Degree project

Automatic memory management system for automatic parallelization

Author: Erik Österlund
Date: 2011-08-10
Subject: Computer Science
Level: Bachelor
Course code: 2DV00E
Abstract

With Moore’s law coming to an end and the era of multiprocessor chips emerging, the need for ways of dealing with the essential problems with concurrency is becoming imminent. Automatic parallelization for imperative languages and pure functions in functional programming languages all try to prove independence statically. This thesis argues that independence is dynamic in nature. Static analysis for automatic parallelization has failed to do anything but trivial optimizations.

This thesis shows a new approach where dynamic analysis about the system is provided for very low costs using a garbage collector that has to go through all live cells anyway. Immutable sub-graphs of objects that cannot change state are found. Their methods become pure functions that can be parallelized. The garbage collector implemented is a kind of replicating collector. It is about three times faster than Boehm’s collector in garbage collection, fully concurrent and provides the dynamic analysis almost for free.

Keywords: garbage collection, dynamic analysis, automatic parallelization
Content

1. BACKGROUND ....................................................................................................................... 1
   1.1. INTRODUCTION ............................................................................................................... 1
   1.2. REPORT STRUCTURE ...................................................................................................... 1

2. PROBLEM AND GOAL ............................................................................................................ 2
   2.1. GOAL CRITERIA ............................................................................................................... 2
   2.2. PROBLEM ANALYSIS .................................................................................................... 3

3. MOTIVATION .......................................................................................................................... 4
   3.1. DYNAMIC FREEZING ....................................................................................................... 4
   3.2. GUESSING ....................................................................................................................... 4
   3.3. CONCLUSION .................................................................................................................... 4

4. REQUIRED KNOWLEDGE ..................................................................................................... 6
   4.1. GARBAGE COLLECTOR TERMINOLOGY ......................................................................... 6
   4.2. PURE OBJECTS ............................................................................................................. 7
   4.3. STRONGLY CONNECTED COMPONENTS (SCC) .......................................................... 7
   4.4. TARJAN’S ALGORITHM ................................................................................................. 8
   4.5. DEUTCH-SCHNORR-WAITE POINTER REVERSAL ....................................................... 8

5. RELATED WORK ..................................................................................................................... 11
   5.1. MARK AND SWEEP ........................................................................................................ 11
   5.2. COPYING COLLECTOR .................................................................................................... 11
       5.2.1. Concurrent copying collector .................................................................................. 12
       5.2.2. Lock-free solutions .................................................................................................. 12
       5.2.3. Replicating collector ............................................................................................... 12

6. SOLUTION ............................................................................................................................... 13
   6.1. THEORY .......................................................................................................................... 13
       6.1.1. Stack depth and pointer reversal ........................................................................... 13
       6.1.2. Condensing the graph ........................................................................................... 13
       6.1.3. Modification of Tarjan’s algorithm ........................................................................ 13
       6.1.4. Dealing with stack overflows ............................................................................... 14
       6.1.5. Algorithm concurrency problems ........................................................................... 15
       6.1.6. Dynamic type information .................................................................................... 16
       6.1.7. Barrier – mutation log ............................................................................................ 16
       6.1.8. Barrier and cache coherency ................................................................................. 16
       6.1.9. Accurate vs conservative collection ...................................................................... 17
   6.2. METRICS .......................................................................................................................... 17
       6.2.1. Allocation time ........................................................................................................ 17
       6.2.2. Real-time fitness ...................................................................................................... 17
       6.2.3. Concurrency & synchronization overheads ............................................................ 18
       6.2.4. Memory compactification & caching ...................................................................... 18
       6.2.5. Barrier cost ............................................................................................................ 18
       6.2.6. Collection speed .................................................................................................... 18
       6.2.7. Scalability .............................................................................................................. 18
       6.2.8. Portability ............................................................................................................... 19
   6.3. IMPLEMENTATION .......................................................................................................... 19
       6.3.1. Basic structure ....................................................................................................... 19
       6.3.2. Accurate root identification ..................................................................................... 19
       6.3.3. Safe-points and safe regions .................................................................................. 20
       6.3.4. Barrier analysis ....................................................................................................... 20
       6.3.5. Guessing mutability with our GC ....................................................................... 21
       6.3.6. Cell layout and memory optimizations ................................................................ 21

7. EVALUATION .......................................................................................................................... 23
List of figures

FIGURE 4.1 Garbage collector on its way forward.................................................................6
FIGURE 4.2 Garbage collector on its way back again .............................................................7
FIGURE 4.3 Pure objects...........................................................................................................7
FIGURE 4.4 Conceptual view of pointer reversal in a GC......................................................9
FIGURE 4.5 Actual view of pointer reversal...........................................................................10
FIGURE 6.1 Cell header...........................................................................................................21
FIGURE 7.1 Splitting loops into independent batches and batches into independent tasks.......26
List of Tables

TABLE 2.1 GOAL CRITERIA METRICS ........................................................................................................3
TABLE 7.1 LINKED LISTS BENCHMARK ....................................................................................................24
TABLE 7.2 GCbench benchmark with holes in memory .............................................................................24
TABLE 7.3 GCbench benchmark without holes in memory ..........................................................................25
TABLE 7.4 GCbench without holes with vs. without dynamic purity analysis ......................................25
TABLE 7.5 GCbench with holes in the heap with vs. without dynamic purity analysis ..........................25
TABLE 7.6 The condition and effect of generated assumptions ..............................................................26
TABLE 7.7 Results of benefits of parallelization using assumptions in Table 7.6 ....................................27
TABLE 8.1 Goal metrics ..........................................................................................................................28
1. Background

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

1.1. Introduction

Computers are getting more CPU cores, but they are not used very well. With Moore’s law potentially coming to an end (Moore, 2005), a necessity to think of ways to do things in parallel has emerged. In functional programming, ”pure” functions may be automatically parallelized, as they have no side effects, thereby using the cores better. In OOP APIs for parallelization are becoming easier and easier to use, but still people do not use them unless they are forced to, due to the inherent complexity of parallel programming. Regardless of how easy APIs become and whatever abstractions we use, people will probably avoid writing parallel code whenever it is possible and use it only when there is no other choice, as a last resort.

With serial code being transformed into concurrent code using automatic parallelization, the code preserves its simple sequential nature and allows programmers to focus their thoughts on the complexity of the system itself, rather than how to manage thread pools and concurrency for small details.

Automatic parallelization has been attempted for imperative languages, with limited success. Static analysis has been applied to code at compile-time, where statements must be proven to be independent, which unfortunately often results in relatively small gain, as very few statements can be statically proven to be independent in a typical program.

The independence of statements, however, is a dynamic landscape, changing throughout the execution of the program. Objects are mutable or immutable. Objects that are immutable are likely to not change any state. If it can be proven that an immutable object cannot in any way change state in the program, all of its methods become pure functions. Objects can then start as mutable and then be frozen when further change is no longer necessary.

While algorithms for automatic parallelization is by no means a silver bullet that will solve all our problems, it can certainly help. The programmers can figure out smart solutions for parallelization, where an algorithm might be blind in tricky cases. But on smaller scales, algorithms can keep track of smaller details for parallelization, which we programmers cannot, because the introduced complexity and costs outweighs the individual gains. By using automatic parallelization, we can utilize the power of extra CPU cores, while preserving good-looking clean serial code.

1.2. Report structure

Chapter 2 discusses the problem at hand in more detail and outlines the goal with this thesis. In Chapter 3 a motivation for dynamic analysis in the problem domain is given, outlining two approaches for dynamic immutability. Chapter 4 covers terminology and knowledge required to understand what others have done in related fields Chapter 5 and my solution Chapter 6. We finish with some benchmarks in Chapter 7, a conclusion in Chapter 8 and a few words about future work and potential research directions in Chapter 9.
2. Problem and Goal

We cannot simply assume an object’s methods will not change state indirectly in the system just because the object is immutable and cannot change its own state directly. The object could have references to other objects that may potentially be mutable and change state in the system. An object’s methods could also change state in different other ways as well such as IO-operations or through static methods, as class objects are accessible by anyone. I will call an object whose methods cannot change state “pure” objects. These are required to be immutable, and all other objects this object references must also be pure, and for simplicity, no static methods may be used. This means that these pure objects come in frozen (immutable) sub graphs in the graph of objects.

