Chapter 1

Introduction

Improving GC in HHVM is a project that aims to improve the efficiency and throughput of the garbage collection implementation inside the Hip-Hop Virtual Machine (HHVM).

1.1 Background Knowledge

The reader is advised to ensure they have a working knowledge of the following topics.

1. All facets of the 2011 standard C++ programming language, including lambda expressions, classes and structs, templates, polymorphism and inheritance, pointer arithmetic, the std library, threads and concurrent programming, and dynamic memory allocation. The engineers who developed HHVM exploited every capability of C++.

2. An understanding of the basic principles of garbage collection, including the main algorithms in use (reference counting, mark sweep) and the difference between exact and conservative collection.

Included in this repository are a number of background documents that the reader should familiarise themselves with. Within the *background*
immix directory are Steve Blackburn's papers on the immix garbage collection algorithm. These papers trace the evolution of immix from a mark-region collector to the conservative reference-counting algorithm that is being integrated into the HHVM.

The other directories within *background* contain all the contributions to this project that have been made so far. In chronological order, they are:

- 2014 - Ben Roberts' exploration of the existing HHVM Memory Manager system.

- 2014 - Tim Sergeant's analysis of the consequences that removal of reference counting will have on PHP semantics.
- 2015 - Peter Marshall's continuation of Tim's analysis. Peter experimented with the removal of reference counting and laid the foundation for the implementation of Immix within HHVM.

It remains to be seen whether this work will make a meaningful contribution to the project. At the very least, it is hoped that by reviewing the work that has been done so far, the learning curve will be lessened for the next person who contributes to this project.

Chapter 2

What Is Not Known

2.1 To do list

- Trace the usage of the *Node structs that are defined within *memory-manager.h*.
- Trace every call to malloc, calloc, realloc and free within the codebase. Do they all go through the safe wrapper methods in alloc.h, or are there some places where the builtin functions are called directly?
- Trace every invokation of every MemoryManager method.
- Trace every instantiation of a MemoryManager object.
- Establish how multiple threads interact with the MemoryManager. The MM documentation suggests that it is thread-local, so there each thread has its own MemoryManager. Find out where these multiple threads come from. Is PHP running in parallel? Or is it just 'server mode' that spawns threads for each incoming request?
- Learn how to use the makefile to take advantage of the existing debug and trace code.
- Investigate methods MemoryManager::iterate and BigHeap::iterate. What do they do? Who calls them? Note that they use the Header struct - what creates these?
- WHERE IS THE ALLOCATOR AND HOW DOES IT WORK.
- Find out what is meant by 'template marker' and 'virtual call marker'.
- Trace the interface declarations of *ObjectData*, *ExceptionData* and *ResourceData* structs.

- Understand the way that *heap-collect.cpp* works, because it is the mark-sweep collector. Find out when it is invoked any why.

2.2 Ways to Improve the Code

- Split the implementation of *TldPodBag* into a .cpp file so changing the implementation won't require a rebuild of the entire codebase.
- Find more opportunities to use the TldPodBag. It is used in one place in the entire codebase. Either delete it or use it more often.
- Another file that needs implementation separated from interface is *heap-collect.cpp*. And *heap-scan.h*.

Chapter 3

What is known

3.1 MemoryManager

This project deals primarily with the *MemoryManager* component of HHVM. The MemoryManager is responsible for the allocation and deallocation of managed memory. ** It is not yet clear whether or not MemoryManager deals exclusively with PHP objects or whether it extends to C++ objects for HHVM itself **

The implementation of MemoryManager is spread across several .h and .cpp files.

memory-manager.h
Contains the following things:

- The interface for the HPHP::MemoryManager class. Some parts of the interface are dependant on #define flags being set; namely, the ContiguousHeap and jemalloc options.
- The interface for the HPHP::req::Allocator class.
- The implementation of the HPHP::req::Allocator class.
- Slab size constants, alignments and so forth.
- Struct declarations for *Node classes. Their function and use has not been investigated yet.
- The interface for the HPHP::BigHeap struct.
- The interface for the HPHP::ContiguousHeap struct, which extends BigHeap.

memory-manager.cpp
Contains the implementation of many of the MemoryManager, BigHeap and ContiguousHeap methods declared in the header file above.

memory-manager-inl.h
Contains the implementation of methods of the *req* namespace, and a lot of MemoryManager method implementations as well. Much of the code in this file is inline code.

memory-manager-defs.h
Contains the following:

- The interface and implementation of the HPHP::Header struct.
- Implementations of MemoryManager and BigHeap methods; namely *iterate* and *forEach* methods that use the *Header* struct defined above to iterate over objects in the heap.

heap-scan.h
This file contains:

- A number of methods declared within the scope of HPHP but not attached to any classes or structs. These include *scanHeader* and *scanRoots*, but there are others.
- The interface for the struct *ExtMarker*, which is the 'bridge between the template based marker and the virtual call based marker'.
- Implementations of *ObjectData*, *ExceptionData* and *ResourceData* methods.

heap-collect.cpp
Contains the following..

- Incredibly, interface declarations for the *PtrMap*, *Counter* and *Marker* structs. Some methods of the Marker struct are not implemented yet - this is a work in progress.
- Implementation of *MemoryManager::collect*. This method instantiates a Marker object (as per above), initialises, marks and sweeps.

This file contains most of the implementation of the mark-sweep tracing collector and is worth investigating in detail.

Another file that is worth taking a look at is *header-kind.h*, which contains the declarations and implemetations of enums HPHP::HeaderKind and HPHP::CollectionKind, and the struct HPHP::HeaderWord, which is a 64 bit (8 byte) carefully designed data structure that packs a bunch of object information into the front of a heap-allocated object in memory. This file also contains a bunch of HPHP methods that are not owned by any class or struct but do perform checking operations on HeaderKinds and CollectionKinds.

3.2 Utilities

HHVM has, within *alloc.h*, 'safe' versions of standard C memory allocation functions. malloc, calloc, realloc and free are wrapped around inline methods safe_malloc, safe_calloc, etc. These methods all potentially throw *OutOfMemory* exceptions, so it is possible that they are not ubiquitous. ¡– investigate this

3.3 TldPodBag

The *TldPodBag* is a data structure that is used to store an array of POD data. It is designed for a small set of data of variable length. The data need not be contiguous; elements of the array can be added and removed from the array without shifting other elements into the hole.

Empty slots in the array are identified by the first eight bytes of the slot being zeroed out. It necessarily follows that the data type that the PodBag stores must be at least eight bytes long. The entirety of the codebase sits in a file called tld-pod-bag.h. It might be worth looking at splitting it into proper .h and .cpp files in due course, because header files are not supposed to be used for implementation details ffs. It might speed up compilation times too.

TldPodBags are used in only two places in the entire codebase. Within *array-iterator.h* the PodBag is used to store iterator/array pointer pairs in excess of the default number of seven, which is hard coded into the program. The documentation inside that file indicates that it will be incredibly rare for the number of iterators being stored to ever exceed seven - it is incredibly rare for it to ever exceed *one*. The result of this is that my effort to 'instrument' the PodBag, to gain some understanding of how often it is used and how expensive its operations are, was almost completely useless, as the PodBag's code is never actually invoked unless it encounters certain types of PHP scripts – **** need to check this, is the array iterator a PHP thing or a C++ thing??? ****