# High Performance Mark-Region Garbage Collection in the HipHop Virtual Machine

**Peter Marshall**

A thesis submitted in partial fulfillment of the degree of
Bachelor of Software Engineering (Honours) at
The Research School of Computer Science
The Australian National University

October 2015

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Peter Marshall
21 October 2015

To my parents Ian and Robyn,
who taught me how to think.

# Acknowledgements

First of all, an enormous thanks to my supervisor Steve Blackburn. Thank you for your wisdom and guidance over the course of this thesis, your encouragement, your insight and your passion. When I started I knew basically nothing about research in computer science. Thank you for welcoming me to your research group.

To Edwin Smith of Facebook, thank you for guiding me through the vast ocean of HHVM and for your extreme generosity in making time to meet with me so regularly. A special thanks for welcoming me to Boston and providing much needed friendship and mentoring over the year. A warm thanks also to Rick Lavoie, Bill Nell, Jason Evans and everybody else who has worked with me from Facebook throughout the year.

To Tony Hosking and Michael Norrish, a great thanks for your interest in my research and for your ongoing support. To Xi Yang, Kunshan Wang, Yi Lin, Luke Angove, John Zhang and Theo Olsauskas-Warren, thank you for all of your conversations, suggestions, detours, work-arounds and commiserations. It's been excellent to work as a researcher alongside such talented people.

To Migara Liyanagamage, thank you for being my human rubber duck and for letting me explain every detail of my work to you in excruciating detail at least four times.

To my friend Chris Wodzinski, a great thanks for your feedback on my work and lunchtime phone conversations to draw me away from my desk.

To my girlfriend and companion Brigitta Quantock, immense thanks for your support and encouragement over the past year. Thank you also for reading my drafts, despite studying History.

To my housemate Geoff, I'm sorry, I'll do the dishes soon, I've had a lot on my plate.

To my parents Ian and Robyn, thank you for encouraging me to pull things apart my entire life, for supporting me in all my endeavors and for showing me the value of hard work.

To my friends and my family, my immense gratitude.

# Abstract

The PHP language is one of the most widely used programming languages for website development, and many large companies such as Facebook and The Wikimedia Foundation rely on it to run their extremely high traffic websites. The speed and efficiency of PHP implementations is paramount to the success of such websites. Garbage collection plays a critical role in the performance of managed languages such as PHP, however all PHP implementations to date use naive reference counting garbage collection algorithms, which are inefficient and unable to compete with tracing garbage collection. The use of reference counting is not a coincidence, because the way that the PHP language was designed strongly implies its use. Some PHP applications in widespread use have even come to depend on implementation-specific features related to reference counting, complicating further the relationship between PHP and garbage collection. These obstacles prevent PHP runtimes from utilising modern, well-performing garbage collection techniques and restrict the performance of PHP.

It is my thesis that the barriers to high performance garbage collection in PHP can be overcome, and that modern, high performance garbage collection algorithms can significantly improve the performance of PHP engines.

This thesis analyses the literature to identify opportunities to improve the performance of PHP through more modern garbage collection techniques. PHP engines rely heavily on a deferred array copy mechanism for their performance, which is implemented using reference counting. I propose a one-bit reference counting scheme, which alleviates the need for naive reference counting. The one-bit reference count still captures much of the information required of the previous reference counting scheme, but at a far lower cost. The HipHop Virtual Machine is the high-performance PHP implementation that powers Facebook, and serves close to one billion unique users per day. It serves as the experimentation platform for this work, which introduces a mark-region garbage collector and produces a functioning, high performance PHP implementation that does not rely on precise reference counts.

The one-bit reference counting scheme introduces a small performance overhead of 1-2% because of additional copy operations caused by copy-on-write, which is cancelled out by the lower cost of maintaining the far simpler one-bit scheme. With copy-on-write taken care of, full reference counts are redundant and the one-bit scheme can be used to inform all garbage collection, copy-on-write and reference behaviour. Without reference counting, PHP suffers a total performance decrease of only 3%, despite poor performance due to incomplete garbage collection. I introduce a mark-region block and line style allocator to replace the free-list allocator, and present novel optimisations to explicitly managed allocations in HHVM, showing that PHP can be made compatible with tracing garbage collection.

This work has implications for other managed languages where the design of the language seems to force decisions in the implementation, by showing that these can be overcome. Additionally, showing that high performance garbage collection can incorporate a one-bit reference counting scheme provides more flexibility to any other language implementations that may depend on reference counting. This thesis opens a new path for improvement of the PHP language in general by showing that it can be compatible with modern advances in garbage collection techniques.

# Contents

Chapter 1

# Introduction

This thesis explores high performance garbage collection in PHP, which until now has relied on outdated memory management techniques. By identifying the ways in which PHP relies on naive reference counting and examining alternatives, PHP can adopt with high performance memory management techniques.

## 1.1 Thesis Statement

There are two main branches of garbage collection work: tracing garbage collection [McCarthy 1960] and reference counting [Collins 1960]. Tracing garbage collection identifies reachable objects by performing a closure over the heap from a set of known roots, while reference counting identifies unreachable objects by maintaining a count of the number of referents to each object. Implemented naively, reference counting can cause considerable runtime overhead [Shahriyar et al. 2012].

The HipHop Virtual Machine (HHVM) has made significant progress in achieving a high performance implementation of PHP, but still makes use of naive reference counting as a garbage collection algorithm. The challenges that must be overcome in order to make use of modern tracing garbage collection algorithms are significant, due to the language's reliance on the semantics of reference counting. All major PHP implementations use reference counting garbage collection, with good reason. The design of the PHP language implicitly encourages reference counting by including language features which are difficult to implement without reference counting.

*My thesis is that the barriers to high performance garbage collection in PHP can be overcome and that modern garbage collection algorithms will increase the performance of current PHP implementations.*

PHP has achieved widespread use as a programming language for web servers and remains the most commonly used language for this task. Facebook relies heavily on PHP, and created HHVM, a PHP engine with the goal of being a high performance alternative to the default PHP implementation by Zend (Zend PHP). HHVM also runs the Hack language, a statically-typed version of PHP with some additional language features, also created by Facebook and used widely across their codebases. However,

HHVM still uses naive reference counting as a garbage collection algorithm, which is slow and inefficient compared to more modern tracing garbage collection algorithms.

In order to modify HHVM to use more modern tracing garbage collection techniques, several issues regarding PHP's reliance on reference counts must be addressed. Examples of this include the copy-on-write optimisation [Tozawa et al. 2009], precise destruction of PHP objects and precise demotion of reference types to values.

## 1.2 Contributions

This thesis aims to (a) identify and analyse the ways in which PHP as a language is tied to naive reference counting as a garbage collection strategy, and (b) implement solutions that make PHP compatible with tracing garbage collection.

I analyse the copy-on-write optimisation in HHVM and reach the conclusion that deferred coping of arrays and strings is essential for the performance of PHP. I find that a one-bit reference counting scheme allows deferred copying of arrays and strings, eliminating the need for exact reference counting. This is an extremely promising alternative to maintaining full reference counts that allows for deferred copying whilst reducing PHP's reliance on reference counting garbage collection.

With reference counting removed, I explore the design space for tracing garbage collection in HHVM. I propose a prototype design for a new garbage collector and provides analysis of the implementation. This design is based on the Immix garbage collector, a modern mark-region memory management scheme that has been shown to perform extremely well, but has not been tailored for PHP or HHVM. With some modification, the Immix-based collector can be implemented in HHVM, resulting in the first PHP runtime that provides competitive performance without reference counting. This implementation can act as a basis for further work to improve the performance of tracing garbage collection in HHVM and for PHP in general.

## 1.3 Meaning

When designing new languages, a creator must carefully consider many competing factors and interests to arrive at a coherent, sensible result. In the early stages of a language, the design of the language and the development of its reference implementation may co-evolve, and over the history of programming languages it is possible to see how decisions made at these early stages of the design process have affected the direction of the language many years later [Jibaja et al. 2011]. By showing the difficulties that arise in searching for solutions to problems that are introduced early in the life of a language, this thesis shows the importance of giving due consideration to garbage collection strategies in the design of managed languages. Language designers should contemplate the interactions between language semantics and implementation options to avoid constraining the language to specific methodologies.

Solving the problems encountered in PHP and the adaption of a high performance garbage collector from literature [Blackburn and McKinley 2008] to a production im-

plementation shows that there are promising pathways for managed languages that have constrained performance due to design decisions. The analysis and techniques presented in this thesis present a basis for further exploration of high performance implementations of PHP and open the door for a new approach to memory management in PHP.

## 1.4 Thesis Outline

Chapter 2 provides background information relevant to the key contributions of this thesis. It introduces and motivates garbage collection, common algorithms and typical collectors, terminology and concepts helpful for the understanding of the later chapters. This chapter also introduces the PHP language and presents the specific aspects relevant to this thesis, along with HHVM.

Chapter 3 gives an outline of the experimental methodology and details needed to understand the context of the results given elsewhere in this thesis.

Chapter 4 addresses the reliance of PHP on reference counting and introduces the one-bit reference count, motivating the usage and detailing results to show that the scheme is a suitable basis for exploration of further garbage collection work in HHVM.

Chapter 5 explores the design space of a tracing garbage collector in HHVM, taking the highly performing Immix garbage collector and detailing the specific considerations for its inclusion in HHVM. It presents the analysis of a working prototype and further avenues of optimisation.

Chapter 6 draws together the results of this thesis and makes the conclusion that PHP can be made compatible with high performance garbage collection techniques, and opportunities for further work are presented.

Chapter 2

# Background and Related Work

This chapter outlines the history of automatic memory management, from the seminal publications in the field through to the most recent evolutions of these ideas. It also introduces PHP and HHVM.

Section 2.1 provides a broad overview of garbage collection, motivating its importance and introducing key ideas. Section 2.2 introduces the details of the various garbage collection algorithms that are discussed elsewhere in this thesis, and introduces relevant terminology. Section 2.3 outlines the canonical garbage collectors and addresses their trade-offs. Section 2.4 introduces the PHP language, describing its history and its peculiar relationship with garbage collection. Section 2.5 describes HHVM and outlines its current approach to memory management. Finally, Section 2.6 briefly outlines some prior work in similar areas of research.

## 2.1 Garbage Collection Overview

Garbage collection refers to automatic memory management in a program or a language runtime. Garbage collection deals with the management of dynamically allocated memory with two main goals: to increase performance, and to increase memory efficiency.