The argument I am making is that the state of programs can be very dynamic landscapes. This makes it very hard for static analysis to do anything useful. If we could master the dynamic landscape we could exploit information about it for various things, such as parallelisation.

If we can find these pure objects during runtime, a new window of opportunity for parallelization opens, where the mutability or immutability of objects can be examined dynamically as the program progresses.

The goal of this degree project is to make a garbage collector that can find immutable sub graphs of objects in memory. The most time consuming thing with garbage collection is tracing through the graph of objects that often lie at places hard to predict by caches. The hypothesis here is that we can get this dynamic analysis of what parts of the system are frozen (immutable) without losing execution time, as the objects are cached when we trace through them anyway. If we have to visit them, we might as well gather some information.

2.1. Goal criteria

The focus of this degree project is finding a good candidate algorithm for the garbage collector as well as implementing a framework for this garbage collector that may be integrated into a virtual machine.

The algorithm has to be efficient, in a parallel environment, and collect garbage in linear time. Simultaneously it shall identify pure objects, without substantial overheads, in linear time.

The correctness of the implementation must also be tested with proof of concept code, displaying results expected for the given algorithm.

If performance, portability or generality has to be chosen, performance will generally be prioritized, unless trade-offs are sufficiently small or implementation of less general variant is substantially much harder. If memory or computational overhead has to be chosen, memory will generally be sacrificed, unless overhead is too big or poor memory layout decreases performance anyway.

To make it more concrete, Table 2.1 shows a list of metrics we will look at, which we can compare against in the conclusion.

<table>
<thead>
<tr>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time fitness</td>
</tr>
<tr>
<td>Concurrency &amp; synchronization overheads</td>
</tr>
<tr>
<td>Memory compactification &amp; caching</td>
</tr>
<tr>
<td>Barrier overhead</td>
</tr>
<tr>
<td>Generational</td>
</tr>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Allocation speed</td>
</tr>
</tbody>
</table>
Portability
Portability

Accuracy
Accurately detects pure objects
Accurately collects live objects

| Table 2.1 Goal criteria metrics |

2.2. Problem analysis
The idea with the solution explained here is that the garbage collector will perform the identification while collecting garbage, as the node graph has to be traversed then anyway. This has to be very cheap as many performance issues are related to the garbage collector.

We note that we want our implementation to be constant in memory. We do not want a stack while collecting, which risks getting overflowed. Therefore we want to use some kind of pointer reversal, which will be explained in coming sections. For now we will only note that the pointer reversal algorithm requires us to modify the pointers in cells. What this means is that the obvious way of doing this would be to have a stop-the-world-collector that stops the world for a small amount of time while collecting. The motivation here is that we cannot let the collector mutate the pointers while our mutator threads tries to access the same data.

Another less obvious solution would be to have several semi-spaces and mutate the cells in one semi-space owned by the collector while letting the mutator threads access another clone of the cells. This would require some kind of synchronization mechanism between the semi-spaces which will be discussed in the following sections.
3. Motivation

As previously stated, the problem with static analysis is that it cannot keep up with the dynamic nature of programs. The mutability of an object changes over time. If this dynamic state could be queried, it could be exploited for parallelization and certainly other things as well. And since we have to trace through the objects while collecting anyway, we might as well get this information for free anyway as they are already cached and will not cost any extra time if we design our algorithm carefully.

3.1. Dynamic freezing

This advocates a style of programming where objects are frozen dynamically when they stop changing. This has several advantages. One is the dynamic automatic parallelization aspect, which has obvious advantages.

Another aspect is that the potential method for freezing may be overridden to for instance exchange internal data structures to more reading-friendly data structures, which also can have obvious advantages.

It is also an idiom in the Ruby programming language for other reasons such as making error handling easier. By freezing mutable objects when they should no longer be changed, they can be passed around in different methods, and if anyone tries to mutate the object, an exception is thrown and the error is found immediately.

Conclusively the good thing with dynamic freezing of objects is that we can also do useful things such as switching internal data structures when we know an object is no longer going to be changed. The bad thing is that programmers have to write code to do this, which many times will be omitted.

3.2. Guessing

Another approach to find out what objects are mutable or not, is to guess. If an object has not changed for a while, one could assume it is statistically improbable that it will change the coming near future. And then if they do, a roll back strategy is employed to clean up the mess made by the incorrect assumption.

This can be done by putting objects that are assumed to be immutable on memory protected pages and having barriers before IO-operations, and catching anything that might change state before it happens, and roll back. Since state was never changed, it can be re-done in a safe serial manner.

By manipulating the statistics, this could potentially lead to increased net performance if enough assumptions are correct to allow higher utilization of the extra CPU cores that today are ever so passive. As a matter of fact, not everything would go to waste if assumptions are false. The worst thing that can happen is that the extra utilization of another core would vanish. But otherwise it would not have done anything useful anyway. So when our assumptions turn out to be wrong, it does not matter much. When they are right, on the other hand, we utilize more CPU cores and speed up the application.

Speaking in a general sense, this guessing mechanism is very useful whenever mutations in heap objects are clustered in time. So whenever there are distinct phases when objects are not mutated, automatic parallelization becomes possible.

3.3. Conclusion

Conclusively the guessing alternative is better in the sense that programmers do not have to write any code and many more opportunities for automatic parallelization will be found. The bad thing with is that when we have incorrect assumptions we have to do some kind of rollback, which could cost a bit depending on implementation.
guessing and freezing could be used hand-in-hand and they are not by any means mutually exclusive. However, the focus will be on guessing, as its pros are more interesting.
4. Required knowledge

In this chapter required knowledge to understand the coming two chapters is given. It covers garbage collector terminology and some new terminology invented by the author to understand the problem. Additionally, strongly connected components and Tarjan’s algorithm are discussed as knowledge about this is key to formulating how pure objects can be found.

4.1. Garbage collector terminology

We shall call each node in the object graph a cell. Each cell can be color-coded. The traditional colors are white, grey and black. A white cell is a cell that has not yet been visited by the collector. A grey cell is a cell that has been visited, but is not done yet as its immediate children have not been visited yet. A black cell is a cell that is done, as far as garbage collection is concerned. It has been visited and so has its immediate children.

We shall also define a new color, completely black. A cell that is completely black is a cell whose full sub graph has been visited by the collector. A completely black cell is finished both with collecting and the identification of pureness.

A root cell is a cell referenced from the stack. A live cell is a cell that can be accessed by following some path from a root cell. Cells that are not live are garbage, and their memory should be recycled.

Figure 4.1 shows the color codes mentioned in this hypothetical example where the garbage collector has gone from Root #1 to the arrow indicating where the garbage collector is currently looking. In this figure, the garbage is on its way forward. In Figure 4.2 we see the garbage collector passing through the same node again but on its way back again. Now it knows all transitively reachable nodes have been visited and hence the sub-graph is colored completely black as it goes back.

![Figure 4.1 Garbage collector on its way forward](image)
4.2. Pure objects
A pure object is an object which cannot by any means change state in a program. This means that it has to be immutable, and its whole sub-graph of objects also has to be immutable. Additionally, it may not access global variables or use I/O-operations. As far as our garbage collector is concerned, the task at hand is identifying sub-graphs of immutable objects while collecting garbage. For the sake of simplicity we shall therefore assume that a pure object, is a memory cell that is immutable and whose whole sub-graph is immutable. A cell that is dirty is a cell that cannot be pure. Either it is mutable or a cell in its sub graph is mutable. In Figure 4.3, the white objects are immutable but only the happy objects are considered pure as they are the only ones that can not reach any of the mutable (yellow) objects.

4.3. Strongly Connected Components (SCC)
A graph can be partitioned into its strongly connected components. Each strongly connected component is a set of the nodes in the graph such that each node can be reached from any other node in the strongly connected component by following their directed edges. In our case the nodes are memory cells and the edges are references. In
In Figure 4.3, the happy objects form an SCC together. The lone white immutable object is its own SCC and the yellow mutable objects are also their own SCCs.

