Cooperative Memory Management

Version	Author	Summary
2014-09-08	PSK	Initial Draft
2014-09-14	PSK	Initial CMM API outlined, benefits listed and CMM usage examples
2014-09-15	PSK	Extended CMM API with HeapRef preparing it for MT processing
2014-09-16	PSK	Also added the means to merge Heaps
2015-03-15	PSK	Cleanup and removal of irrelevant sections

Abstract

CMM combines the benefits of both manual and automated memory management. It represents on one hand an API for application / systems developer to provide the runtime with hints regarding their memory utilization and on the other hand a virtual machine (runtime) that is capable of interpreting these hints in order to optimize the application beyond the current possible.

Given the ever increasing importance of the memory hierarchy on modern hardware, memory management represents a fundamental factor of system performance and is a deciding factor when scaling systems on current hardware.

Contents

Introduction

Garbage collection is a central feature of modern programming environments / languages. It frees up the programmer of the tedious and error prone task of memory management and automates the reclamation of unused memory. Garbage collection, since its inception in the 60s, has come a long way.

The two major forms of garbage collectors are tracing and reference counting. As reference counting cannot handle cyclic references, systems tend to employ tracing collectors. Escape analysis appears to be one of the current research areas by which objects are stack allocated, where possible, to delay the collection of the heap.

Assumption A: The complexity (cost) of tracing collectors is dictated by the number of live objects!

Assumption B: Most objects die young - The generational hypothesis!

State of the art tracing garbage collectors found in commercial products are (concurrent, incremental), parallel and generational collectors utilizing mark sweep (compact) for the old generation and mark copy for the new generation with the intention of reducing and/or evenly distributing the cost of garbage collection.

- Concurrent collectors execute while the mutators (the application) are running
  - Timing, concurrency and CPU scheduling (CPU oversubscription) issues
- Incremental collectors adhere to a time budget (time boxing) in which they perform incrementally some tasks relevant for garbage collection
  - Timing, scheduling and interruptions in critical sections issues
- Parallel collectors utilize multiple CPUs (threads) to collect the heap
  - CPU cache trashing issues, due to massive pointer tracing
- Generational collectors, originating from the generational hypothesis, divide the heap into generations and utilize the most optimal strategy per generation
  - Old generations collected infrequently and full collections are very time consuming
- Escape analysis determines the scope of objects in order to allocate the viable ones on the stack, which are then de-allocated upon popping the respective stack frame
  - Very difficult to determine and analyze complex and dynamic call graphs statically

The performance of garbage collectors is generally measured by throughput and latency.

- By throughput one understands the speed at which the application is performing, that is, the impact the collector has (by means of barriers etc…) on the execution time of the application
- Latency measures the time a collector delays the actual application from executing by stopping it in order to perform memory reclamation

Thus, the properties of a good garbage collector are high throughput and low latency.

The remainder of this document focuses on classifying current applications and devising strategies to achieve the above mentioned properties for a good garbage collector.

Application Types

Server Applications

Most server class applications work on a request / response basis. Their memory usage pattern is linked to the number of in-flight requests. That is to say, during the processing of requests memory is used and upon finishing a request memory becomes, in theory, available for the processing of other requests.

Client Applications

Interactive client applications are most often event based, that is, they react to events triggered by a user. For example, when loading a file for editing the memory usage increases and upon closing that file the memory can be reclaimed by the system. Non interactive client applications behave in a similar manner, just that there is no user interaction involved in the whole process.

Processing style

Single threaded

In single threaded processing a request or event is “fully” processed by a single thread. It is safe to assume that at the time of writing most applications follow the single threaded processing model, be they client or server applications. Although a lot of client applications use multiple threads to increase responsiveness, they usually process a user event within a single thread. Application servers also utilize multiple threads, but servicing a request is traditionally done within the scope of a single thread. Multi threaded programming / processing of a single request is even explicitly forbidden by the J2EE specification.

Multi threaded (SMP)

Multi threaded processing, as the name implies, utilizes multiple threads to service a request or event. While not very mainstream at the time of writing, several multi threading models (actor, continuations, fork/join, disruptor, thread networks, etc…) and frameworks (Play, Fast Flow, TBB, OpenMP, etc…) are emerging and challenging classical processing models.