In a language where no garbage collection is provided, the task of dynamically allocating and reclaiming memory is left to the programmer. Programmers are required to explicitly request space for each dynamically allocated data structure and must also explicitly indicate that the space is no longer required. Mistakes in properly managing memory can lead to subtle and hard to find bugs, as well as poor utilisation of resources. Memory management walks a fine line between efficiency and correctness. Any object which is retained when it is no longer required is considered a memory leak, and any object reclaimed while it is still needed can cause severe failures. Automatic memory management takes this task out of the hands of the programmer and provides a memory-safe environment, relieving the programmer of this burden.

*Managed languages* provide automatic memory management. Any dynamic memory management scheme has several main tasks:

1. Memory allocation: providing space in memory for new objects.

2. Garbage identification: locating memory that has been allocated, but is no longer needed.

3. Garbage reclamation: reclaiming unused memory so that it can be allocated again.

Each of these tasks can be accomplished through any number of different approaches and algorithms, which Section 2.2 describes in detail.

Proper management of dynamic memory allocation is extremely important to the performance of managed languages, such as PHP. The garbage collection strategy being employed can significantly affect the performance of the running program, referred to as the *mutator*, and also controls the memory footprint of the running process. For this reason, there has been significant research into improving garbage collection strategies.

## 2.2 Garbage Collection Algorithms

Garbage collection algorithms must allocate space for objects, identify dead objects and reclaim unused memory. Many choices exist for each of these three tasks. The following sections present an introduction to each, before the major garbage collectors are introduced.

### 2.2.1 Allocation

Allocation is the task of assigning space for an object in memory. An allocation algorithm must take care to ensure that the same space in memory is never allocated to two or more objects at the same time. Several typical allocation algorithms exist.

#### Bump pointer

In bump pointer allocation, a pointer is held to the beginning and the end of some block of memory. When an allocation is requested, the allocator simply checks that the space requested will fit between the beginning and end of the current block of memory. If it does, then the first pointer is incremented or 'bumped' to indicate that the memory before it is no longer available. Bump pointer allocation is easy to implement and has the advantage of allocating objects contiguously in memory. Objects that are allocated sequentially are more likely to be accessed sequentially at a later time, and as a result bump pointer allocation generally increases locality [Blackburn et al. 2004]. Bump-allocated memory can be difficult to re-use except in bulk, or unless a hybrid scheme such as an overlaid free-list is used.

#### Free-list

Free-list allocation maintains a list or multiple lists of free chunks of memory, and responds to allocation requests by finding a chunk of a suitable size. When memory

is reclaimed, it can simply be appended to the appropriate free-list for re-use. Finding an appropriate chunk of memory can be potentially costly and involves the trade-off of quickly finding a chunk that is at least as big as the requested space, but not much larger, to avoid wasted space. Many different strategies exist to optimise the use of free-lists, such as maintaining size-segregated lists where every chunk in a particular list is the same size. Finding a chunk in a list can then be completed in constant time provided that requested sizes can be mapped to individual lists [Ugawa et al. 2010].

### 2.2.2 Identification

Identification is the task of recognising which allocated spaces in memory are no longer needed. Two categories exist: *implicit*, where all live objects are identified and any other memory that was allocated is implicitly dead, or *explicit*, where dead objects are identified and all other memory is assumed to be live. In practice, reachability is used as a conservative proxy for liveness, meaning all reachable objects are assumed to be live.

#### Reference Counting

Reference counting [Collins 1960] maintains a count of the number of references that exist to each heap allocated object. When there are no more references to an object, the reference count drops to zero and the space it occupies is identified as garbage. This is explicit identification: each piece of unused memory is explicitly known to be eligible for reclamation. Reference counting by itself is not able to identify cyclic garbage, which must be identified by some additional algorithm or leaked. In HHVM a 32 bit reference count is used, which introduces a memory overhead for every allocated object.

#### Limited Bit Reference Counting

In practice, the vast majority of objects do not exceed a very low reference count [Shahriyar et al. 2012]. This leads to interesting optimisations for reference counting schemes. Several optimisations to reference counting have introduced the notion of limited-bit reference counts, which provide a much lower maximum count. Objects that exceed the maximum count may have their counts continued in some additional data structure or they may simply 'stick' the reference count at its maximum value, ignoring all decrements. Such counts could optionally be restored during the object marking phase if a tracing collector was also present.

The most extreme form of this idea is a one-bit reference count [Wise and Friedman 1977], which simply represents that an object is currently singly-referenced, or that it is shared. This scheme is also referred to as a *multiple reference bit*, and was used in logic programming languages which sought to reduce unnecessary copying and allow incremental collection between tracing collections [Chikayama and Kimura 1987; Inamura et al. 1989; Nishida et al. 1990].

### Tracing

Tracing garbage collection [McCarthy 1960] periodically performs a transitive closure over the heap, marking every object that can be reached from a known set of roots, typically including global variables, the stack and registers. A second pass over the heap then releases all objects that are not marked, and are thus unreachable. This is implicit identification, given that only live objects are actually identified, and all other allocated space is considered to be dead.

Tracing garbage collection requires significant effort to identify all live objects at some point in the execution of the program. Simple tracing garbage collectors must 'stop the world' to perform their marking and sweeping passes over the heap, potentially introducing significant latency to the executing program.

### Generational Tracing

Generational tracing is an optimisation to tracing collection that takes advantage of the weak generational hypothesis which states that the majority of objects die young [Ungar 1984]. New objects are allocated into a nursery partition which is the most frequently collected partition. Surviving objects are copied to an older generation which requires less-frequent scanning, reducing overhead. Generational tracing pays the price of copying objects only in the rare case that they survive.

### Conservative Marking

During the marking phase of a tracing collector, all references reachable from a root set are followed and the referred object is marked as live. However, simply reading the contents of the stack does not provide the required type information to distinguish between references and values.

When scanning the stack for references into the heap, implementations that do not have sufficient type information about the stack must use conservative marking. Conservative marking involves looking at each potential reference on the stack and determining whether it could legitimately point to a heap object, in which case the object must be marked so that it is not inadvertently reclaimed. This technique is conservative because in some cases it may mark objects in the heap that were not actually referred to by a reference in the stack, but rather a program value in the stack could be interpreted as a reference to that object.

A further consequence of conservative marking is that objects in the heap that are marked conservatively cannot be moved by the collector during compaction or evacuation, because the pointers to that object would be updated, causing a change to the value in the stack that was conservatively considered to be a reference.

Stack maps are a solution that provide up-to-date type information for the entire stack, meaning that any potential reference on the stack is known to be a reference or a value, alleviating the need for conservative marking. In practice however, stack maps are very difficult to implement as they require significant cooperation from the

compiler, and research into the costs of conservative marking have found it to be a competitive strategy [Shahriyar et al. 2014].

### 2.2.3 Reclamation

#### Append to Free-list

In reference counting, objects are reclaimed individually as their reference counts drop to zero. The memory that they occupied is simply added to the appropriate free-list, ready for re-use by the allocator.

#### Sweep to Free-list

After a marking phase has occurred in a tracing collector, all objects that are not marked are reclaimed, and the memory for each is added to the appropriate free-list. This differs from the way in which naive reference counting uses a free-list in that all implicit garbage is added in one go, rather than explicitly and incrementally.

#### Evacuation

In evacuation, objects that survive the collection phase are moved into a separate heap space and re-allocated sequentially. At collection, the entire original heap space is reclaimed. This requires that the heap be partitioned into separate spaces, one of which cannot be used for allocation, as it must remain reserved for the evacuation phase, and requires that all objects can be moved.

#### Compacting

In compacting collection, surviving objects are typically scattered across the heap. In order to provide large, contiguous free space for subsequent allocation, surviving objects are moved next to each other in memory, eliminating *holes* within the heap. This differs to evacuation in that the moving occurs within the same area of the heap, eliminating the need for a reserved space.

## 2.3 Canonical Collectors

### 2.3.1 Naive Reference Counting

Naive reference counting garbage collection requires that the runtime keep an accurate record of exactly how many references exist to each heap-allocated object [Deutsch and Bobrow 1976]. Allocation is by means of a free-list allocator. Such schemes are easy to implement and provide a simple means for reclamation of unused space: it can simply be appended to the appropriate free-list. The typical life-cycle of an object is as follows:

1. An object is to be created, and space is requested from the free-list allocator.

2. The allocator finds an appropriate chunk of memory and returns the address to the object creation routine.

3. The data of the object is written into the provided space, and the object is created, typically with a reference count of one.

4. As this object is used and altered, its reference count is incremented or decremented as references to it are created and destroyed.

5. When the final reference is destroyed, the reference count is decremented to zero.

6. The memory that the object occupied is immediately returned to the free-list allocator, which appends it to an appropriate free-list, ready for re-use.

This strategy has the advantage that it is relatively simple to implement. The runtime must follow a particular discipline when copying references to ensure that the count is appropriately incremented or decremented, but the stored count gives immediate information about the object in a local scope, meaning that garbage identification can happen incrementally.

Naive reference counting is referred to as 'naive' because it does not take advantage of underlying properties of memory management that provide opportunities for optimisation. Reference counts are maintained precisely and for all objects. Analysis of a variety of programming languages and runtimes have shown that a large proportion of objects die young [Ungar 1984], and therefore only ever reach low reference counts. However, the cost of maintaining this count is still paid for every object. An exact reference counting algorithm pays the cost of maintenance in the extremely common case that an object does not require it. Free-lists also do not provide the side-by-side allocation of sequentially allocated objects, meaning that cache misses are typically more common [Shahriyar et al. 2013].

Reference counting by itself cannot identify cycles, which occur when two or more objects refer to each other. When all references into the cycle are removed, each object in the cycle still has a positive reference count, despite not being reachable from elsewhere. Reference counting implementations may either leak this cyclic garbage, employ a backup tracing collector, or use additional algorithms with the specific purpose of identifying cycles [Bacon and Rajan 2001].

As is discussed in Section 2.3.5, reference counting can be improved significantly if careful optimisations are made and if free-list allocation is replaced.