4.4. Tarjan’s algorithm
With Tarjan’s algorithm (Tarjan, 1972), we can identify the strongly connected components in a quicker way. The strongly connected components can be identified with a single pass. In the typical case of Tarjan’s algorithm, we take the nodes and edges as input, and produce a set of strongly connected components as output. This is done with a depth-first traversal (Cormen, Leiserson, Rivest, & Stein, 2001). Each node gets an index representing when it was visited by the depth-first traversal. So the first node would be one, next one would have index two and so on. Each node also has a lowest index which represents what the lowest index found by the algorithm is. This lowest index is initialized to the index of the node as it is first discovered. This property is then propagated backwards by assigning the minimum of the lowest index for the child we came to and the parent we go back to, to the parent’s minimum index.

In case of a cycle, a node will be visited which has a lower index, as it is already visited. This property is then propagated backwards. The lowest index represents the “root” of the strongly connected component. Of course, normally a strongly connected component would not have one single root node, but in Tarjan’s algorithm, it has, and it is the first node in the strongly connected component to be seen by the depth-first traversal.

Tarjan’s algorithm also has a stack to keep track of what nodes should be part of a strongly connected component. When we enter a node we push it to the stack, and when we go back from a root node of a strongly connected component, we pop all nodes belonging to the strongly connected component and output it to its SCC.

4.5. Deutsch-Schnorr-Waite pointer reversal
Pointer reversal is a technique that can be employed in order to get rid of stacks when traversing graphs in depth-first order. It was found by (Schnorr & Waite, 1967) and Deutsch (Knuth, 1973). The basic idea is that whenever we follow a pointer from a cell A to another cell B, we replace this pointer by the parent of A. This will effectively create a path for us back from where we came, as we go forward. And when we go backward, we restore the pointers to their original values.

We can consider this algorithm as a state machine with basically 3 states. One state is to go forward, where we have cell A pointing to cell B, and we replace the pointer in A to B to point to the parent of A instead, and move on to B and make it the current cell.

The other state is when we retreat and go back when we can not go further. This is where we restore pointers to their original values. We do this until we find a grey cell (a cell which has not had all of its children examined), or until we get to the root and we are done.

The third state is when we switch child. So if we have cell A pointing to B and C. Say we have examined B and are on our way back, but C has yet to be discovered, then this switch state will take the back-pointer from A to B that actually points to the parent of A, and put this in the pointer from A to C instead, and then restore the pointer between A to B to actually point to B. Then we make C the current cell.

In the case where we have two semi-spaces we have to consider what cells we modify. We may only modify replicas of the objects in from-space, which reside in to-space. So whenever we go forward from cell A to B and B is in from-space, we create a new replica in to-space and store a pointer from from-space B to to-space B. We are now free to mutate the pointers in the replica of B in to-space. The same reasoning goes
from the switch phase. When we switch we also encounter new cells, and thus have to make replicas and mutate only pointers in replicas.

In Figure 4.4 we see the conceptual view of pointer reversal in a garbage collector. The color coding of the cells is as previously described. The directed blue edges are the actual edges of the cells. The red edges have inverted direction, effectively creating a path back from wherever the garbage collector is. In Figure 4.5 we see the actual view of this algorithm executed as described in the text as a state machine. Here some of the edges are not shown because they are stored in the local state that remembers the previous, current and next cell.

Figure 4.4 Conceptual view of pointer reversal in a GC
Figure 4.5 Actual view of pointer reversal
5. Related work

This chapter describes related work in the field of garbage collection. It mentions two of the three most used general approaches – mark and sweep collectors and copying collectors. The third type, reference counting collector was left out because it is not useful in any sense for our purposes.

5.1. Mark and sweep

Mark and sweep collectors such as the famous Boehm-Demers-Weiser collector (Boehm & Weiser, Garbage collection in an uncooperative environment, 1988) have 2 distinct phases as the name implies. The first phase is the mark-phase. Here it will trace through the live objects and mark them as live. When this is done, the sweep phase starts. It will sweep through the whole heap and add cells that are no longer used to a free-list, which is maintained by the collector.

Scanning through the whole heap including dead space is time consuming. It should be noted however that scanning linearly through the heap is faster than tracing as the pattern is predictable for caches. Also techniques can be employed to get rid of the costly sweeping. We can use lazy sweeping. With this technique, sweeping is done when allocating storage. If the heap consists of primarily dead space, which is unmarked, it will quickly find a free spot to allocate in.

Another problem with mark and sweep is fragmentation. As it looks for holes of free memory, the standard problems of fragmentation occur. We need to determine what holes we are looking for – best fit, first fit or worst fit. Also fragmentation leads to another obvious problem – memory is wasted.

In order to solve fragmentation problems, a technique can be employed where each data type with a certain size has its own place in the heap. The sub-heap can then be divided into cells of constant size, and no fragmentation will occur. A common technique is to have pages maintaining cells of sizes of powers of 2. If wanted cells are not a power of 2, the power of 2 above will be chosen. Ironically, this would result in internal fragmentation equivalent to the memory loss of a copying collector. However, pages can be kept for each data type, and there will be no fragmentation.

Yet another problem with mark and sweep is that even though we can get around fragmentation, the locality of the memory layout will get worse and worse as the mutator progresses. The caching performance will become horrible. The way to get around this is to use a so-called mark compact collector. This type of collector would use a strategy compactify the heap while sweeping. This however will come at a cost. Since mark and sweep uses only one heap, the mutators must be stopped while compactifying the memory, which could cause more overhead than we gain performance. This is especially the case in an environment where we have several mutators that all have to wait for the compactification.

Finally, all mark and sweep solutions were eliminated because they could not use pointer reversal with concurrent collection (without unreasonable memory overheads) because only one heap is used. It is possible to use a stack instead of pointer reversal, but we decided we do not want to do that because it is less consistent in its behavior.

5.2. Copying collector

Copying collectors such as he one made by (Cheney, 1970) divide the heap into two semi-spaces called from-space and to-space. The mutator will fill from-space from the bottom with newly allocated objects. A key advantage with using copying collectors is that the allocation time is minimal as it only has to increment a pointer. When the heap is filled, garbage collection starts. What happens is that live objects in from-space are
moved to to-space. In the simplest case with a stop-the-world-collector, the mutator execution would be suspended and all live objects would be evacuated to to-space, and then the mutator would be started again.

The advantages with copying collectors are that allocation is very efficient; there is no fragmentation as memory is contiguous, and the locality will be good as memory is compactified automatically in the process of evacuating live objects. Also it has one single phase compared to the 2 phases for mark and sweep. This single phase also traces through only the live portion of the heap, which generally is very small compared to the whole heap.

The disadvantage is the memory requirements. It requires twice the amount of memory for the heap as it has two semi-spaces.

5.2.1. Concurrent copying collector

A concurrent copying collector such as the standard one made by (Baker, 1978) will allow the mutator to continue while collection is going on. When garbage collection starts, the mutator continues allocating memory at the end of to-space while the collector evacuates live objects to the beginning of to-space. Each cell in from-space will typically have a pointer to its corresponding copy in to-space. When copying is done, the sides are flipped and roots are replaced with the evacuated cells. A read-barrier is generally used, which will evacuate the read cell and its immediate cells, rendering it black and the children grey. With this kind of approach, like Baker’s algorithm (which is actually incremental but same principle holds), the mutator will never see from-space. It will only see to-space.

However, since the barrier moves cells, and the mutator also lives in to-space, we cannot really use pointer reversal with this approach either.

5.2.2. Lock-free solutions

Lock-free solutions were considered. In general they the overheads of using them were not reasonable. One solution that was attempted for copying collectors would copy whole objects on every change and link the different versions together with linked lists, and having complex handshaking protocols to communicate between threads if any versions were in from-space or to-space in order to determine if a small piece of to-space is ready to flip or not. The versions would not be necessary of a compare and swap instruction for two variables could be used. However this is not accessible in any conventional hardware. The solution was considered not practical.

Another solution for mark and sweep would let only a single mutator thread own a cell. If another mutator wished to change it, the cell would have to be mailed over to that thread with a mailbox and a handshaking protocol, and then be given back once it has changed. Additionally, a billion extra fields would have to be stored in each cell. The solution was also considered impractical.