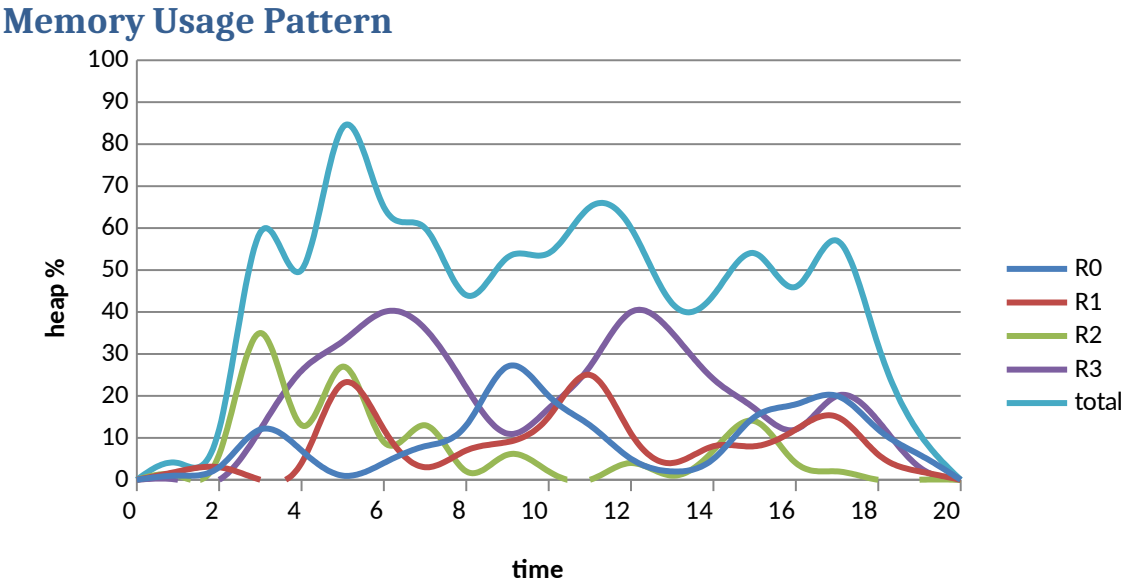

Figure 1: Heap Memory Usage

The above diagram depicts imaginary heap memory utilization during the processing of four requests. Given Assumption A, timing of the garbage collection can help reduce the collection cost. The best time to perform a whole heap collection appear to be at t=4, t=8, t=13 and t=16.

While these represent local minima for the whole heap utilization,

- They are notoriously difficult to find
- They require tracing of about 50% of the objects in the heap for each collection (naïve)

Based on these observations, we conclude the first strategy.

Strategy A: Perform garbage collection at a request scope.

Given Strategy A, we now look at the imaginary memory utilization of a single request.

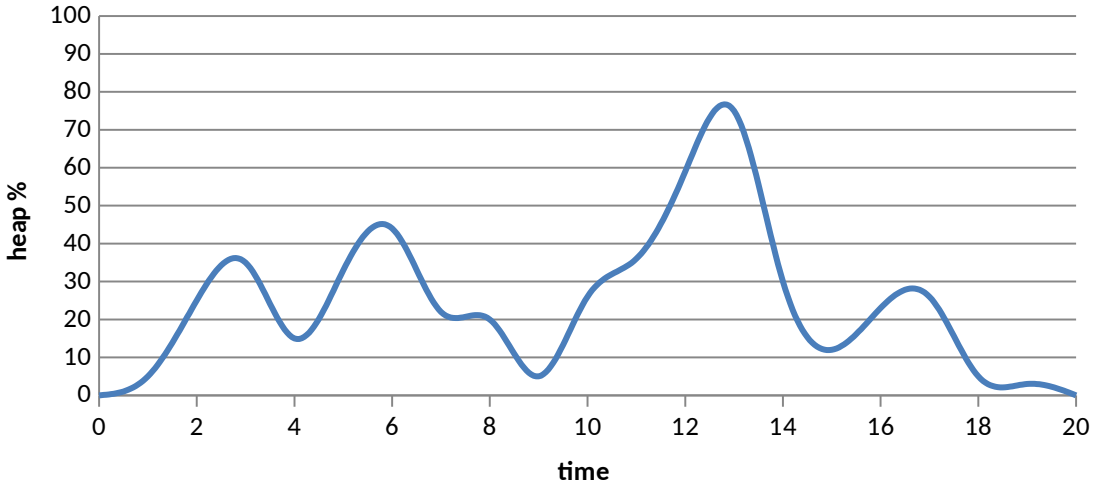

Figure 2: Heap Memory Usage - Single Request

Given Assumption A and Assumption B, the timing of garbage collection becomes now a fundamental factor. The best times to perform GC for the above memory usage pattern would be at t=4, t=9, t=15 and t=18, because at these times the number of live objects are at their respective local minimum and thus require the least tracing effort, leading to the second strategy.

Strategy B: Perform garbage collections at times of minimal memory utilization.

A word of warning, the graphs above are not statistically backed up, that is to say, they represent an educated guess by the author regarding system memory utilization.