### 2.3.2 Mark-Sweep

The canonical mark-sweep garbage collector allocates and reclaims memory by use of a free-list, and identifies garbage by performing a trace over the heap [McCarthy 1960]. Mark-sweep collectors impose a low overhead at collection time due to their simple approach of an object marking phase followed by a sweep-to-free-list, but suffer poor locality as newly allocated objects are spread across the heap, not allocated

side-by-side. As objects are allocated into holes in the heap, mark-sweep collectors can suffer fragmentation leading to poor memory utilisation, unless compaction is used.

Mark-sweep is often used as a backup tracing algorithm for reference counting implementations to collect cycles as they both share the free-list heap layout, allowing for easier integration, but it can also perform all garbage collection without the aid of reference counting.

### 2.3.3 Semi-Space

The simplest semi-space collectors use bump-pointer allocation into a block of memory that represents half of the total available heap space [Cheney 1970]. At collection time, a trace is performed and all live objects identified, and each live object is evacuated to the empty second half of the heap and re-allocated contiguously. After the collection has completed, bump-pointer allocation continues in the second half of the heap, beginning after the last evacuated objects. This provides excellent locality because new objects are allocated together, and older objects are moved next to their contemporaries. It also fundamentally limits the usable heap size because it sets half aside for evacuation. Copying operations can also be expensive if there are a large number of live objects, but this is uncommon.

### 2.3.4 Mark-Compact

Mark-compact collectors use the bump-pointer allocation seen in semi-space, but make use of the full heap size when doing so [Stygar 1967]. After the trace is performed and live objects are identified, several additional passes over the heap arrange for surviving objects to be moved to the start of the heap and re-allocated contiguously. Mark-compact relies on moving objects within the heap, rather than moving them to a new space as in semi-space, but this causes additional passes over the heap which introduces increased overhead at collection time, despite the benefits to locality and decrease in fragmentation.

### 2.3.5 Immix

The Immix collector [Blackburn and McKinley 2008] addresses the issues of collection speed, mutator performance and space efficiency by introducing a class of collectors known as mark-region. The Immix collector segregates memory into blocks and lines, allowing bump-pointer allocation with reclamation at either the line or block level, but not at the level of individual objects (Figure 2.1). During the marking phase, lines and blocks are marked in addition to individual objects and at reclamation time, blocks are either totally reclaimed if no lines are marked, or noted as recyclable, meaning free lines are present for allocation.

Immix combines this heap layout with opportunistic copying during the marking phase, using a set of heuristics to detect fragmented blocks and possible destinations for evacuated objects.

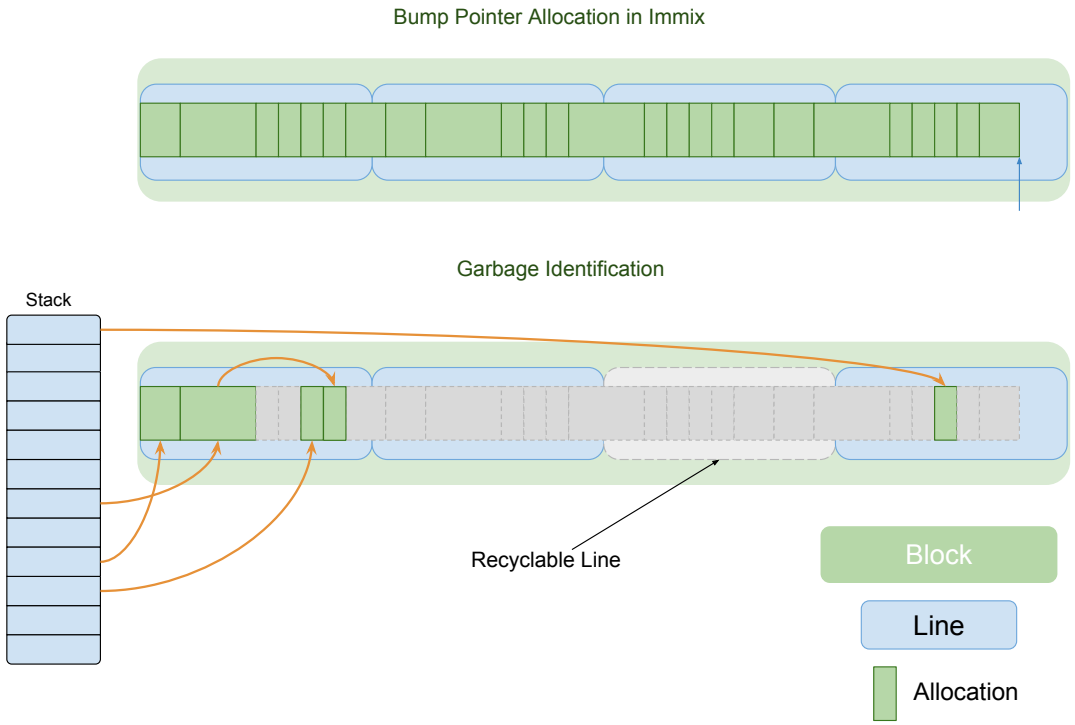


**Figure 2.1:** Immix segregates the heap into lines and blocks, allocating new objects into empty or partially free blocks and reclaiming at the line or block level.

### 2.3.6 RC Immix

RC Immix combines highly optimised reference counting with the region layout of Immix [Shahriyar et al. 2013]. By taking advantage of the underlying properties of reference counting, RC Immix reduces the runtime overhead of maintaining reference counts. RC Immix differs from other reference counting schemes in that it does not use free-list allocation. The Immix line and block layout is combined with per-line reference counts, resulting in a high performance reference counting implementation that is the fastest garbage collector in the literature.

## 2.4 The PHP Language

PHP is a dynamic scripting language largely used for website development. PHP began as a personal project for dynamically creating sections of HTML within a web page, and grew over a period of years into a programming language that became widely used across the internet. In this respect, PHP had an unusual beginning in that it was not originally designed as a programming language, and many of the design decisions made at this early stage have affected the evolution of PHP over the years. An example is the use of naive reference counting, which is often chosen due to its relative ease of implementation, compared to far more complex, but better performing tracing garbage collectors.

For much of its life, PHP has operated without a language specification, relying on the Zend PHP implementation as a de-facto standard. Zend PHP was the only widely used PHP virtual machine prior to the creation of HHVM (Section 2.5), which is an oddity compared to similar dynamic languages which have multiple, largely cross-compatible implementations. Examples include Python which is implemented by CPython, Jython (JVM) and PyPy (meta-circular Python), and Ruby, implemented by MRI (de-facto), JRuby (JVM) and Rubinius (meta-circular Ruby).

Zend PHP also offers a large selection of extensions that provide additional features such as interaction with various types of databases and libraries to support data formats such as XML, JSON and YAML. These extensions are in wide use across existing PHP projects, making the task of creating a compatible PHP virtual machine more difficult, as it would have to also offer support for these extensions.

Several decisions regarding the design of the PHP language have had perhaps unintended consequences in restricting virtual machine design.

### 2.4.1 Value Semantics

The Zend PHP engine defines pass-by-value semantics by default for all types within PHP. Objects have their reference passed by value, but strings and arrays are passed directly by value, suggesting a new copy of the data is present inside the function. When an array or string is passed to a function as a parameter, that function receives its own local copy which it can modify, and those changes will not be visible in the

calling scope (Figure 2.2). Other languages such as Ruby, Python and Java do not offer these semantics.

A naive implementation may simply copy every array or string that is passed as a parameter, however this could significantly slow the performance of the mutator, particularly with large data structures. In some number of cases, the data would only ever be read inside the function, but not modified, meaning that the copy was never actually needed.

```
function reveal($arr) {
  $arr[1] = "Targaryen"; // copy-on-write occurs
}

$words = ["Jon", "Snow"];
reveal($words);

echo $words[1]; // prints "Snow";
```

**Figure 2.2:** PHP pass-by-value semantics. The reveal() function can modify the array, but this is not visible outside the function scope because the array has been copied.

This introduces the possibility for a deferred copy mechanism. The guiding principle of this optimisation is that not all structures that are passed by value need to be copied, only those that are modified, and only in certain circumstances. A sensible solution would be to provide a reference to the original data in the calling scope until such time that it would be modified inside the function, and then lazily copy the array, modifying the new copy instead. This would be indistinguishable to the programmer, but would save copying as often, ultimately reducing computational work. This scheme where the copy is delayed until write is fittingly referred to as *copy-on-write*, and this variant where every array is copied at write time is referred to as *blind copy-on-write* [Sergeant 2014].

The above scheme can be improved further, however. There may exist cases where the modification to the data inside the function would not be visible outside of the function, even when a reference to the original array, not a value-copy is used. If the runtime can dynamically determine that the array is not referenced elsewhere, then the copy at write time can be avoided completely. In practice it would be very difficult to determine whether an array was referenced elsewhere in the runtime on request, unless such a statistic was constantly maintained, as it is with naive reference counting.

If an array has a reference count of one, then the runtime can determine that modification inside the function will not be visible in any other scope, even if the original array is modified in-place, rather than copied. Zend PHP makes use of this insight, and checks the reference count prior to copying an array inside a copy-on-write check. This particular optimisation is a design choice of the implementation, and because it does not affect semantics, it is not required by the language definition itself. However as Chapter 4 shows, removing this optimisation can have drastic effects on perfor-

mance.

By utilising reference counts for the copy-on-write optimisation, Zend PHP has tied its performance in implementing its parameter passing semantics to its garbage collection strategy, which are two separate concerns. Because of this, PHP strongly implies the use of reference counting garbage collection because without accurate reference counts, performance suffers.

### 2.4.2 Object Destructors

Zend PHP, and by extension, the PHP language, also provides constructors and destructors for objects. A destructor is a user-defined method associated with an object class, typically used to perform 'clean-up' actions such as to release resources that were acquired by that instance. The PHP specification defines when destructors are called as follows:

> The destructors for instances of all classes are called automatically once there are no handles pointing to those instances or in some unspecified order during program shutdown. [The PHP Group 2015]

In Zend PHP, the implementation of destructors follows the first option above, and calls the destructor method when no other references to an object exist, again using reference counting to make this determination. Because of the determinism in object lifetime that reference counting provides, destructors in PHP are sometimes treated as guaranteed to be run at a particular time (Figure 2.3). This may be compounded by the fact that other languages that provide user-defined destructors, for example C++, do guarantee deterministic calling of these methods [ISO/IEC 2014].

```
class DBWriter {
  Database db;
  function __construct() {
    db = Database::getDB();
    db.lock();
  }
  function __destruct() {
    db.unlock(); // release resource in destructor, RAII-style.
  }
}
```

**Figure 2.3:** PHP destructors are sometimes used to free resources, but this isn't guaranteed.

The definition of PHP destructors could be considered much closer to the concept of a finalizer in Java, which makes no guarantee in relation to when the method will be called, from the perspective of the programmer [Gosling et al. 2014]. While PHP does not explicitly bind implementations to using reference counting in this instance, in practice existing PHP applications may make use of destructor behaviour as if it

were deterministic. Any changes to the garbage collection used in PHP must consider this issue.

