated.

Note that we wish to log what cell mutated, not what field mutated inside of it. This because of various reasons. Firstly, copying the whole cell instead of a field is not much of an overhead as it is the caching of that piece of memory that takes time. Unless the cell is very big, there is not much overhead in copying the whole cell again. Also, if different parts of a cell changed at different occasions (which is not unlikely), it is more
efficient to copy the whole cell once instead of copying it piece by piece. Furthermore, the mutation log itself will have a bigger memory footprint if storing more details about what was stored, and it is difficult to do so in atomic operations so the cells in from-space and to-space are properly synchronized as noted by (Azagury, Kolodner, & Petrank, 1998).

The garbage collector reads the mutation log, copies the mutated cells over to to-space and then and re-starts garbage collection from those cells, treating them as roots.

All store operations need logging, even those to primitive values, as the two semi-spaces need synchronization.

5.2. Purity analysis

The purity analysis aims at finding pure objects. We recall pure objects are immutable objects and may only transitively reach immutable objects. Hence, every pure object is pure iff its strongly connected component consists only of immutable objects.

The approach picked is to conceptually condense the object graph into its strongly connected components, effectively transforming it into a Directed Acyclic Graph (DAG).

If an object is mutable or can transitively reach mutable objects, it is marked as dirty, meaning it can not be pure, nor can any object transitively reaching this object. When traversing this DAG in DFS order, SCCs containing a mutable object are marked as dirty. This dirty property is then propagated bottom-up in the graph. As a result, all dirty SCCs will be correctly marked, meaning that an object is pure iff its SCC is not marked as dirty.

5.2.1. Tarjan’s algorithm

Tarjan’s algorithm (Tarjan, 1972) is an efficient algorithm for detecting SCCs in a rooted graph. It traverses the graph in DFS order from its roots. In order to work, it keeps track of an index field in every node visited. This field indicates the DFS order in which a node was visited. The algorithm also keeps track of a “lowest” field, determining the lowest index field that can be transitively reached from a given cell, hence the first cell to be entered in a given SCC. Another requirement of the algorithm is a stack to keep track of all cells in the current SCC being visited.

When the algorithm enters a cell it sets the index field to the previous index + 1, and the lowest field to its index field, if the index field was not yet set, meaning it has not yet been traversed. We also push these nodes to the stack to remember them. If it has an index field set, that means we reached a cycle, and traversal forward stops. When traversing backwards, the lowest field of the previous cell is set to the minimum of the lowest fields of the previous and current nodes. This means that the lowest field is propagated backwards. When stepping backwards to a cell that has its lowest field equal to itself, that means we found a complete SCC. The stack is then popped until the current node is found. The nodes popped from the stack are those that belong to the same SCC.

5.2.2. Mutation log

The mutation log used to keep from-space and to-space needs to log whenever cells are changed with store operations. This information can be used by the purity analysis to decide what cells are flagged as dirty. Since this write barrier is needed regardless, the extra cost to record information needed by the purity analysis is almost nothing.

When the garbage collector is active, cells are flagged as dirty if they mutate and an entry is added to the mutation log. However, if the garbage collector is not active, no
record is saved for the mutation, but it is still flagged as dirty. In this case, the garbage collector does not need the information, only the purity analysis.

5.3. Automatic JIT parallelization

Using the purity analysis for automatic JIT-parallelization is outside the scope for this thesis. However, it is useful to still mention how this could be used to further motivate the work.

The purity analysis can be used for automatic JIT-parallelization. Well tuned virtual machines such as the hotspot JVM have profilers to detect methods that the VM spends more time in. This is typically done by having a timer sample the stack frames in use. This information is then used to optimize methods that are frequently used by using an appropriate JIT-compiler, emitting machine code for such methods. In hotspot JVM from OpenJDK, there are different JIT compilers for different optimization levels. We observe that the same optimization detection technique can be used also for automatic parallelization of method invocations.