5.2.3. Replicating collector

Replicating garbage collectors such as the one described by (Nettles & O'Toole, 1993) are quite similar to normal copying collectors such as Baker’s but have their distinct differences. The collector has two semi-spaces, like a normal copying collector. But instead of letting the mutators see only to-space while collecting, using a read-barrier, such as described in Baker’s algorithm, we instead use a write-barrier and let mutators see only from-space while collecting. In order to keep from-space synchronized with to-space, a replicating collector stores a mutation log. The mutation log keeps track of what changed in a cell. The collector reads this mutation log and adjusts the to-space to be the same as from-space, while collecting.
6. Solution

This chapter describes the proposed solution on a conceptual level to solve the problem of getting purity analysis dynamically, starting at section 6.1. It describes the type of garbage collector to be used, and a few algorithms and algorithm tricks embedded into the garbage collector to get promising results. It also includes some analysis of the efficiency of the garbage collector with different metrics in section 6.2. Section 6.3 includes some important insight in the implementation domain of the system.

6.1. Theory

We have chosen to use a replicating collector for various reasons. There were three primary reasons.

First of all we want to have a concurrent garbage collector that does not decrease the performance of our mutator thread. This also allows us to collect garbage more frequently even if we don not have to when only one thread is executing, just to get some analysis of the system and see if we can speed things up. And we do this without slowing the mutator down.

Secondly, Tarjan’s algorithm becomes more straightforward if we can guarantee cells are copied in true depth-first order. Because then the index telling in which order they are traversed can be replaced by their actual memory address. Normal copying collectors break this order with their read barriers. We get to the details of this algorithm later.

Thirdly we do not want to have a stack that overflows with deep data structures. Therefore we want to use pointer reversal. And given that we have concurrent collection, we need another semi space where we can do the pointer reversal without interfering with the mutator threads.

6.1.1. Stack depth and pointer reversal

The problem with a depth-first algorithm that follows cycles, is that our stack can grow very big, which causes problems if a stack overflow occurs. We can prevent this by not using a stack. Instead we can use pointer reversal which will allow our algorithm to run in constant space. What this means is that whenever we follow a reference, we replace that reference with a reference to the previously visited cell. Thus we can track what path we took in the graph and go back. When we go back we restore the original references. Note that this cannot be done with reasonable efficiency while multiple mutators have direct access to the cells as the references are replaced. We solve that later on.

6.1.2. Condensing the graph

If any cell in a strongly connected component is dirty, then all other cells in the strongly connected component have to be dirty too. An easier approach to use when finding the pure objects is to condense the graph by identifying the strongly connected components, and then propagate the dirty property in this condensed graph instead. What we would do is to propagate the property backwards in the opposite direction a reference is pointing, meaning that we would do a depth-first traversal of the condensed graph and propagate the property back as we leave cells and go back. This would require several passes to be done.

6.1.3. Modification of Tarjan’s algorithm

Tarjan’s algorithm will output the SCCs of a graph. However, we are not interested in using these SCCs for anything else than to determine if a cell is pure or not. It turns out
we do not have to allocate SCC nodes. It suffices to know what cell is the root as we shall see.

It should be noted now that not everything in this algorithm is necessary for our purposes. First of all, we do not have to store an index for each node, because the address already serves the same purpose. Since we have a type of copying collector that copies cells in depth-first order, and the collector alone is allowed to do this (no read-barrier copying as normal copying collectors), the address will increase in depth-first order, just as a potential index would have done. Note that a replicating collector is the only type of collector that can guarantee that the cells are laid out in depth-first order and remain concurrent. Mark and sweep cannot be used because it does not move objects. Copying collectors have read-barriers that copy objects over and hence violating depth-first order of memory addresses.

Secondly, we do not need the stack either. We are not really interested in looking at what our SCCs are. All we want to know is if a cell is pure or not. Remember that if any cell in the SCC is dirty, so is every other cell. In our algorithm, if the root cell of a SCC is dirty, so is every other cell in the SCC. So what we do is to propagate the dirty property backwards in the depth-first traversal of the graph. And instead of looking if a cell is dirty or not by looking at that particular cell, we introduce one level of indirection; we check if its root cell in its SCC is dirty instead. If a cell is immutable and its root is not dirty, we know it is pure.

We still need to be able to determine if a cell is in the stack or not. When we go forward to a cell that has been visited by the collector, we must be able to determine if it is in the stack or not. If it is in the stack, they belong to the same SCC, and if it is not, they do not belong to the same SCC. This can be solved by using one bit for a flag. When we enter a cell we set the flag bit to true indicating that the cell is grey and when we exit the cell we set it to false to indicate it is completely black. In order to check if a visited cell is in the stack, we check if the cell’s root is grey or black. If the cell’s root is completely black, it is a different SCC and is not in the stack. If the cell’s root is not completely black, we know it is in the stack and that it belongs to the same SCC as the cell we came from.

Conclusively we can employ a modified version of Tarjan’s algorithm without an index or a stack and get the wanted result. The overhead we do need, however, is the lowest index, which would be a pointer to the root cell in the strongly connected component.

6.1.4. Dealing with stack overflows

A normal copying collector typically traverses the graph in depth-first order because the heap can then be used as a stack. The good thing is that we do not need a stack. The bad things are that we get worse caching performance for the mutator threads and Tarjan’s algorithm cannot be used. Conclusively we want to use depth-first traversal. Hence we might run into problems with stack overflows. We have the choice of using a stack or using pointer reversal.

Pointer reversal is not by any means the one true solution to our problem of avoiding stack overflows. The good thing is that it is constant in memory. The alternative, however, is using a stack instead.

The obvious problem with stacks is that we can get stack overflow when we run out of memory in the stack. This can be solved with different approaches.

One approach is to adaptively change the size of the stack. Whenever it runs out of memory, we allocate new memory with more memory available, say twice the size, and copy the old data over, and then free the old data. This solves the problem of stack overflow making our algorithm ineffective. However it has other issues – the memory
requirement is not bounded. This stack could grow huge and get out of control, depending on how the heap looks. This would have to keep pages in secondary memory storage, and render the algorithm inefficient.

Another approach in handling stack overflows is to start over from the beginning of the stack whenever we get an overflow. This will cause our stack to only remember the end of the stack, and not the beginning. We then collect in multiple passes instead. So we start by collecting parts of the heap, probably leaves and neighbors in the first pass, and then in the second we work ourselves toward the roots. So the size of the stack will approximately determine how many cells we can collect per pass. The advantage of this solution is that it is constant in memory, but the obvious disadvantage is that it would require multiple passes when graphs are too deep.

When the stack is big enough, these solutions will be very effective as the memory layouts of stacks are very predictable and work well with our caches. On the other hand, when the stack is not big enough, the results become unpredictable. Questions such as “how big should the stack be?” will arise. Therefore we have decided to go for pointer reversal instead. Here the results are predictable, and it is constant in memory.

What could be done is to put the stack in to-space. Because a very small portion of the objects in from-space survives, the rest of the memory in to-space will not be used while collecting. A stack could be created from the bottom, growing upward, while relocating live objects to the top, growing downward. Since only a small fraction of all cells are live and survive the collection, the amount of free memory in to-space is big and we would generally be able to put a stack at the bottom without problems. This is also good, because if we use pointer reversal and 2% of the objects survive garbage collection, that means that the extra field in the headers of the other 98% of the objects are just wasted memory. Because these fields are needed only when we traverse through them while collecting garbage. So by using a stack, we waste memory only for the live objects that we actually traverse while the other 98% of the cells remain smaller.

In the end we decided to use pointer reversal because it has a guarantee that we will not run into stack overflows. And we do not want to hope that sufficiently many cells die and have no strategy except crashing if that is not the case. And we do not want to re-traverse the graph either. With pointer reversal everything just works under all circumstances and the pay-off is log(n) bits of memory in the headers where n is the number of children for a cell.

6.1.5. Algorithm concurrency problems

When we traverse the graph of cells, they were marked as completely black whenever we stepped back from them. However, if a cell in from-space was changed by the mutator and we see this in the mutation log, we have to change this in the to-space as well. When this cell is changed, all parents are incorrectly colored completely black, while in fact, they are not, as the mutated cell is not done tracing yet. It should be remembered though that the color completely black is important only for finding pure objects. The pure objects will be correctly identified in all cases where the completely black color is broken, because the only place where it would possible be a problem is when an immutable object mutates to point to a dirty object.