### 2.4.3 Reference Variable Semantics

PHP provides pass-by-value semantics by default but also allows reference assignment using the &= operator as well as argument passing and value returning by reference. The way in which this language feature has been designed once again encourages the use of reference counting.

When a variable is created, the underlying representation is stored as a value. If this value becomes shared between two or more variables due to a reference-assignment operation, it is then considered to be a reference. When variables that share the reference are removed and there is only one referring variable remaining, it is demoted back to a value. In Zend PHP a flag called `is_ref` is used to explicitly track the reference status of a value. HHVM represents references as a separate data structure `RefData`. Both of these methods explicitly track whether the value is a reference using the current reference count.

These reference variable semantics become visible in PHP when combined with array copying. In PHP it is possible to create a reference to a value that is an element of an array, sharing that value. When an array is copied (which, semantically, happens at every assignment operation to an array) the elements that are references are not copied by value, but instead re-created in the new array as another sharing of the same reference. Any changes to the referenced value in one array are visible in the other, consistent with the element being shared, but this produces visible behaviour that relies on reference counting.

```
$original = ["A", "B", "C"];
$value_ref =& $original[1]; // "B" is now shared
$copy = $original; // "B" is shared between three

$copy[1] = "Z";
// $original[1] == $value_ref == $copy[1] == "B"
```

**Figure 2.4:** A reference inside an array is not copied by value and remains shared.

Figure 2.4 shows an example that will work regardless of reference counting because no reference demotion is performed. Figure 2.5 is almost identical, except that `unset()` should demote `"B"` to a value, causing it to be copied by value when the array is copied. Zend and HHVM both rely on reference counting to produce this behaviour.

### 2.4.4 Summary

The direction of the PHP language has been largely influenced by the de-facto implementation, Zend PHP, and this has resulted in the design of language features that are

```php
$original = ["A", "B", "C"];
$value_ref =& $original[1]; // "B" is now shared
unset($value_ref); // "B" should be demoted to a value
$copy = $original; // "B" is should be copied by value

$copy[1] = "Z";
// $original[1] == "B"
// $copy[1] == "Z"
```

**Figure 2.5:** The reference should be demoted to a value before the array copy occurs, relying on reference counting.

most easily implemented with the use of reference counting. An important conclusion, however is that none of the issues raised definitively require PHP to use reference counting. Copy-on-write is an implementation optimisation to achieve efficient value semantics, but this does not mean it is required in practice. Destructors, while sometimes used as though deterministic guarantees about calling times are present, do not require this guarantee. Finally, it is possible to duplicate Zend's reference variable behaviour by other means, or simply implement slightly different semantics for this unusual language feature. This conclusion suggests that PHP is not inherently incompatible with garbage collection that does not use naive reference counting.

## 2.5 The HipHop Virtual Machine

HHVM is an open source virtual machine created and maintained by Facebook. HHVM serves as an implementation for PHP as well as Hack, Facebook's own language based on PHP which introduces static typing, generics, support for asynchronous functions and several other new language features, whilst maintaining backward compatibility with PHP.

Facebook runs one of the world's largest websites, which is written in PHP and Hack. Server operating costs for supporting a user base of approximately 1.5 billion are astonishing, and Facebook has entire teams dedicated to reducing these costs. In addition, the scalability and stability of their infrastructure plays a key role in keeping such a large site running. In 2010, Facebook open sourced HPHPc, a PHP to C++ transpiler aimed at achieving greater efficiency than Zend PHP. It was successful in this goal but was replaced in 2013 by the new HHVM, which featured a just-in-time (JIT) compiler allowing further performance gains.

HHVM is under active development and continues to make improvements in both CPU and memory efficiency. Even small improvements can result in large savings in cost to Facebook.

### 2.5.1 Garbage Collection

HHVM uses naive reference counting for automatic memory management. This allows it to achieve a high level of compatibility with Zend PHP and provides a simple path for the implementation of copy-on-write using reference counts, as well as fully compatible destructor and reference variable behaviour.

HHVM allocates objects using a *reap* (region-heap) which imposes free-list structures over regions of memory. This allows individual objects to be reclaimed while supporting typical operations that apply to regions, such as the ability to reclaim large chunks of contiguous memory simultaneously. At the end of a request, HHVM simply reclaims all regions in use, freeing any objects which were not incrementally collected. Bump-pointer allocation is used to fill new regions, which are requested if the size-segregated free-lists cannot fulfill the allocation request. This still results in free-list allocation begin used in the common case, which is known to result in fragmentation and increase cache misses [Shahriyar et al. 2013].

While the state-of-the-art RC Immix garbage collector does make use of reference counts, this is a highly optimised implementation, and does not rely on free-list allocation [Shahriyar et al. 2013]. Due to the high impact that garbage collection can have on program efficiency, research into better garbage collection strategies for HHVM is justified.

HHVM has recently added a backup mark-sweep collector for use in collecting cycles and as a basis for future garbage collection work, although this is experimental and not enabled by default. HHVM does not implement stack maps, so conservative marking is used for the stack and some other roots, as well as for some heap objects until functions to scan every possible heap object for references are complete.

### 2.5.2 Reference Count Optimisations

The HHVM JIT performs some reference count optimisation in an attempt to reduce the overhead caused by unnecessary operations [Facebook Inc. 2015]. This analysis attempts to find matching pairs of *incRef* and *decRef* operations between which no events happen that could rely on the incRef having occurred. These pairs can then be safely removed, as the increased reference count would only be used to protect against destructive overwriting of the object during a mutation. It also attempts to weaken *decRef* operations to *decRefNZ* operations, which are used when it is known that the count cannot reach zero, saving having to check this at runtime for destructor or garbage collection behaviour.

## 2.6 Related Work

### 2.6.1 Garbage Collection in PHP

Sergeant [2014] measured the memory characteristics of HHVM and found that the demographics of heap objects in PHP are suited to tracing garbage collectors such as Immix. Sergeant also proposed and evaluated blind copy-on-write, which was found

to cause unacceptable performance loss, and began work on a mark-region garbage collector. Chapter 4 of this thesis builds on these findings and introduces the one-bit reference count in order to maintain copy-on-write in HHVM. Chapter 5 continues this by analysing a new prototype collector based on Immix which is able to run several major PHP frameworks. This collector is a complete implementation that is able to collect all garbage, and also fully implements conservative heap scanning and novel optimisations for explicitly managed allocations, which has not been completed previously.

### 2.6.2 Copy-on-write

Tozawa, Tatsubori, Onodera, and Minamide [2009] analysed the impact of copy-on-write on language semantics in PHP and found that copy-on-write actually caused PHP to be inconsistent with copy-on-assignment/pass-by-value semantics. This work provides extensive background and analysis of the semantics and further reinforces PHP's peculiar reliance on reference counting, and highlights the difficulty of implementing a new runtime for an implementation-defined language.

## 2.7 Summary

This chapter introduced the topic of memory management and motivated its importance, detailing concepts and algorithms which will prove important in Chapters 4 and 5. It also introduced the PHP language, describing how early design decisions have shaped garbage collection strategies, providing the required motivation for Chapter 4, and HHVM was introduced as the proving ground for this work. Chapter 3 will first provide an outline of the experimental methodology before Chapters 4 and 5 detail the main contributions of this thesis.

Chapter 3

# Experimental Methodology

This chapter outlines the components and configuration used to carry out the experiments present in this thesis, presenting the motivation behind these choices where applicable.

## 3.1 Benchmarks

The HHVM project uses the *hhvm/oss-performance* suite of open source application benchmarks that are designed to provide a realistic macro-benchmark framework for measuring the performance of PHP engines [Bissonnette 2015]. This thesis uses results from WordPress, MediaWiki and Drupal7 as they are the most realistic benchmarks included in the suite. These applications apply typical usage patterns of PHP in interacting with databases and dynamically creating HTML.

This suite creates a HHVM process connected to a web server and uses Siege (a HTTP load testing utility) to bombard the server with requests, simulating 200 users as an analogy for heavy traffic. Under these benchmarks HHVM is configured to sensible defaults for production use, including the use of Repo-Authoritative mode which caches the PHP bytecode derived from PHP source code rather than re-creating it for each request. Warm-up runs are performed by the suite prior to the measured run in order to warm up the JIT compiler and accurately simulate a web server in a steady state. The suite performs a timed test of 60 seconds and measures the number of requests that successfully complete, measuring the result in requests per second (RPS).

The suite's batch run facility is used to run the three frameworks in succession with each different build being benchmarked. The figures throughout this thesis present the mean result from ten batch runs for each experiment.

Facebook has used these benchmarks to measure improvements to HHVM and they are the standard performance measurement for open source contributors to HHVM. Facebook also measures HHVM's performance improvements internally using a closed-source benchmark based on www.facebook.com.

## 3.2 Software

The benchmarking operating system is 64 bit Ubuntu 14.04 Server with kernel version 3.13.0-32. *hhvm/oss-performance* makes use of Siege (2.70), Nginx (1.4.6) and MySQL (5.5.41).

## 3.3 Hardware

The benchmark machine has a 3.40 GHz Intel Core i7 2600 (Sandy Bridge) processor, which has four cores with hyper-threading, one 32 KB L1 data cache and one 32KB L1 instruction cache per core, a shared 1 MB L2 cache and a shared 8 MB L3 cache. The machine has 4 GB RAM.

Chapter 4

# Overcoming Obstacles to Tracing Garbage Collection

Chapter 2 introduced memory management as well as the language (PHP) and experimental platform (HHVM) for this thesis. This Chapter outlines the ways in which PHP is tied to naive reference counting as a garbage collection strategy and proposes solutions to these barriers to further garbage collection work, providing both implementation and analysis.