Several versions of methods can be compiled, some with assumptions that certain objects are pure, where parallelization transformations are applied and pure methods are executed in parallel, and a sequential version where such assumptions are not made. Then using context aware composition, the best version yielding the best performance can be picked.

An alternative to picking between several pre-compiled versions of a method is to make JIT-compilers compile the parallel variants on the fly as pure objects are found in the code.

One strategy to exploit this is to parallelize loops. As programs spend a lot of time inside of loops, it is useful to be able to execute their bodies in parallel. The iteration-space can be split into blocks of different granularity to be enqueued and executed in parallel. A sequential version would simply execute these blocks in order, while a parallel variant may elect to execute blocks ahead of time in parallel.

5.3.1. Rolling back

It is important that it is kept in mind that the speculative analysis may be wrong; a supposed pure object can turn out to be not pure. This needs to be handled efficiently.

One way of dealing with this is to let the write-barrier detect consistency violations before they happen. By letting write barriers be emitted before the mutation takes place, the write barrier can see if the given object is assumed to be pure, and this is currently exploited by the runtime system. If this is the case, it stops the mutator executing this block, and all blocks executing ahead of time in comparison to this block.

Note that the blocks being executed in sequential order are unaffected by this. The only blocks that can be cancelled are those executing ahead of time. This means that in the worst case, computations ahead of time are discarded, but without negatively affecting the execution time compared to the sequential version.

To prevent the rollback from happening too often in case many assumptions were to be wrong, it is possible to stop parallelization efforts until next garbage collection cycle. This way, the cost for rolling back is negligible as it happens at most once per garbage collection cycle.

5.3.2. Concurrency considerations

The purity analysis algorithm needs the rooted graph to be traversed in DFS order to accurately detect immutable SCCs and conclude their purity property. As mutators are allowed to execute in parallel, the rooted graph changes topology over time as the
garbage collector traces live cells. This means that the DFS order can be broken, and the purity analysis rendered incorrect.

This is less of a problem than it seems to be. First of all, this is a speculative optimistic analysis, and it can handle incorrect assumptions already. The important thing is that mutators keep a consistent state and hence do not allow incorrect assumptions about purity. If the garbage collector makes an incorrect assumption of purity, that does not mean that mutators allow this to make invalid state transitions in the program. Since memory store operations are protected by write barriers, the incorrect assumption will simply be rolled back if the garbage collector was wrong.

It is still useful to look in depth at what can happen. In this garbage collector can be in three different states:

1. Not collecting garbage
2. In first partial collection
3. In second final partial collection

The vast majority of all time is spent in the first state, and here the garbage collector can not make incorrect assumptions about purity as it is not executing. Furthermore, this only leads to roll-backs when assumed pure objects turn out to be not pure. The only way this incorrect assumption can be made during the garbage collection, is if cells are marked as pure in the first partial collection then mutate; this is not properly detected.
6. Implementation

This chapter describes some key points how this was implemented to reach the performance requirements. The garbage collector was implemented in C++ and integrated into the hotspot JVM in OpenJDK 7 (Oracle). C++ was chosen because the GC interfaces in the VM require GC implementations to be written in C++.

6.1. Contiguous allocation region

To make allocation of JVM oops faster, a garbage collector should support contiguous allocation regions, referred to as “eden” in hotspot JVM. The benefit with contiguous allocation regions is that they support linear writes to memory which is very beneficial for caches, instead of finding free memory fragments with random access patterns.

Another benefit with contiguous allocation regions is that the JIT compilers support them. This means that machine code for oop allocation is generated, making memory allocation much faster.

In the replicating garbage collection introduced in this thesis, contiguous memory allocation is natural as it is a type of copying collector. The allocation region is located in from-space, right after the memory that was evacuated the last garbage collection. The whole heap is thus contiguous.

6.2. Thread local allocation buffer

The idea of thread local allocation buffers (TLABs) is the same as for contiguous allocation regions. It is a contiguous allocation region by itself, but is local for each thread. To allocate memory efficiently with multiple thread, this is crucial. If two mutators allocate memory in the same region, the caches conflict as they access the same cache lines.