However, since it is immutable, this is not allowed by definition, and thus we do not have to be afraid of this. We just want to note that we cannot trust non-immutable objects to be in the correct SCC with the concurrent modifications going on. For mutable objects they can be marked as being in another dirty SCC, but not in a pure SCC. And for our purposes this is allowed and does not cause any problems.
6.1.6. Dynamic type information

We require that only objects are stored in our cells, that have a dynamic type. The dynamic type must contain certain data for our collector to work properly. We have to know the size of our cells and at what offsets we have pointer fields, in order to do accurate pointer identification. In our implementation we have chosen to store this data in the dynamic type data structure. So each dynamic type, which is a class object, will store the pointer offsets and cell size. This means that types with dynamic size, such as an array, will have one class-object for each specific size. This is not a constraint by our type of collector, but merely a choice made to keep it simple. Instead of storing the size, we could store a function pointer and dynamically fetch the size of cells. But this is not what we are interested in for this thesis, so we will keep it simple.

6.1.7. Barrier – mutation log

With a replicating collector as described by (Nettles & O'Toole, 1993), we have two semi-spaces. One of the semi-spaces is used by the collector thread and one for the mutator threads. In order to keep to-space that the collector works with in sync with from-space, some synchronization mechanism is required. A replicating collector uses a mutation log for this. This is a buffer where pointers are written to objects that are mutated in from-space. This is a kind of write-barrier. Whenever a mutator mutates a heap cell, it writes the pointer to the mutation log. This is then read by the collector. The collector then has to copy this cell over to to-space again, and follow pointers and see if anything else has changed.

6.1.8. Barrier and cache coherency

If we were to assume that we were to write only what changed in a heap object instead of writing a pointer to the whole heap object, we might run into problems as pointed out by (Azagury, Kolodner, & Petrank, 1998).

We cannot write both the mutation log entry and modify the cell at exactly the same time. This leads to a problematic race condition when two mutator threads attempt to mutate the same field in the same cell at the same time. What the mutation log says might be inconsistent with what the actual cell in from-space says, and the collector will change to-space incorrectly. We got around this problem by string pointers to the whole heap cell instead and copying all of it. In the future a mechanism for handling large objects better might be needed.

However there is another problem. If the collector is running on a separate core from the mutator, its faster cache memories might not have propagated down to whatever level of cache the cores might share. This means that if the mutator changes and writes in the log what cell changed, and the collector reads the mutation log and reads the cell before the caches synchronized. We run into trouble, because the old data is read.

In order to get around this, we can re-arrange the mutation log. Instead of having one global mutation log, we let each thread of execution have its own mutation buffer where it fills in mutation entries. When it is filled, it is appended to a globally linked list of mutation buffers that together form the mutation log. Right before adding it to the end of the list, the mutator will call a machine-dependent instruction to flush/synchronize caches. The overhead of this synchronization is insignificant as it is inversely proportional to the buffer size, which can be made arbitrarily big.

With certain modern hardware, we are allowed to be more naïve. Some processor architectures have different mechanisms that make sure caches are always synchronized by using different protocols of communication between them. Some architectures allows us to keep the data safe by using memory fences before and after important
assignments such as writing to the mutation log. We empirically found out this is very expensive.

The x86 architecture also has a fetch and add instruction called xadd. It will increment the value and store the old value. This allows us to go back to having one single global mutation log, atomically incremented with xadd surrounded by memory fences. We do however still record only what cell changed, and not its new content, as we would otherwise still run into inconsistencies when several mutators write to the same location in a cell at once.

Having a mutation buffer per thread is more memory wasteful but improves caching performance with more linear memory accesses and requires no synchronization costs for xadd and mfence instructions. As for our solution, we used one mutation log without synchronization mechanism if there is only one mutator thread, which is the case in our benchmarks. Otherwise we use the xadd and mfence instruction solution due to its simplicity. We have however implemented both solutions.

6.1.9. Accurate vs conservative collection

A conservative collector does not have to accurately identify root pointers. It can scan through the stack and guess what seems to be a pointer and what does not. The probability of inaccuracy is normally low with modern 64-bit processors where the heap address space is an insignificant part of the 64-bit addresses. However, since it cannot accurately identify pointers, this technique works only for non-moving collectors such as mark and sweep or the treadmill.

Since we have chosen to use a replicating collector, which is a type of moving collector, accurate collection has to be used. This means that it must be possible to accurately identify root pointers when collecting garbage. Root pointers reside in global variables, the stack and registers.

6.2. Metrics

Here we will evaluate some properties of how good the candidate algorithm is at different things.

6.2.1. Allocation time

Allocation is very fast. We simply use the xadd instruction (exchange and add) to increment the pointer to free space. This is very cheap compared to any reasonably efficient mark and sweep implementation. It should be noted however that we also need a memory fence before and after the xadd operation which can take some additional time with the extra synchronization overhead. This can be eliminated completely by having thread-local allocation-spaces.

6.2.2. Real-time fitness

Replicating collectors are one of the better solutions for real-time applications. Arguing my solution is hard real-time is not my intention and beyond the scope of this thesis. It can however be noted that it suits well in a soft real-time environment. The collection is done concurrently, and the only overhead on the mutators is the writing in the mutation log, and the time required for that is small and bounded. It does have to freeze though when a collection cycle is starts or ends to flip sides and replace all root cells with the corresponding cells on the other side. This takes a very small amount of time to do, but makes it difficult to prove it works in a hard real-time environment.
6.2.3. Concurrency & synchronization overheads

One advantage with the replicating garbage collectors is that they are competitive in having low synchronization costs, which is good for our purposes. The only synchronization costs are 1) writing an entry in the local mutation buffer (without competition), and 2) appending the mutation buffers to the end of the mutation log (which can be made arbitrarily insignificant by varying the buffer size). An alternative solution is using the fetch and add instruction with memory fences, which is also relatively cheap.

6.2.4. Memory compactification & caching

It is fairly important to compactify the memory and improve the locality of our memory. A miss in a cache page can cause severe overhead to our application. We note that for our purposes it is more important to employ strategies for increasing caching performance for our mutators, compared to increasing caching performance for our collectors. The reason for this is that we expect to function in an environment with many mutator threads on many CPU cores, working concurrently. As the time spent collecting will become relatively little, it becomes increasingly important to lay out the memory in a way that fits our mutator threads best.

This solution arguably performs very well in this regard as it collects with a depth-first traversal, which has better caching performance than breadth-first traversal. Additionally, the order could be damaged by an ordinary copying collector which would copy cells to the other side with its read-barrier. We let the collector do all the copying so that the lay-out of the memory remains good locality.

We do not suffer from any fragmentation problems which is normal for mark and sweep collectors, since we use two semi-spaces and copy over live objects only and keeping the rest free. A mark and sweep collector would typically have a very hard time compactifying the memory layout, and would not be able to do so without stopping and letting the mutator threads wait, which is not desirable.

6.2.5. Barrier cost

The cost of the barrier is relatively small, as write-barriers are smaller than read-barriers, and the work done by our barrier is very small. However, we do need to keep track of all changes in cells, and not only changes to references, as would be the case with a generational collector or a mark and sweep collector. It should be noted however that since we use accurate garbage collection and have safe-points and safe regions, we can use one single barrier for a long series of mutations, so not each and every mutation has to be stored. Depending on the policy for the safe-points, the overhead for the write-barrier could be made potentially smaller than the write-barrier of a conservative mark and sweep collector.

6.2.6. Collection speed

The collection is done in linear time with respect to the number of cells and references in the graph and cells are visited the same number of times, compared to normal collection, so we should perform pretty well. With our mutation logs helping in finding inter-generational pointer changes for free, we can also employ generational collection to further push down the collection speed. It is also constant in memory as we use pointer reversal.

6.2.7. Scalability

As the synchronization costs are low and the only place the system has to be frozen is when flipping sides, the system should scale fairly well with large heaps. Also the
synchronization costs are very low, so it should scale well with an increasing amount of
mutator threads and CPU cores. However, having one collector thread with 1000
mutator threads is not ideal. Fortunately our solution can be done incrementally as well,
or additional collector threads can be started.