Section 4.1 details the ways in which the PHP language relies on naive reference counting. Section 4.2 explores the design space for possible solutions that allow PHP to maintain performance with reference counting removed. Section 4.3 proposes a one-bit reference count as a solution and presents the analysis undertaken to show that this is an effective strategy. Section 4.4 discusses differences between the proposed behaviour that the new one-bit scheme introduced and the existing behaviour within the context of the PHP specification.

## 4.1 Obstacles in PHP

The two major PHP implementations, Zend PHP and HHVM, both use naive reference counting for garbage collection. The PHP language itself heavily encourages the use of reference counting, and relies on it for performance and correctness, making the removal of reference counting a difficult task.

### 4.1.1 Copy-on-Write

As discussed in Section 2.4.1, the pass-by-value semantics for arrays and strings in PHP rely on copy-on-write for efficient implementation in Zend PHP and HHVM. A naive implementation of pass-by-value might make a full copy of an array or string at every assignment statement or function call (*copy-on-assignment*), but this would cause severe performance degradation and require a much larger heap. Copy-on-write stems from the observation that eager copying is often unnecessary to provide value semantics, because if the array is not modified or modifications are not visible elsewhere in the runtime, the copy can be safely ignored.

In practice this suggests that implementations should attempt to delay the copying of arrays and strings passed by value until it is absolutely necessary, or should be able to analyse code to determine which copies will be necessary. Strategies for delaying copies until necessary are referred to as *deferred-copy mechanisms*.

HHVM and Zend PHP use copy-on-write as a deferred-copy mechanism, using the reference count of the array or string inside the check to determine whether a copy is necessary. This behaviour relies on the presence of exact reference counts, which is incompatible with high performance garbage collection.

### 4.1.2 Precise Destruction

PHP provides destructors for object types, which allow user code to be run when the object is reclaimed by the memory manager. In Zend PHP and HHVM, destructors are run when an object's reference count is decremented to zero, which is deterministic from the perspective of the programmer. If PHP does not use reference counting garbage collection, this behaviour cannot be easily guaranteed.

The PHP specification does not require that destructors are run immediately when an object is no longer reachable, yet some applications written in PHP rely on this behaviour for their own correctness. Unlike copy-on-write, which is purely an optimisation, changes to the timing of destructors is visible to PHP code.

This thesis does not explore options for replicating the precise destruction behaviour of Zend PHP and HHVM without reference counting. The PHP specification only requires that destructors are run before the end of the request, which can be achieved easily using tracing garbage collection. Section 4.4 discusses the impact this has on common PHP frameworks and explores how it can be resolved.

### 4.1.3 Precise Demotion of References

When a value in PHP becomes shared by two or more references due to reference-assignment or pass-by-reference, it becomes a *reference type*. When a reference type is no longer shared between multiple references, it is *demoted* back to a value type. Zend PHP and HHVM both implement this behaviour using naive reference counting, which provides the exact number of references to any object.

This behaviour is visible to PHP code because reference types that are members of an array are not copied by value, but rather re-shared when the array is copied. HHVM lazily demotes reference types when copying arrays if the reference count is one. This behaviour is difficult to replicate without reference counting, because references to a reference type are not necessarily bound to the same scope and could be anywhere in the runtime, meaning that it would be extremely difficult to track these down on demand. These peculiar semantics are unique to PHP, and have been analysed in detail previously [Tozawa et al. 2009]. Other languages avoid such semantics due to the implementation headache they can introduce, as exemplified by PHP.

## 4.2 Solution Design Space

The are a few ways in which PHP relies on reference counting, but copy-on-write is the most problematic, due to the performance impact that it causes. This section explores possible solutions. Providing pass-by-value semantics for dynamically sized and potentially large data structures such as strings and arrays is not a design issue limited to PHP. Several approaches have been used in other languages and a consideration of each is provided.

### 4.2.1 Pass References By Value

Most other languages do not pass arrays directly by value, but instead pass a reference to the array by value, most likely to avoid the exact issues this causes for PHP. In Java, Python, JavaScript, C# and many more, object references are passed by value. The function receives a local copy of the object reference, and contents of the data structure itself are not copied. By following the object reference, the internals of the data structure passed to a function can be modified and the results of this are visible outside of the function scope. Tracing garbage collection is used in mainstream implementations of each of these languages.

This particular passing semantic is the default in the above languages and does not require copying underlying data, saving time and space. Unfortunately, PHP defines pass-by-value semantics for arrays and strings, and changes to this specification are not within the scope of this thesis, as they would likely render most existing PHP code incorrect. Any suitable solution must maintain pass-by-value semantics.

### 4.2.2 Specialised Data Structure Support

The motivation behind copying data structures to provide pass-by-value is that a function must be able to modify the data structure in a way that is not observable outside of that function scope. The simplest way to provide this behaviour is to provide a new copy of the structure which can be modified, but other approaches also exist.

*Ropes* [Boehm et al. 1995] are one such example for strings which provides an alternative implementation that does not suffer the cost of recreating the entire string in memory to modify it. Ropes also provide other operations that can avoid copies where they would normally be required, such as string concatenation, and are particularly effective for operations on large strings.

Ropes achieve different performance characteristics by representing strings as an ordered tree of smaller strings with each node representing the concatenation of its children (Figure 4.1). The leaf nodes are the traditional string structure of an array of contiguously allocated characters. HHVM does not implement any such data structure and to introduce Ropes may involve considerable engineering effort, however they still provide a promising path for exploration.

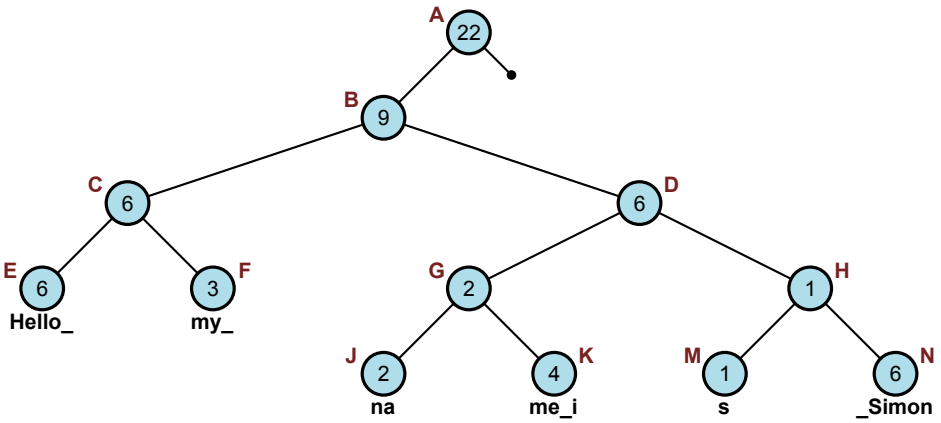


**Figure 4.1:** A Rope representation of the string "Hello my name is Simon". The weightings for leaf nodes indicate the number of characters at that leaf, and the weightings for inner nodes represent the sum of weightings of the leaf nodes in the left sub-tree [Yao 2013].

### 4.2.3 Blind Copy-On-Write

A simple solution that does not involve reference counting is to always copy the data structure when a mutation occurs. Mutating operations are no longer required to perform a check to see whether the reference count is above one, so the reference count is no longer tied to copy-on-write behaviour, allowing far greater choice in garbage collection strategies. This approach has been analysed previously by Sergeant [2014], and found to cause unacceptable performance loss. These results are replicated for the sake of completeness in Section 4.3.2.

### 4.2.4 Static Analysis

Copy-on-write attempts to ensure that pass-by-value semantics are maintained by lazily copying whenever a modification might have been observed through a different handle to the underlying memory. One optimisation copy-on-write does not consider is that it may not be the case that a modification is actually observed, just because it is possible to be observed.

Static analysis of PHP code at compile-time may provide enough information about the usage of these temporarily shared data structures to remove copying where it is not required and to insert it where it is needed for correctness. However, building such a tool could prove quite difficult, and it may not be possible to accurately estimate the effect on performance until it is complete.

### 4.2.5 Improved Reference Counting

Reference counting garbage collection is not without benefits. The local scope of operation and immediacy of reclamation are two benefits often cited, however a naive

implementation cannot compete with tracing garbage collection. Improvements and optimisations to reference counting have been considered and analysed for some time, and it may be simpler to modify the existing reference counting implementation which has already seen the investment of considerable engineering effort. Potential optimisations include those proposed by RC Immix [Shahriyar et al. 2013] and more extreme schemes, such as the one-bit reference count.

Many optimisations to reference counting involve leveraging the fact that completely accurate reference counts are not typically needed at all points in a program. This allows temporarily incorrect reference counts which are later corrected or accounted for. Because PHP does make use of the reference counts during copy-on-write, extreme care would be required to ensure that reference counts are at least conservatively high - a reference count that is too low could lead to incorrect behaviour.

## 4.3 One-bit Reference Count

This thesis explores the use of a one-bit reference count in HHVM. This is chosen due to the simplicity of its implementation given the existing reference counting scheme, and due to properties of the runtime and common PHP programs that make it an attractive scheme.

### 4.3.1 Prototype Design

A one-bit reference count has two states, representing that the object is *unshared* or *shared*. Conceptually, this is identical to a limited-bit reference count that 'sticks' at a value of two. Objects are born unshared with their bit set to 0. Note that there is no representation of a zero reference count in this scheme - similarly in most reference counting implementations, objects are born with a reference count of one, and when the count would be decremented to zero, the object is instead reclaimed, never actually reaching zero and saving the cost of the decrement operation.

If the object were to have its reference count incremented, it must be to a value of two or higher, indicating that the object has become shared, so the bit is unconditionally set when a new reference to the object is created. Once an object has become shared, any further increments will have no effect, but will still result in an unconditional set of the bit. Because it is not possible to track how many references there are once the bit has been set, the one-bit scheme simply ignores decrement operations to a shared object.

When an object is mutated and a copy-on-write check is performed, instead of consulting the reference count, HHVM will check the one-bit reference count and copy if the object is marked as shared. The nature of one-bit reference counts mean that this scheme must perform the same or a greater number of copies, due to the inaccuracy introduced by losing track of reference counts that exceed a value of one.

This scheme can be implemented using the existing reference counting operations within HHVM by transforming all increments to unconditional sets of the bit and removing all decrements that occur to objects that may potentially be shared. This

scheme introduces one operation, *setShared*, which sets the bit. All existing *incRef* operations are replaced with *setShared*, and all existing *decRef* operations are removed. The rationale for this design is to a) save space in the object header by reducing the number of bits from 32 to 1, and b) remove operations from the hot path by unconditionally setting the bit.