TLABs also have JIT compiler optimizations just as the normal contiguous allocation region does, resulting in very high allocation speeds. The speed of cell allocation is roughly equal to the speed of calculating the size of the corresponding oop. The size of the TLAB itself should be aligned with given caches to avoid conflicts with other threads.

To implement this, threads have the ability to commit ownership of contiguous regions of the heap. When a TLAB is owned, machine code efficiently allocates cell. Eventually the TLAB runs out of memory. Then a request is sent to the garbage collector to provide a new memory region for a new TLAB. Then this process is repeated. Important to notice is that synchronization mechanisms are necessary only when acquiring a new TLAB, which can be arbitrarily cheap by varying the size of the TLAB.

6.3. Optimizing Tarjan’s algorithm

Tarjan’s algorithm can be further optimized for replicating garbage collectors. It traditionally requires us to keep track of an index field for each cell, which is its corresponding DFS order value. Further more, it needs to keep track of a root field for each cell, a pointer word pointing to the first cell in a given SCC. The third requirement is a stack for finding which cells are inside of a given SCC. In a replicating garbage collector, all of these storage overheads can be eliminated.

The index field for the DFS order value of the cell is given to us for free because cells are replicated to to-space in DFS order. This means that the memory address of the cells will also have the same DFS order.

The stack for finding which cells are in the same stack may also be removed safely. For our purposes, it is sufficient to know what root cell any cell in an SCC has, not which the other cells are. This information can be obtained by checking if the root of a
certain cell is black. The cell is in the conceptual stack iff its root is not black which is flagged by the garbage collector.

The root field requires one word in each cell. But it can also be optimized away by carefully examining what information is needed at what time in the two semispaces. We observe that the root field is needed in to-space for the collector to find out which cells are pure. Further we note that all moving collectors need a forwarding address to the corresponding new cell. In the case of a replicating garbage collector, this points to the replica in to-space and is used only by cells in from-space.

Conclusively a pointer word for forwarding is only needed in from-space and a pointer word for the root cell is needed only in to-space. Hence they can use the same value. Care needs to be taken to make sure the mutators accurately detect pure objects as the content of this field has two cases. With this optimization, the mutators need to check if the replica pointer word is in from-space. If it is, it is the old root and if it is marked as not dirty, the given cell is pure. In the second case, where a mutator finds a replica pointer word pointing to to-space, it simply looks if this cell is flagged by the garbage collector as pure (which is done as it is replicated using the old root).

The flags for the purity analysis can also be optimized away by embedding them in the pointer word for replicas or roots. Since cells have an alignment in the heap, a few bits can be spared for these flags.

To summarize, the memory overheads of the purity analysis including tarjan’s algorithm can all be optimized away from the replicating garbage collector, and it suffers no higher memory overhead than a regular replicating garbage collector without any purity analysis. The cell header memory overhead is reduced to a single pointer word.

6.4. Closures

The hotspot JVM extensively using something they call closures to perform operations on the rooted object graph, cell by cell. In this context, a closure is an object that can provide answers to queries, i.e. if a cell should be kept alive or not, or to perform operations, i.e. keeping a cell alive. There are different kinds of closures with different names.

An object closure is an object with a method, do_object() which takes an object as an input and does something arbitrary with it like for instance marking it. Likewise, an oop closure is a closure that has a method, do_ooop() which takes fields of objects as parameters and does something arbitrary with them like for instance following references to children.

From an engineering point of view, the purpose of this mechanism is to separate the general traversal strategies and general virtual machine logic, from the specific garbage collection algorithms, letting specific closures perform tasks specific to different garbage collectors. As the hotspot JVM supports multiple garbage collectors, this mechanism is useful for abstracting away the specifics of certain garbage collectors. However, it comes at a great cost if not treated with care. These closures are invoked many times. When traversing the heap, the number of times they are invoked is proportional to the heap residency.