6.2.8. Portability

This collector does depend on atomic machine operations, rendering it platform
dependent. Fortunately these are standard operations available for most hardware. Our
target platform in our implementation will be x86 and x86_64, since this architecture
dominates the PC market at the time of writing this.

We need to be able to atomically fetch and add in the mutation log when advancing.
X86 has an xadd operation which does this for us. Many more platforms have this. The
ideal platform has this as our mutation log becomes wait-free and very small and cheap.
Should a platform not have such an operation, compare and swap or load link/store
conditional can be employed in its place. This will be a lock-free but not wait-free
solution.

Another machine-dependent property we rely on is cache coherency. We have used
memory fences to make sure CPU cores are up to date before and after fetch and add.
Both the fetch and add instruction and the memory fences can be avoided by having
thread-local mutation logs and thread-local allocation spaces. For this particular
implementation, this approach was not chosen.

6.3. Implementation

We have implemented a proof of concept garbage collector using the replicating
approach described, with pointer reversal and the special implementation of Tarjan’s
algorithm. A small runtime system was also created to make it possible to test how
efficient our algorithm is compared to other garbage collectors. The implementation is a
GC framework that can be integrated into a virtual machine, which has yet to be done.

6.3.1. Basic structure

Our garbage collects garbage until a fixed portion of the heap has been filled. Then it
will wake up the collector thread. The first thing the collector thread will do is to freeze
the mutator threads to get samples of their roots. Then it resumes their execution, and
starts collecting garbage with our tarjan and pointer reversal algorithm that replicates
objects to to-space. When the first collection cycle is done, we freeze the mutator
threads again, read the mutation logs and copy mutated objects over again. Then we
take a new sample of the roots and collect again for whatever new objects we might
encounter. When this is done, we switch sides from from-space and to-space, resume
mutator execution and have successfully finished a collection cycle. The mutator
threads are running most of the time and are frozen only when taking samples of their
roots and when collecting the last objects before switching sides. It might also be
possible to fetch roots without freezing for the intermediate steps but that is not our
focus.

6.3.2. Accurate root identification

When a mutator thread terminates we can easily identify the roots of the current stack
frame if we are the compiler. Because we know what function the stack frame belongs
to at the safe-point. In order to identify the stack roots in previous stack frames, we
simply follow return pointers and see what function that return pointer points to, and
continue scanning for roots in this, for the compiler, known stack-frame. We can also
tag stack frames to have kind of an isa-pointer identifying the stack frame, like a heap object.

Some compiler architectures such as LLVM (Low Level Virtual Machine) do this for us. The only thing required is that when pointers are emitted, we inform the compiler that this is a pointer so that it knows this should belong to the garbage collector root set in the stack.

Since the purpose of our implementation is to make a garbage collection framework, and not a compiler infrastructure, nor a compiler, we have not created such a mechanism for accurately identifying stack roots. For the sake of simplicity, the root set is maintained in a separate stack. This obviously is not ideal, but it suits our purposes.

6.3.3. Safe-points and safe regions

While we have a concurrent garbage collector, we only allow switching sides of the semi-spaces to occur at safe points. A safe-point will check a handshake bit from the collector that says we should freeze, and then freeze so the collector can get the roots or switch sides. However, sometimes we might have a blocking I/O system call or something and we could not just let the whole system freeze until we get a response. So there we have safe regions instead. A safe region will start before the I/O system call and end after it and the blocking is over. When the safe region begins, we tell the collector that we are practically down and it can take roots or whatever it wants to. After the I/O system call, we check if we are supposed to be alive or dead, and act accordingly. If the thread is expected to be sleeping, we suspend its execution.

6.3.4. Barrier analysis

The overhead of our write barrier would normally be greater than that of some mark and sweep write barrier, since we have to log all mutations, not only references. But it should also cost less than a copying collector’s read barrier that traps to the operating system and copies objects. The question then is if it is worth it. It turns out that many good things can be combined to use the same overhead, the same barrier.

1) The mutation log can serve to keep track of inter-generational pointers and allow us to use generational collection as well, which can speed up the collection time.
2) Having two semi-spaces separated, letting the collector work on one of them and the mutators on another, keeping a mutation log for synchronization, allows us to traverse the cells in the way we wish to using pointer reversal. The memory overhead for using this is nothing, since we need a non-destructive reference from cells in from-space to to-space anyway, and the same field can be used to keep track of where in the traversal a given cell is.
3) Using a (locally stored) mutation log for synchronization will keep the synchronization costs very low, compared to the competition, which is crucial for our purposes.
4) The time our barrier takes is bounded, making it a very suitable candidate for soft real-time applications.
5) Keeping track of changes to cells provides us with invaluable analysis for free about whether cells have changed or not. This information can help us analyze how the stateness of the dynamic landscape of our application changes. It allows us to guess what cells will not change in the future, based on what did not change in the past. We can then dynamically freeze good candidate cells.
6) Since we use accurate garbage collection and have safe-points and safe regions, we are fine as long as we invoke the write-barrier once before a safe-point or a safe region. So we can emit one write-barrier for a long series of mutations on a
cell. Depending on the strategy for emitting safe-points and safe regions, the 
cost of the write-barrier could be made very low, as the cost of the safe-points 
are normally 1% of mutator performance, and we never have more than at most 
1 write-barrier for each safe-point, which is very cheap.

6.3.5. Guessing mutability with our GC

As discussed previously, it could be very beneficial to keep track of what objects 
change in the system, so we can guess about their coming near future and whether they 
will change or not, and identify when objects’ mutability/immutability are clustered in 
time.

Interestingly enough, since we are using a replicating garbage collector, we would 
have to keep track of mutations anyway. So we get this analysis for free. For each 
collection cycle, we can determine if an object has been mutated or not. Therefore we 
can easily keep track of how many cycles it was since it got mutated. With this 
information we can determine how likely it is that it will change during the coming 
cycle.

6.3.6. Cell layout and memory optimizations

Our memory cells will consist of a header with meta-data for our garbage collector, and 
then with the payload, which will be memory for objects. The objects will also have a 
“header” containing the isa-pointer with a pointer to the dynamic type of the object. 
Since we have control over the run-time in our virtual machine, we will for simplicity 
include the isa-pointer in our cell header in this discussion to make it easier, having only 
one header to speak of.

In our proof of concept implementation, we have chosen a memory wasteful layout 
of cell headers. It looks like Figure 6.1 describes.

<table>
<thead>
<tr>
<th>replica</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags and child counter</td>
</tr>
<tr>
<td>lowest cell</td>
</tr>
<tr>
<td>isa</td>
</tr>
</tbody>
</table>

**Figure 6.1 Cell header**

The replica is a pointer to the corresponding to-space object and is used only from 
from-space to to-space. And among the cells in from-space, only the live ones, often 
about 2%, will use this. In other words, 2% of one of the semi spaces actually use this. 
The others are wasted space. This could be improved by removing the replica field from 
the headers, and putting this information in a hash table instead. This would make the 
collector slightly more memory efficient by becoming slightly slower.

The same goes for any header field that is needed only during collection. These 
fields are used only for the small amount of live objects, so storing them in hash tables 
might be more space efficient. This would remove all fields but isa at the cost of 
collection time.

Other optimizations are also possible, such as analyzing what information is needed 
when. Some of the fields are required only a very small amount of time, and are used 
for a very small amount of cells. Sometimes the cells are used on only one semi-space. If 
we for instance put objects that are found out to be pure on separate (write protected) 
pages, the mutator threads would no longer need the flags or lowest cell fields. And the 
collector would not need the isa-field, so it could be used for storing the lowest cell.

And apart from that, if we used the extra space in to-space that is otherwise useless 
during collection as a stack instead of using pointer reversal, the child counter and flags
could also be removed. But this would require some mechanism to take care of the mess if the amount of live objects get really big and the relocated objects collide with the stack.