To analyse what sort of effect the one-bit reference count might have on performance, it is important to understand how many copy-on-write checks occur and what proportion of these result in a copy under reference counting compared to the proposed scheme.

### 4.3.2 Prototype Results

The builds of HHVM used throughout this section implement one-bit RC alongside existing reference counting to ensure correctness and allow perform comparative analysis The changes required to implement the one-bit reference count also provided a simple way to analyse blind copy-on-write, the results of which are provided here as a point of comparison.

The results presented use the macro-benchmark suite which contains several common open source PHP applications in order to gauge the systemic effects of these changes. Several micro-benchmarks were initially analysed, but copy-on-write performance impacts are typically non-existent or completely dominate the results, depending on how the benchmark was written. These results are not significant in showing the typical use of PHP as a web server language and are not idiomatic PHP, and are not presented.

#### Statistical Results

Use of a one-bit count to inform copy-on-write checks drastically reduces the number of copies compared to blind copy-on-write, which results in a copy at every check. One-bit RC should perform as well as RC in the best case and as poorly as blind-cow in the worst case. Any array or string which is currently shared between multiple references will be copied under one-bit RC, just like regular reference counting. On top of this, in the case that the object was shared and unshared and would have a reference count of one, one-bit RC will copy unnecessarily because it cannot distinguish these from currently shared objects. This proves to be relatively uncommon compared to the case where an object is still shared at the time of the check.

Table 4.1 shows the results of the statistical analysis. On average, the number of arrays copied under one-bit reference counting increases in the range of 21%-36% compared to the existing reference counting implementation, and the number of strings copied increases in the range of 4%-10%. This compares extremely well to blind copy-on-write, which increases by up to 13x and 2x respectively.

**Table 4.1:** Number of copy-on-write checks resulting in a copy, normalised to RC

| Application | RC | One-bit RC | Blind-CoW |
|---|---|---|---|
| Wordpress | | | |
| *array* | 1.0 | 1.30 | 13.22 |
| *string* | 1.0 | 1.04 | 1.54 |
| Mediawiki | | | |
| *array* | 1.0 | 1.36 | 10.04 |
| *string* | 1.0 | 1.11 | 1.86 |
| Drupal7 | | | |
| *array* | 1.0 | 1.21 | 5.45 |
| *string* | 1.0 | 1.07 | 2.00 |

**Table 4.2:** Requests per second, normalised to maintain one-bit

| Application | RC | One-bit RC | Blind-CoW |
|---|---|---|---|
| Wordpress | 1.0 | 0.981 | 0.626 |
| Mediawiki | 1.0 | 0.986 | 0.161 |
| Drupal7 | 1.0 | 0.989 | 0.845 |

**Table 4.3:** Total heap size of copied objects, normalised to RC

| Application | RC | One-bit RC | Blind-CoW |
|---|---|---|---|
| Wordpress | 1.0 | 10.8 | 1898.1 |
| Mediawiki | 1.0 | 2.4 | 7462.1 |
| Drupal7 | 1.0 | 1.9 | 26.3 |

**Performance Results**

Given that all garbage collection in PHP relies on reference counting, the builds presented here for analysis still have reference counting in place and use it to collect garbage, due to the large performance impact that removing it would incur. To properly assess the impact that one-bit RC has on performance, results are normalised to a build which maintains the one-bit count but does not use it, so that just the performance impact caused by copying behaviour is measured. The results presented in Table 4.2 confirm previous work that suggests blind copy-on-write causes unacceptable performance loss, degrading total performance by more than 80% in the case of Mediawiki. One-bit reference counting shows extremely promising results, only degrading performance due to additional copying by 2% in the worst case.

To further explain these results, total heap size of all objects copied during copy-on-write checks is presented in Table 4.3. Table 4.1 shows the raw number of copies that occur but does not account for the size of the data structures. The increase in total heap size of copied objects is far more drastic in blind-cow than the increase in the raw number of objects. This analysis better explains the performance results of blind copy-on-write, which must copy every array and string, and shows that the total heap size of copied data can increase by up to 7500x under blind copy-on-write.

### 4.3.3 Prototype Analysis

Maintaining reference counts can impose a significant overhead and this has motivated the removal of reference counts from HHVM. Introducing one-bit counts must still incur some overhead, and measuring this impact is crucial to its analysis.

The results in the previous section show the impact that the additional copying of arrays and strings has on performance for the open source applications, but the build used still maintains full reference counts which it uses for garbage collection. In order to fairly evaluate HHVM with reference counts removed, some garbage collection must be present so that benchmarks do not run out of memory, and so that systemic effects on locality are accounted for.

*no-rc* is a build of HHVM which uses the one-bit reference counts for garbage collection, in order to measure the performance of one-bit RC without a significant overhead due to absent garbage collection. The build is still able incrementally release objects that do not have their bit set. Given that most objects are never shared and never have their bit set, it is possible to reclaim most unreachable objects, although shared objects will be leaked. This can be normalised to a build of HHVM which incurs the same maintenance overhead as reference counting, but behaves like the one-bit scheme, *double-inc*.

*Double-inc* is identical to HHVM master except that *incRef* increases the reference count by two instead of by one, while still decrementing by one. This has the effect of emulating the behaviour of a one-bit reference count while using almost identical instructions to a reference counting scheme. Objects are born with a reference count of one and all increments are by two. Decrement operations are not changed, meaning that any object that is ever incremented will never reach zero and thus never be

reclaimed. This produces a build which has the same maintenance costs as reference counting, but the garbage collection and copy-on-write behaviour of *no-rc*.

The performance of *double-inc* should match *no-rc* if the overhead of maintaining the one-bit count is equal to the overhead of maintaining full reference counts. The difference between *double-inc* and *no-rc* is therefore the difference in the cost of maintaining full reference counts compared to a one-bit count.

Table 4.4 compares these two builds. *no-rc* performs better across the benchmark suite by a range of 0.9-1.4% showing that one-bit counts are cheaper to maintain than full reference counts.

**Table 4.4:** Requests per second, normalised to master

| Application | RC | No-RC | Double-Inc |
|---|---|---|---|
| Wordpress | 1.0 | 0.960 | 0.949 |
| Mediawiki | 1.0 | 0.981 | 0.967 |
| Drupal7 | 1.0 | 0.972 | 0.963 |

### 4.3.4 Discussion

Reference counting builds of HHVM have precise information regarding sharing when performing copy-on-write checks, minimising the number of copies required to maintain pass-by-value semantics. Blind copy-on-write builds however are extremely uninformed, and have no extra information regarding the array to be copied, and so always elect to copy, in the case than the array may be shared. This results in blind copy-on-write being too conservative in its categorisation of objects, as additional copies are acceptable in terms of correctness. A build that can use better heuristics during a copy-on-write check can therefore be less conservative, and alleviate the performance impact of removing reference counting.

Ideally, the number of additional copies should be as small as possible to avoid incurring too high a performance penalty, otherwise any further work to improve garbage collection will not actually result in a net improvement. One-bit RC is promising; an optimal scheme would see a 0% increase, but would likely require more than one bit of memory and greater runtime overhead in order to maintain it. One-bit RC provides a compromise that allows the copy-on-write optimisation to continue to reduce the overhead associated with maintaining pass-by-value semantics while helping to reduce the cost of maintaining reference counts.

The results presented are based on a simple implementation of one-bit RC that replaces reference counting operations one-for-one with one-bit RC operations. This does not take into account the way in which the rest of HHVM has been designed around reference counting. The presence of reference counting is a pervasive assumption throughout HHVM, and because much of the code must deal directly with reference counting operations, this cannot be easily avoided. With a much simpler one-bit RC scheme in place, design choices that assume reference counting is present could

be revisited, particularly in the JIT reference counting optimisation pass, resulting in even better performance of one-bit RC.

## 4.4 Behaviour, Semantics and the PHP Specification

### Precise Destruction

The PHP language specification only makes the guarantee that destructors will be run during the shutdown sequence of the virtual machine, which means that the one-bit RC build is compatible with the specification as long as destructors are run during shutdown. In practice, some existing PHP code has come to rely on destructors being run at a precise time or in a defined order. To examine this behaviour, *no-rc* is used to run the tests suites of the open source applications used for benchmarking.

The Wordpress and Drupal7 applications are able to run their benchmarks on this build without issue and pass their unit tests, however Mediawiki cannot. Mediawiki uses a class called *ScopedCallback* which allows the creator to register a callback to be run in the destructor of the *ScopedCallback* object. These objects are typically used as local variables inside a function to ensure that a lock or resource is released when the function scope is exited or if an exception is thrown. Figure 4.2 shows an example of Mediawiki code which relies on precise destruction.

```
$acquired = $this->mMemc->add(
        $statusKey, 'loading', MSG_LOAD_TIMEOUT );
  if ( $acquired ) {
  # Unlock the status key if there is an exception
  $statusUnlocker = new ScopedCallback(
  function () use ( $this, $statusKey ) {
      $this->mMemc->delete( $statusKey );
  } );
  ...
  # Unlock
  ScopedCallback::consume( $statusUnlocker );
```

**Figure 4.2:** Mediawiki relies on precise destruction to ensure resources are released in the case of exceptions being raised.

While this code cannot be guaranteed to be correct when using one-bit RC, it can be easily modified to produce the same behaviour in a way that is supported by the specification, as seen in Figure 4.3. Finally blocks were introduced in PHP 5.5 and because precise destruction times are typically relied upon for resource unlocking or similar behaviour, they provide a convenient replacement that is guaranteed to achieve the intended result.

```
try {
  $acquired = $this->mMemc->add(
        $statusKey, 'loading', MSG_LOAD_TIMEOUT );
  if ( $acquired ) {
    ...
  }
} finally {
  # Unlock the status key every time, even if there is an exception
  $this->mMemc->delete( $statusKey );
}
```

**Figure 4.3:** This modified version of the code removes the reliance on precise destruction with minor re-factoring. The finally construct is used to ensure that clean-up code is always run, even in the event of an exception. This is guaranteed by the specification, and behaves in the same way.