These closures use virtual member functions to perform their work. Virtual member functions are the answer to dynamic binding in C++. Like all operations with dynamic binding, they come at an extra cost for looking up the correct version to be used. In order to minimize this cost and allow the modularity with minimal performance losses, C macros were added to translate the calls to known statically bound methods instead. If not statically bound, some performance is lost.
Another note on closures is how they impact the time complexity of the two traversal strategies used. The way cells are traversed is that oops can tell closures which their children are. But it is not possible to query the number of children and the child at a certain index. All children have to be queried at once.

For the stack-based traversal this has no impact. All children are queried and put on the stack, and traversed in DFS order. However, for the pointer reversal algorithm where no stack is used, this becomes a problem. When entering a cell, going to the next cell becomes $O(n)$ over the number of children $n$, instead of constant as it ought to be. It is possible but cumbersome to make it possible to query the number of children and specific children at specific indices. This was not done as it affected far too much code. Instead, stack-based traversal is used whenever possible, and pointer-reversal is used as a backup-plan in case the stack overflows.

6.5. Compressed oops

The hotspot JVM from OpenJDK supports compressed oops for 64-bit architectures. Pointers in 64-bit architectures are 64 bits, while they are only 32 bits in 32-bit architectures. 32-bit addresses can address 4 GB of memory. However, in a heap, oops have an alignment and the size of the heap being addressed is big enough for most applications. Hence the hotspot JVM supports having 32 bit addresses for oops instead, by using the start of the heap as a baseline. The address is then an offset from the base.

Currently the implementation does not support compressed oops, so memory overhead is a bit bigger, but memory accesses are slightly faster as they are direct with no computations required (although this is heavily optimized).

6.6. Mutation logs

In order to make mutation logging as efficient as possible, a thread local buffer was used. If not thread-local, threads would compete for the pointer to the next position in the log and have high overhead.

Mutation logs were implemented in C. They are called from the interpreter before any store operation is executed. The JIT-compilers emit stub routines calling the mutation log write barrier before any write operation. Ideally, this would be done in assembly. But time limitations prevented me from doing so.

6.7. Permanent space

In Java there is a permanent space, dedicated for cells used internally by the virtual machine. What characterizes cells in this space is that they do not change a lot, as suggested by the name. Historically it was kept permanent; this space was not collected at all. However, with more complex applications requiring the dynamic loading and unloading of classes for instance, the permanent space is typically considered a generation by the garbage collector.

Currently this implementation does not support generational garbage collection. But the permanent space still needs special treatment, but without losing too much performance. Since the heap residency is typically low, copying all permanent objects back and forth would cost a lot; this is something to avoid if possible.

In this garbage collector, the permanent space is not collected normally. It is assumed that it does not need more than a certain chunk of memory ever. This is a safe assumption in normal programs.

Ideally a remembered set would remember intergenerational pointers from permanent space to the rest of the heap, and be maintained by the mutation log. Unfortunately this is not yet supported. Instead it traverses the permanent space without copying anything, simply marking cells as live and giving them index numbers for tarjan’s algorithm.
This indicates that the whole permanent-space works as if the cells’ pointers were roots which is more expensive compared to having inter-generational pointers.

Another observation is that the stack has to be used for the permanent objects. Since it is accessed both by the collector and the mutators concurrently, any use of pointer reversal by the garbage collector on these cells would break the mutators. Hence stack-based traversal is always used for these cells.

### 6.8. Coupling and software engineering principles

When software is small, architecture and engineering principles are less relevant, but it is believed that as software grows big, it becomes increasingly important to have a sound architecture and conform to software engineering principles to be able to cope with the exponentially increasing complexity of the software.

However, in the case of the hotspot JVM, this has been compromised in favor of increased performance. This led to a number of difficulties when developing the garbage collector.