So theoretically we have a few options of implementations where we only have the isa-field required by the runtime system in the header. We did however choose to use the memory-wasteful approach as memory and time are easily traded anyway with a GC. It will not necessarily take more space – it will just have to collect slightly more frequently. And to get rid of the wasted space we would have to waste time. And would we gain anything? We do not know and we did not investigate it further. Toying around with small memory optimizations is not the goal with this degree project. But it is worth mentioning that we can get rid of that extra space if we want to.
7. Evaluation

This chapter covers benchmarks used to test the efficiency of the garbage collector as well as hypothetical tests on the potential in JIT-parallelization using this analysis. In section 7.1 an introduction to the benchmarks is provided. Section 7.2 and 7.3 describe benchmarks testing efficiency of garbage collection. Section 7.4 compares the efficiency of collection with and without the dynamic analysis. Section 7.5 covers the hypothetical benefits.

7.1. Introduction

In order to test how good our implementation is, we created a few benchmarks to get some numbers. Now it should be noted that numbers can be deceiving and that no single program is expected to run as these benchmarks. But they are useful nevertheless to get an idea that we are not totally off.

First off we compare our garbage collector to the well-known Boehm-Demers-Weiser collector (Boehm & Weiser, Garbage collection in an uncooperative environment, 1988). This is a conservative mark and sweep collector. We chose it because it is well known and easy to test against. It has an adaptive heap size, which we do not yet. So in order to keep things fair, we fixed both heaps to 512 MB. All tests were run on my machine with 2.93 GHz core 2 duo and 4 GB 1067 MHz DDR3 memory.

I will have tables where algorithms are compared. Running time means how long it takes for the program to finish. For stop-and-go collectors this includes garbage collection and for concurrent collectors such as ours, it is almost excluded. Collector time shows how much time the collector thread required and within parenthesis this shows how long collection time / (running time + collection time) is. This is a percentage of all the time spent by the program. Collection cycles show how many times the GC was invoked. In our case 2 collections are done for each collection cycle. So a collection cycle means in our case swapping of to-space and from-space. But as we sample roots twice, we collect twice per cycle, so the values can be multiplied by 2 for our collector. The second collection is, however, much smaller than the first.

It should also be noted that no single program is expected to behave as these benchmarks. Sometimes numbers are just numbers. We checked how we perform in benchmarks only to see that we are not totally off. In reality however the results might be different. So the results should not be read and taken too seriously.

The time is measured by measuring difference between start and stop of the program for running time. For collection time we added the time length of each collection cycle. However for Boehm’s collector we were unable to measure the collection time. We tried measuring the time of allocations as the collections are hidden within them. However the time taken for measuring time then became dominant and ruined the results. Therefore we show collection time only for our GC. We can for simplicity assume that collection time should be quite close to the difference in running time. Because since we have a concurrent collector, ideally no collection overhead exists in the running time of the program. And therefore the difference in running time can be considered collector overhead.

In reality it is a bit more complicated as fragmentation and caching performance plays a role in the results, which might make our mutator thread run faster even if both had no collector overhead. But on the other hand the execution time is not completely without collector overhead. The delay for putting the thread to sleep and awakening it again takes time, and so do safe-points and mutation logs. So for simplicity we shall assume the difference in execution time is close to the collection time.
7.2. Linked lists

Our first benchmark consists of creating many linked lists. Their length is random. They are doubly linked but link back only sometimes. This was just a way to show that our implementation actually can handle loops, and do so without losing too much performance. The results are displayed in Table 7.1:

<table>
<thead>
<tr>
<th></th>
<th>Running time</th>
<th>Collector time</th>
<th>Collection cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boehm stop-and-go</td>
<td>17123 ms</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Boehm incremental</td>
<td>74774 ms</td>
<td>many</td>
<td></td>
</tr>
<tr>
<td>Our GC</td>
<td>16929 ms</td>
<td>1802 ms (9.6%)</td>
<td>42</td>
</tr>
<tr>
<td>Manual memory management</td>
<td>3083 ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 7.1** Linked lists benchmark

As we can see, these results are not optimal. When we compare the number of collection cycles we see that Boehm does a lot less. This is due to several facts. One is that we have two semi-spaces and thus use half the heap size. Apart from that, we already discussed our wasteful memory layout in our cell headers. In this benchmark most of the space was wasted. But we still managed to stay competitive with Boehm’s collector in execution time. And whenever real-time fitness is important and the GC has to respond quickly, we would compare our GC to the incremental Boehm GC, and be way much faster.

7.3. GCBench

The second benchmark is GCBench (Boehm, HP Labs, 2000). This is a famous benchmark originally created by Hans Boehm and has been adapted by many others. We made slight modifications to make the benchmark run with our runtime system. Costs associated with our collector such as write barriers and safe-points were used only with our GC while costs for the runtime system was taken into consideration for both Boehm and our GC, because we want to compare collection time, not how efficient our runtime system is. In Table 7.2 are the results when having holes in the memory:

<table>
<thead>
<tr>
<th></th>
<th>Execution time</th>
<th>Collection time</th>
<th>Collection cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boehm stop-and-go</td>
<td>2555 ms</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Boehm incremental</td>
<td>12255 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our GC</td>
<td>1717 ms</td>
<td>425 ms (19.8%)</td>
<td>7</td>
</tr>
<tr>
<td>Manual memory management</td>
<td>3083 ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 7.2** GCBench benchmark with holes in memory

Note that the difference between ordinary Boehm and us in execution time is 838 ms. If we assume the time difference is garbage collection since our implementation is concurrent, we note that our GC only collected the garbage for 425 ms, so we seem to be about twice as fast here when it comes to collecting garbage. This is not exactly the truth as our mutator threads also have to do some additional work such as safe-points, write-barriers and delays from freezing when collector needs roots, as we discussed previously. So in practice we have half the GC time overhead, meaning we are twice as fast, and this is done on a separate core without interfering running time of the program.

With the same reasoning we are collecting garbage approximately 20 times faster than the incremental Boehm collector, which can be important if responsiveness is crucial. This test was better for us because cells were a bit bigger but still relatively
small except some big objects. Note that we still have to collect more than twice as many times. The same benchmark run without holes in the memory are shown in Table 7.3.

<table>
<thead>
<tr>
<th>Collector Type</th>
<th>Execution time</th>
<th>Collection time</th>
<th>Collection cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boehm stop-and-go</td>
<td>1882 ms</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Boehm incremental</td>
<td>5395 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our GC</td>
<td>1296 ms</td>
<td>157 ms (10.8%)</td>
<td>3</td>
</tr>
<tr>
<td>Manual memory management</td>
<td>3029 ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3 GCBench benchmark without holes in memory

Here we seem to be about 3.4 times faster than Boehm in collection speed with the same reasoning as before. In either case it is noticeable that we perform much better when there are no holes in the memory as we use depth-first traversal and get great caching performance when memory is contiguous in its layout.

As stated before, numbers can be deceiving. All we wanted to show is that the collector is competitive in collection speed compared to other collectors. Apparently it is even faster than Boehm during most circumstances.

7.4. With vs without dynamic analysis

The third experiment involves comparing the running times when using vs when not using dynamic analysis to find pure objects in GCBench.

The results are displayed in the tables below. We measured the running times from start to finish of each respective algorithm with the time taken by the garbage collector within parenthesis. The results are displayed in Table 7.4 when not having holes in the heap.

<table>
<thead>
<tr>
<th>Collector Type</th>
<th>With analysis</th>
<th>Without</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our GC</td>
<td>1296 ms (157 ms collector)</td>
<td>1333 ms (148 ms collector)</td>
</tr>
</tbody>
</table>

Table 7.4 GCBench without holes with vs. without dynamic purity analysis

As we can see now, clearly our original assumption that we can get this analysis for free (timewise) was correct. Not surprisingly we can observe the same cost-free characteristics when holes are present in Table 7.5.

<table>
<thead>
<tr>
<th>Collector Type</th>
<th>With analysis</th>
<th>Without</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our GC</td>
<td>1717 ms (425 ms collector)</td>
<td>1750 ms (403 ms collector)</td>
</tr>
</tbody>
</table>

Table 7.5 GCBench with holes in the heap with vs. without dynamic purity analysis

The benchmarks show that we are as efficient as we want to be and our hypothesis that we can get free dynamic analysis was correct. We could even make collection faster.