#### Precise demotion of references

One-bit RC does not provide any way to demote references to values, and can therefore display different copying behaviour to Zend PHP and HHVM without modifications. This could impact the correctness of programs, but Wordpress, Drupal7 and Mediawiki are not affected by this change in semantics. Allowing sharing of array elements through reference assignment is a peculiar semantic which PHP provides, although in practice it is likely that it is rarely relied upon.

## 4.5 Further Opportunities

### 4.5.1 Use in garbage collection

In addition to providing information regarding the number of references to an object inside copy-on-write checks, one-bit RC has also been used in conjunction with mark-sweep garbage collectors to free objects outside of a garbage collection sweep [Chikayama and Kimura 1987]. This allows reclamation of objects between collections that are not shared and therefore don't have their bit set. One-bit reference counts could potentially be used in the garbage collection strategy, which is further explored in Chapter 5.

### 4.5.2 Reset stuck bits during marking

In the same way that a tracing collector can reset reference counts during the marking phase of a collection, it can also restore stuck one-bit counts for objects that are no longer shared [Roth and Wise 1998]. Because a tracing collector was not present in the builds used in Section 4.3, this was not analysed, however it may provide further opportunity for increased performance from the one-bit count by reducing unnecessary copies.

### 4.5.3 Storing the one-bit count in references

Past work [Chikayama and Kimura 1987; Stoye et al. 1984] has described storing the one-bit count in pointers to objects rather than in the object headers themselves. Unused bits in object references in the runtime can be used to 'tag' a reference as a shared or unshared reference. This implementation detail saves dereferencing each pointer every time the bit must be checked, reducing memory accesses. This is highlighted as particularly important in multiprocessing systems where the object being pointed to is not actually in the local memory of the machine with the pointer, although this is not a concern in HHVM. This technique has not been explored in HHVM in the analysis presented due to the complexities of introducing it, but may provide further opportunity for improvement.

## 4.6 Summary

This chapter outlined the issues that surround PHP in regards to garbage collection strategies. It described how PHP has been tied to naive reference counting in the past, and introduced and evaluated the one-bit reference count as a solution to these problems. It showed that these barriers to high-performance garbage collection in PHP can in fact be overcome, and that PHP can be made compatible with tracing garbage collection. This chapter also outlined how these changes to the behaviour of PHP may affect development of PHP programs.

The results of this work thus far are a build of HHVM that does not rely on exact, naive reference counting, yet can still successfully run major PHP applications. This work provides the basis for Chapter 5, which provides the design and analysis of the first tracing garbage collection scheme for PHP that does not include traditional reference counts.

Chapter 5

# HHVM Without Reference Counts

Chapter 4 described the state of garbage collection in PHP and outlined the historical reasons this has occurred. It also introduced the use of a one-bit reference count that is used in place of full, naive reference counting, which enables further exploration of tracing garbage collection for PHP.

This chapter explores the design and implementation of a tracing garbage collector in HHVM. Section 5.1 outlines the design considerations that are specific to HHVM. Section 5.2 outlines the details of the proposed garbage collector, taking into account considerations from Section 5.1. Section 5.3 details the results of the implementation of this garbage collector and compares performance and memory characteristics to the naive reference counting implementation. Section 5.4 outlines opportunities for future work in this area and discusses potential improvements that could be made to the existing implementation.

## 5.1 Design space considerations

This Section outlines the design considerations for implementing a tracing garbage collector in HHVM. The design for a new memory management system in Section 5.2 takes into account the considerations presented here as well as the current state of garbage collection in HHVM, the results from literature and the underlying properties of typical PHP programs.

### 5.1.1 Conservative Marking

HHVM does not maintain stack maps and therefore must perform conservative marking. To determine whether an ambiguous reference points to a valid object in the heap, a conservative marking algorithm must be able to identify whether the referenced location is an allocated object. It must then find and mark the object as reachable. This requires the memory manager to maintain a record of where live objects are located in the heap or to calculate this on-the-fly by using a parsable heap structure.

HHVM uses a *parsable heap* which involves writing a special object header at the start of each hole in the heap. Each object and hole in the heap can then be iterated over, starting at the beginning of a block. This can be quite costly to maintain as reclaiming an object requires writing to the heap.

Moving collectors must also be able to pin objects that are referred to by ambiguous references in the event that the reference is actually a value, so that the value is not changed when the object is moved, and must be resilient to the potential fragmentation caused by object pinning.

### 5.1.2 Incremental Collection

A one-bit reference count as introduced in Chapter 4 allows for the incremental collection of objects that are never shared. RC Immix combines limited-bit reference counting with a line and block layout, but this has not been explored for one-bit reference counts. A build of HHVM that uses one-bit reference counts for copy-on-write can decide whether to also use these for garbage collection. However, unless an optimisation like coalescing [Levanoni and Petrank 2001] is used like in RC Immix, the build must include *decRef* operations, which can be completely removed if the one-bit RC is not being used for garbage collection. A potential pitfall of this strategy is that it once again ties garbage collection strategy to language semantics, which this thesis shows to be a dangerous choice.

### 5.1.3 Heap Partitioning

Garbage collection algorithms often make use of separate heap spaces for logically different classes of allocations. A typical example involves maintaining a large object space where allocations over a certain size are individually allocated and reclaimed, allowing different behaviour for different classes of allocations. Finding suitable cut-off values for these classes is a key consideration in designing a garbage collector.

### 5.1.4 Explicitly Managed Allocations

HHVM has a number of explicitly managed allocations that are also allocated on the garbage-collected heap. These are not reference counted by the virtual machine and cannot be reclaimed by the garbage collector. However, they do need to be scanned during the marking phase because they may contain pointers into the heap, and so must participate in reachability analysis.

These types of allocations are typically used by extensions in HHVM which need to heap allocate objects internal to the VM. Raw calls to malloc/free and smart-pointer managed internal VM objects are typical cases which result in an explicitly managed allocation. In these cases, the garbage collector cannot know when these need to be collected, and must wait until they are explicitly released by the caller.

Currently HHVM allocates these objects using the same size-segregated free-list as reference counted allocations and ignores them when it finds them unmarked during the sweep phase instead of reclaiming their memory. A new garbage collection scheme must consider how these allocations need to be involved in the tracing phase and whether they should be allocated in the garbage collected heap, given that they are not collectable objects. In a moving collector, explicit allocations must be pinned

because the collector cannot know of all references to the allocation, and is therefore unable to update them to point to the new location.

### 5.1.5 Triggering Collection

The backup tracing collector in HHVM can be called explicitly through the use of the gc_collect() API in PHP and is also run at the end of request threads, both of which only occur if enabled by a runtime flag. The heuristics for triggering garbage collection in HHVM are quite basic, because reference counting is still responsible for most garbage collection, with only cycles being left to the tracing collector. Controlling the triggering of the garbage collector is an essential part of the design.

### 5.1.6 Threading Considerations

HHVM currently uses a thread-local heap which serves one request at a time. Concurrency is not provided at the level of PHP code, which simplifies the design, because each request simply has its own heap which does not need to include synchronisation operations. A single HHVM instance acts as a server, paired with a HTTP server to handle each incoming request. At this level, multiple threads execute at the same time, handling separate and independent requests, each with their own heap. Thread-local heaps request blocks of memory from malloc, and free all allocation at the end of each request.

## 5.2 Proposed Design

This thesis introduces a prototype mark-region garbage collector based on Immix for use in HHVM. Immix cannot be used as a drop-in replacement for reference counting and this section discusses the adaption of Immix to HHVM, specifically addressing the design considerations from Section 5.1.

### 5.2.1 Heap Organisation

The first major difference between HHVM's reference counting and Immix is the heap layout. The new collector separates the heap into 128 byte lines in 32 Kbyte blocks, bump-allocating small and medium sized objects allowing them to span multiple lines but not multiple blocks. Small objects are defined as 128 bytes or less, medium as all other objects 8Kbyte or less, and large as all remaining objects. Large objects are allocated in a separate large object space which uses malloc/free for each allocation.

Medium objects that are unable to fit in the current free space are overflow-allocated to a special-purpose block to limit the wasted space at the end of each block. As each block becomes full, a new block is requested using malloc which provides a centralized API for block distribution to the thread-local allocators/collectors.

### 5.2.2 Object Map and Conservative Marking

The backup mark-sweep collector used in HHVM includes conservative marking, but makes use of expensive operations that iterate over the entire heap in order to locate valid headers. As objects are incrementally reclaimed through reference counting, their header in the heap is overwritten with a *FreeNode*, which stores the size of the free space until the next header. Maintaining a mix of allocations and FreeNodes in the heap allows the entire heap to be parseable by beginning at the start of each block and following the stored size of each allocation or FreeNode to the next header.

This strategy is not compatible with Immix, which does not reclaim individual objects, only lines or blocks. Writing the FreeNode header to the heap for every object that is reclaimed causes a performance overhead, and introduces a complex discipline that must be followed in order to ensure that the heap is parseable at all times.

To limit the space and performance overhead of conservative marking, the prototype collector uses one bit in an object map [Shahriyar et al. 2014] per 16-byte alignment in the Immix heap, limiting the space overhead to $1/128$ of heap size. When an object is allocated, the bit that corresponds to the start of the object in the object map is set.

Shahriyar, Blackburn, and McKinley [2014] propose completing all conservative marking at the start of the marking phase, then zeroing the object map and re-creating it when performing exact marking. This requires that all scanning of heap objects is exact, which is not the case in HHVM. The prototype collector instead performs all marking, and then re-creates the map based on the mark bits of each heap object.

To diagnose an ambiguous reference during conservative marking, the pointer is first range checked to see whether it is within an Immix block or a large object. If it is within a large object, the object is marked and added to the scan queue. If it is within an Immix block, the object map is consulted and the appropriate header is marked and added to the scan queue. If the pointer does not point into the Immix heap or a big allocation, it cannot be a reference to a live allocation and is discarded.

Conservative marking may also encounter interior pointers, which reference a location within an allocated object, rather than the header of the object. These objects still need to be marked as live so that they are not collected while a pointer into them remains. The prototype conservative marker back-tracks to the nearest header in the object map in order to correctly diagnose interior pointers.

Scanning progresses through the queue to perform a transitive closure over the heap, marking each object as it is found. The sweep phase then iterates over each object in the heap using the object map and finds marked objects, setting the corresponding bit in the object map and marking its Immix line.

Blocks that contain unmarked lines are then made available for allocation, and the allocator will re-use empty lines within partially full blocks before requesting new blocks.