Cooperative Memory Management

So far, we have manual memory management and automated memory management in systems. CMM represents a memory management policy that tries to combine the strengths of both manual and automatic memory management. The main concepts behind CMM are scoped heaps and the ability to collect these scoped heaps individually. CMM provides an API to application developers to created, collect and dispose of scoped heaps, thus allowing an application developer to fulfill both Strategy A and B, while at the same time upholding the benefits of automated memory management.

```
public interface CMM {
	/** creates a new scoped heap */
	Heap createHeap();
	/** returns the Heap used by the current Thread */
	Heap getCurrentHeap();
	/** instructs the current thread to use the given scoped heap */
	void useHeap(Heap heap);
	/** instructs the current thread to use the global heap */
	void useGlobalHeap();
	/** gets the global heap */
	Heap getGlobalHeap();
}

public interface Heap {
	/** merges this heap with the given one */
	Void merge(Heap heap);
	/** creates a heap reference to an object */
	HeapRef createRef(Object obj);

	/** instructs the system to collect the heap */
	Void collect();
	/** disposes this heap */
	Void dispose();
}

public interface HeapRef {
	/** retrieves the object associated via this reference */
	Object getObject();
	/** retrieves the heap */
	Heap getHeap();
}
```

Given the simple API above, it is now possible for the developer to define the scope of a heap and decide when this heap should be collected and disposed off. Once this API is used correctly by a developer, the runtime becomes able to manage memory more efficiently.

CMM Examples

Here are presented some examples of how to utilize CMM. They also show the simplicity of CMM and highlight that code having CMM logic will perform without any issues in non CMM runtimes by providing Mock CMM implementations.

Single threaded

Simple single threaded application demonstrating the usage of CMM.

```
public class Test3 {
   public static void main(String args[]) throws Exception {
      // bootstrap CMM
      Heap heap = CMM.createHeap();
      CMM.useHeap(heap);
      long start = System.currentTimeMillis();
      ArrayList list = new ArrayList();
      int count = 0;
      for (int i=0; ; i++) {
         Object[] data = new Object[1024/24];
         for (int j=0; j<data.length; j++)
            data[j] = new Object();

         list.add(data);
         if (i%1000000==0) {
            long delta = System.currentTimeMillis() - start;
            System.out.println(""+count+"\t"+delta);
            count++;
            list.clear();
            // collect the scoped heap here,
            // cause we know that here the number
            // of live objects are at their lowest
            heap.collect();
            if (count>100)
               break;
         }
      }
      // fallback to the global heap
      CMM.useGlobalHeap();
      // now dispose the heap altogether
      heap.dispose();
   }
}
```

Thanks to collecting the scoped heap at the correct time (Strategy B), the GC pauses will be very small here.

Multi threaded example (single threaded processing)

The same as before, but multi threaded at a processing scope. The realization of Strategy B is clearly visible here.

```
public class TestMt3 {
   private static class Allocator implements Runnable {
      private String id;

      Allocator(String id) {
         this.id = id;
      }

      public void run() {
         // bootstrap CMM
         Heap heap = CMM.createHeap();
         CMM.useHeap(heap);
         long start = System.currentTimeMillis();
         ArrayList list = new ArrayList();
         int count = 0;
         for (int i = 0;; i++) {
            Object[] data = new Object[1024 / 24];
            for (int j = 0; j < data.length; j++)
               data[j] = new Object();

            list.add(data);
            if (i % 100000 == 0) {
               long delta = System.currentTimeMillis() - start;
               System.out.println(id + "\t" + count + "\t" + delta);
               count++;
               list.clear();
               // collect the scoped heap here,
               // cause we know that here the number
               // of live objects are at their lowest
               heap.collect();
               if (count > 100)
                  break;
            }
         }
         // fallback to the global heap
         CMM.useGlobalHeap();
         // now dispose the heap altogether
         heap.dispose();
      }
   }

   public static void main(String[] args) {
      for (int i=0; i<10; i++)
         new Thread(new Allocator(""+i)).start();
   }
}
```

J2EE Web Container (via ServletFilter)

A ServletFilter based demonstration, when the Server does not support CMM, of how CMM could be used to optimize the processing of client requests, demonstrating the realization of Strategy A.

```
public class ScopedHeapFilter implements Filter {

   public void doFilter(ServletRequest req, ServletResponse res,
         FilterChain fc) throws IOException, ServletException {

      // bootstrap CMM
      Heap heap = CMM.createHeap();
      CMM.useHeap(heap);
      try {
         fc.doFilter(req, res);
      } finally {
         // fallback to the global heap
         CMM.useGlobalHeap();
         // now dispose the heap altogether
         heap.dispose();
      }
   }

   public void init(FilterConfig fc) throws ServletException {
   }

   public void destroy() {
   }
}
```