The first problem faced was the size of the code. OpenJDK 7 was chosen as a target to integrate the GC to. It consists of multiple projects. Hotspot, one of these projects, is the implementation of the JVM. It consists of 500,000 lines of code, excluding code for testing. It was very difficult to grasp how everything fits together. Unfortunately it was also difficult to find appropriate documentation. Each project had a readme file which barely described how to compile. Their webpage had some very coarse-grained documentation.

In order to comprehend everything and to be able to work with the code, an Eclipse project of hotspot was created and used several tools like Doxygen and CppDepend.

Another problem is that hotspot is written in C++ while my GC was written in C. Hence, it had to be ported to C++ to follow their conventions, abstractions and datastructures for GCs. In the end, everything except the conceptual algorithms had to be re-implemented.
7. Evaluation

In this chapter the solution is evaluated vis-à-vis garbage collection efficiency. The benchmarks are run on a MacBook Pro with 2.53 Intel Core i5 dual core processor and 4 GB 1067 MHz DDR3 RAM.

7.1. GCBench for Java

GCBench (Boehm, HP Labs, 2000) is an old benchmark for garbage collectors. It iteratively constructs trees of various depths. The Java version of the benchmark was used to give some hints about efficiency and to compare my implementation against various state of the art garbage collectors. Table 7.1 summarizes the initial benchmark results.

In the table, FiskGC STW is my stop-the-world garbage collector, FiskGC parallel is my implementation using parallel garbage collection and Parallel Scavenge is the default garbage collector used by the OpenJDK hotspot JVM.

<table>
<thead>
<tr>
<th>Collector</th>
<th>Time elapsed (ms)</th>
<th>Heap size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FiskGC STW</td>
<td>12487</td>
<td>600</td>
</tr>
<tr>
<td>FiskGC parallel</td>
<td>11458</td>
<td>600</td>
</tr>
<tr>
<td>Parallel Scavenge</td>
<td>11674</td>
<td>600</td>
</tr>
</tbody>
</table>

Table 7.1 Benchmark results

The collector time including analysis and synchronization mechanisms for my parallel GC is 1098 ms which is 8.74% overhead.

Conclusively, at the moment, my STW garbage collector is 7% slower than the parallel scavenge garbage collector in hotspot JVM. This means the speeds comparable. The parallel version is slightly faster than the Parallel Scavenge GC, which is the default collector in OpenJDK 7. This is a strong claim and will be further analyzed in the future.
8. Conclusion

I was able to successfully integrate my garbage collection algorithm into the hotspot JVM from OpenJDK 7. It successfully collects garbage and performs dynamic purity analysis at the same time at almost no extra cost. The performance is comparable to that of the high performance state of the art mark & sweep garbage collectors used by Hotspot.

With thread local allocation buffers and thread local mutation buffers, allocation speeds were high and synchronization costs low for parallel garbage collection, achieving good performance in parallel codes. With parallel garbage collection, execution times for mutators in parallel programs are kept high.

The analysis results are available and can be used either in Java code or by the virtual machine for automatic parallelization.

Therefore, the goals of the thesis were all reached.

This thesis is a technical report written as partial fulfillment of my master degree. An accompanying paper will be published as partial fulfillment.
9. Future work

There is much potential for future work in this project. For instance, it would be good to introduce generational garbage collection to the implementation. This would increase garbage collection efficiency as the objects in permanent space would be left alone mostly, and long-lived objects would not be replicated as many times.

Another idea relating to performance is to add fine-tuned assembly for the write barrier, instead of emitting stubs to C++ code performing these operations. Furthermore, analysis could be made to emit only one write barrier per object between any two safepoints.

As for analysis, it would be good if it could find not only subgraphs of objects that are independent of the rest of the program, but pairs of subgraphs that are independent from each other. Then the potential for parallelization would be much higher.

Another interesting aspect is that of software transactional memory (STM), an increasingly popular synchronization mechanism, as an alternative to locking. It makes atomic memory transactions instead of locking. This requires logging of memory accesses and copying when committing transactions. This overhead could possibly piggyback on the replicating garbage collector instead, providing a cheap STM mechanism.
10. References