7.5. Using the analysis

The fourth benchmark was just a proof of concept as an inspiration to show that the result of this dynamic analysis could potentially result in a potential speedup. What we considered here was the application of loop splitting. What we do is to split up large loops into smaller batches or fragments. To be more concrete, we took the example of finding out what numbers in a list are prime. So we randomly generate prime numbers
and put them in a list. Then we go through these numbers and add the prime numbers to a new list. The loop we are considering looks something like the following:

```
for each number in potential_primes do
    if number.is_prime()
        add number to primes
    end
end
```

Several assumptions can be made here. One assumption is that the lists potential_primes and primes are pure. If so is the case, then all the batches of the loop can be run in parallel. If not, we can still make the assumption that the potential_primes list is pure, which it is in our case. If it is pure, then all objects we get from it, must also be pure. So we can split this batch into two parts – finding if numbers in potential_primes are prime, and adding primes to the primes list. There is a dependency here that the second part depends on the first, but not the other way around. So if the number list is pure, we can always compute if they are prime in parallel, regardless of mutations going on in the primes list. Another thing that can be done in parallel is the second part of the first batch and the first part of the second batch. This is also possible as long as the potential primes list is pure – an assumption that is never broken. Even if adding a number to primes would somehow break the assumption that potential numbers is pure (which it does not), we may still get potential benefits if the mutation is seldom enough, since the majority of numbers we go through are not prime in our example. Figure 7.1 shows the relationships between what can be done in parallel (independently), which could be automatically generated.

![Figure 7.1 Splitting loops into independent batches and batches into independent tasks](image)

The assumptions that can be made and their effects are displayed in Table 7.6.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumption #1 primes and potential_primes are pure</td>
<td>Batches can be executed in parallel</td>
</tr>
<tr>
<td>Assumption #2 potential_primes is pure but primes is not</td>
<td>Finding if numbers are prime in each batch can be done concurrently. Finding if numbers are prime in the next batch can be done while serially adding primes to the primes list.</td>
</tr>
</tbody>
</table>

| Table 7.6 The condition and effect of generated assumptions |

26
Of course more assumptions can be made. For instance we could look at individual objects if they are pure as perhaps not the whole potential_prime list is pure (even though in our case it is). Then pure ones could be executed in a concurrent queue while the not pure objects could execute in a serial queue to keep the serial semantics of the system. In our benchmark we simply invalidate all work in a batch if an assumption proves to be wrong. Note that there are more optimal solutions where we do not invalidate all work when we reach a mutation that proves our assumption wrong. We can invalidate only the work done after the mutation and keep the rest as it still preserves the synchronous nature of the program.

The benchmark was implemented using Grand Central Dispatch, or libdispatch developed by Apple but released open source (Apple, 2009). It is an extension of C with closures, or anonymous functions. These can be scheduled asynchronously or synchronously to run on either concurrent or serial queues. This is a very handy when we want to preserve the synchronous nature of the program. The results we got are displayed in Table 7.7.

<table>
<thead>
<tr>
<th></th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous execution</td>
<td>26.7 seconds</td>
</tr>
<tr>
<td>Natural concurrent version with assumptions #1 and #2</td>
<td>15.1 seconds (76% faster)</td>
</tr>
<tr>
<td>All assumptions being wrong for no apparent good reason the second they are assumed (which is naturally not the case)</td>
<td>30.1 seconds (11 % slower)</td>
</tr>
</tbody>
</table>

**Table 7.7** Results of benefits of parallelization using assumptions in Table 7.6.

If assumptions were to break, the running time depends on a few things. First of all it depends on the batch size. Bigger batches failing results in longer running times if they have to be re-done, and higher thresholds for how long the system must go unchanged before being considered immutable also plays a key role. If however all work in the batch was not deleted, only what violated serial execution order, it would matter less and this worst-case scenario could get cheaper.

What we did find when it comes to the natural execution is that with smaller batches we utilized less CPU power. Larger batches resulted in better CPU usage, but at the same time, it leaves smaller time windows between mutations where parallelism can be exploited.

This example of loop splitting is only one application where this is useful. It is also useful whenever we have filters or streams where mutations are rare. In a general sense we can exploit this analysis whenever mutations are clustered in time. So if we for instance have a neural network, it has a training phase with mutations going on to weights, and when that is done, we use it to query information. With this system it could be trained serially and then be queried in parallel automatically.

The list of applications goes on and on. We think this can be quite useful in practice.
8. Conclusion

The overhead of the write-barrier is higher than the write-barrier for a mark and sweep collector but lower than a read-barrier, and can be made arbitrarily low by having only one barrier before each safe-point. This only overhead allows us to use generational collection, keep track of unchanged cells, use pointer-reversal concurrently, keep low synchronization costs and good scalability. It is also portable and should work on almost any machine. It is however a moving collector and will not work without a cooperative compiler which can accurately identify pointers.

Table 8.1 is a table of metrics from the goal criteria and how we managed to solve the problem.

<table>
<thead>
<tr>
<th>Performance</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time fitness</td>
<td>Good</td>
</tr>
<tr>
<td>Concurrency &amp; synchronization overhead</td>
<td>Low</td>
</tr>
<tr>
<td>Memory compactification &amp; caching</td>
<td>Good</td>
</tr>
<tr>
<td>Barrier overhead</td>
<td>Low</td>
</tr>
<tr>
<td>Generational</td>
<td>Yes (but not implemented)</td>
</tr>
<tr>
<td>Scalability</td>
<td>Good</td>
</tr>
<tr>
<td>Allocation speed</td>
<td>High</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Portability</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Portability</td>
<td>Good</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accuracy</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Accurately detects pure objects</td>
<td>Yes</td>
</tr>
<tr>
<td>Accurately collects live objects</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 8.1 Goal metrics

As the table suggests, this is a good candidate solution. Note that this collector is very good when it comes to generational collection as the mutation log could be used to keep track of inter-generational pointers for free. However it was not implemented in this proof of concept.

We managed to create a concurrent collector that collects garbage while the mutator threads are executing, relieving us of waiting times. It allows multiple mutator threads and has low synchronization costs, fulfilling our requirement of working well in a parallel environment. We managed to keep the algorithm down to one single depth-first pass through the graph, in linear time with respect to the number of edges and nodes, and keep the memory constant while collecting. Most importantly, we also managed to accurately identify the pure objects, and additionally, we made it possible to approximate how likely it is that an object will mutate based on its history. Therefore the goal criteria have been met.

Moreover, the benchmarks showed that we got the dynamic analysis for free as anticipated, while also managing to collect faster than Boehm-Demers-Weiser’s state of the art mark and sweep collector.
9. Future work

There are many things to be done. The main contribution in this thesis is the idea of using the garbage collector to gather information about the dynamically changing landscape of our programs, without significant overhead, and use this information to allow for instance dynamic automatic parallelization. Making the transformations from serial code to parallel using the dynamic assumptions provided by the garbage collector is a future project.

What was implemented was a framework for the automatic memory management system that virtual machines can use. The next step would be the integration of the garbage collector into a host VM for an object oriented programming language with accurate pointers, or the creation of a new language.

The garbage collector could also be improved with generational collection (as we already have a free mechanism for keeping track of inter-generational pointers). This would give shorter delays until we find new pure objects and reduce collection time in programs where old data persists longer.

A mechanism could also be developed to dynamically vary the heap size to get rid of delay times (prevent stop when memory runs out by adapting heap size) and make it more space efficient (to-space could be eliminated almost entirely) as well as reducing synchronization overheads with thread-local and allocation spaces.

Cell headers could also be made more space efficient, which is useful for smaller objects. Several methods has been discussed such as using stacks instead of pointer reversal, using relocation maps instead of replica header fields and having only one isa-pointer in either the replica or the original cell.

Another thing that might be of interest is adding a mechanism for handling large objects. Currently every object is copied when it is mutated. This is not ideal for large objects, and a mechanism for treating them differently could prove useful.

I also wrote a paper on this. Unfortunately it has not yet been accepted and thus I cannot reference it at this moment.
References


SE-391 82 Kalmar / SE-351 95 Växjö
Tel +46 (0)772-28 80 00
dfm@lnu.se
Lnu.se/dfm