### 5.2.3 Explicit Allocations

Explicit allocations are outside the control of the memory manager and pose a problem when combined with the Immix line and block layout. This collector allocates explicitly managed allocations into a separate space to avoid the excess pinning caused by explicit allocations, which cannot be moved.

It uses a *lazy bump-pointer* allocator (see Section 5.3.1), which we designed specifically to reduce allocation cost and memory usage in the event that an object is reclaimed before the next allocation, which is extremely common for explicit allocations in HHVM.

## 5.3 Experimental Results

### 5.3.1 Explicit Allocations

Using the virtual machine's memory manager to allocate internal VM objects is uncommon, so literature discussing Immix does not explore whether to treat these allocations specially. This section profiles explicit allocations in HHVM to ensure that the allocation strategy used is appropriate. Several choices exist for managing explicit allocations:

(a) Allocate explicitly managed allocations in the Immix heap, with no special treatment,

(b) Pass explicit allocations through to the standard library's malloc/free implementation, the same strategy that is used for big allocations, or

(c) Use a separate allocation space.

#### Properties of explicit allocations

The experiments presented in this section show that explicitly managed objects in HHVM have extremely unusual lifetime characteristics. Remarkably, the majority of allocations are reclaimed before the next explicit allocation is performed, from the perspective of the allocator. This chapter refers to these allocations as *ephemeral* allocations.

**Table 5.1:** Demographics of explicitly managed allocations

| Application | Ephemeral Allocations | Immortal Allocations |
|---|---|---|
| Wordpress | 93.9% | 3.7% |
| Mediawiki | 90.2% | 1.1% |
| Drupal7 | 70.3% | 5.7% |

Table 5.1 shows analysis of the demographics of explicitly managed allocations in HHVM for several PHP applications. The results are extraordinary, with ephemeral allocations making up more than 90% of allocations for Wordpress and Mediawiki.

If a typical bump-pointer allocator is used where ephemeral allocations are the common case, the typical result is a large chain of allocated objects where only the most recent allocation is actually live. This wastes heap space because the majority of objects have been reclaimed, but their space cannot be re-used.

Explicit allocations are also unusual in that of the allocations that are not immediately reclaimed, many live until the end of the request, meaning they will never be reclaimed during execution for re-use. These are called *immortal* allocations.

#### Lazy bump pointer

This combination of observations motivates the use of a *lazy bump pointer* allocation space for explicitly managed allocations, which I present here as a novel optimisation for explicit allocations.

When an object is allocated, the size allocated is stored, but the bump pointer is not changed. If the next event from the perspective of the memory manager is an allocation request, the stored size is added to the bump pointer and the new allocation request is handled. If the next request was instead a call to free the most recently allocated object, the stored size is cleared, resulting in the next allocation overwriting the now-reclaimed previous allocation. This enables immediate re-use of memory that would otherwise be unused until the next collection, which increases memory efficiency as the rate of ephemeral allocations increases.

Table 5.1 shows that remarkably, ephemeral allocations are the common case for explicitly managed allocations in each of the PHP frameworks tested. Table 5.2 shows that the amount of memory saved by the lazy bump pointer ranges from 37.7% up to 93.7% in the best case, for explicitly managed allocations. This translates to between 5.6% and 17.2% of total heap space saved by the use of the lazy bump pointer allocator for explicitly managed allocations.

**Table 5.2:** Relative heap size of ephemeral allocations

| Application | Ephemerals as % of Explicit Heap | Ephemerals as % of Total Heap |
|---|---|---|
| Wordpress | 93.7% | 17.2% |
| Mediawiki | 86.7% | 5.6% |
| Drupal7 | 37.7% | 5.6% |

Lazy bump pointer is conceptually similar to *unbumping* in order to deallocate an ephemeral allocation. Guyer, McKinley, and Frampton [2006] proposed that explicit calls to free could be statically inserted at compile time and used to reclaim unreachable objects at runtime. One variant of this scheme involved unbumping the bump pointer if the call to free was for the most recently allocated object. This scheme was not able to insert explicit free operations for all objects, and required additional compiler cooperation. In contrast, explicitly managed allocations must have a call to free, unless they leak memory. Lazy bump pointer also inverts the assumption by expecting that explicit allocations are ephemeral, which is the common case in HHVM, and

only paying the cost of maintaining them in the uncommon case that they live.

### Comparison of different strategies

Table 5.3 presents a comparison of the performance of three different allocation strategies for explicitly managed allocations.

**Table 5.3:** Performance of different explicit allocation strategies

| Application | in-immix-heap | malloc/free | lazy-bump |
|---|---|---|---|
| Wordpress | 1.0 | 0.813 | 1.125 |
| Mediawiki | 1.0 | 1.050 | 0.984 |
| Drupal7 | 1.0 | 0.863 | 0.881 |

Treating explicit allocations as regular heap objects causes no overhead at allocation time, but scatters these allocations throughout the heap, even though they cannot be reclaimed by the memory manager and must be pinned. The benchmarks used do not perform collection until the end of the request, due to the small heap size needed and therefore do not pay the cost of this fragmentation, but longer running requests with larger heap sizes would. Additionally, the conservative marker does not need to consider them as valid allocations, because it does not matter whether they are marked or not, meaning they have quite different properties to other objects in the Immix heap. This motivates a separate allocation space with different behaviour.

Lazy bump pointer does not outperform the other two strategies in every case, but does provide better memory usage. It can also be further optimised and would be highly affected by the choice of the medium/large size cut-off. With further tuning, lazy bump pointer can gain even more ground over other strategies.

Use of malloc/free allocates the same amount of memory as regular allocation in the Immix heap, but causes more performance slowdown. The lazy bump-pointer is able to immediately re-use memory with a low performance overhead, and does not fragment the heap by pinning additional lines and blocks.

### Summary

Explicit allocations are not released by the memory manager, meaning they cannot be moved, and the associated lines must be pinned. In HHVM, explicit allocations need to be scanned because they may contain pointers into the heap, which must be followed as part of the reachability analysis. Explicit allocations do not need to be marked, however, because they cannot be reclaimed by the memory manager even if they are unreachable, until the owner of the allocation releases them.

Our analysis shows that explicit allocations in several PHP open source applications have remarkable properties, in that a large proportion are ephemeral allocations. By allowing the most recent allocation to be reclaimed using a lazy bump pointer, the allocator can save significant amounts of memory that would otherwise be unusable

until at least the next collection. This would most likely also improve locality for allocations that survive.

### 5.3.2 Collector Performance Results

The prototype presented in this chapter is able to run each of the three PHP applications used for benchmarking, but it does not yet outperform naive reference counting garbage collection in HHVM.

**Table 5.4:** Performance comparison of the prototype collector

| Application | RC + Backup tracing | Immix (lazy-bump) |
|---|---|---|
| Wordpress | 1.0 | 0.754 |
| Mediawiki | 1.0 | 0.744 |
| Drupal7 | 1.0 | 1.050 |

Table 5.4 shows the final results of the new prototype collector compared with the naive reference counting implementation with backup tracing enabled. The prototype collector performs slightly better for Drupal7, but falls short of the backup tracing collector by a margin of 25%.

## 5.4 Further Work

The memory manager profiled in Section 5.3 is a prototype implementation of a high performance garbage collector (Immix) adapted for HHVM, but is far from complete. Many opportunities exist to continue to improve its performance.

### 5.4.1 Implement Defragmentation

The prototype collector does not opportunistically evacuate allocations at collection time in order to defragment the heap, as Immix does. Due to the nature of PHP as a language for web back-end scripting, many of the requests used in the macro benchmark suite do not invoke the collector until clean-up at the end of each request. This makes defragmentation less crucial, as it would never be used in such sort-lived requests. However, longer running server scripts or large requests would make use of the collector throughout the request, and heap defragmentation would be more beneficial to these types of requests.

### 5.4.2 Remove reference counting assumptions

HHVM has been built with reference counting as the garbage collection algorithm, and the way reference counts are maintained throughout the codebase reflects this. The one-bit reference count implementation was developed to directly replace reference counting in HHVM and did not refactor code in HHVM that assumes the presence of full reference counts.

The reference counting optimisations in the JIT implement a number of strategies to reduce the number of reference counting operations emitted in native code, but are based on the assumption that full reference counts exist. There may be further opportunity to improve this pass when dealing with the much simpler one-bit count.

### 5.4.3 Improve timing of collection

The prototype collector runs a full collection at the end of each request, including marking live objects and identifying recyclable lines and blocks in the heap. The heap then immediately frees all blocks during the shutdown sequence of the request. If the request reaches its end without triggering the collector, there is no need to run the collector before returning all outstanding memory because there is no discernible difference.

Heuristics for invoking the collector are also quite basic, and this is another area where careful evaluation and experimentation could improve this prototype.

### 5.4.4 Tune collector parameters

Immix has several parameters which are easily adjusted that may result in increased performance. Experimentation with line and block size may be able to adjust the prototype collector to better suit the object demographics of HHVM. Line size also determines the maximum size of small objects in Immix, but the cut-off point between medium and large objects can be adjusted independently, depending on fragmentation concerns and the cost of calls to malloc/free for large objects. Profiling of the various factors at play and experimentation with different parameters may yield further performance gains.

### 5.4.5 Remove costly heap-size dependent operations

In the prototype collector, during the final phase of collection when lines are being marked as recyclable, any object that has been reclaimed must have a small fragment of clean-up code run in order to check for and remove the dynamic properties array associated with the PHP object. This is a costly operation, as it requires scanning over the entire heap using the object map, so its runtime cost scales with the size of the heap. This piece of clean-up and others like it could be ignored, delayed, batched or otherwise made cheaper through a variety of implementation options, speeding up the final phase of collection.

Each heap object has a mark bit which maintains the state of whether an object is currently reachable by the collector or not. The prototype implementation iterates over the entire heap to reset these mark bits before the marking phase, which accesses every single heap object. By alternating whether the bit represents marked or unmarked at each collection, this costly iteration can be avoided.

## 5.5 Summary

This chapter introduced a tracing garbage collector for PHP, implemented in HHVM. This is the first PHP implementation that can claim widespread compatibility with existing PHP programs without the use of naive reference counting, proving that PHP can be made compatible with different garbage collection strategies and opening the door for future research in this area.

The garbage collector in this chapter can be used as a basis for further improvements or novel optimisations by providing a test-bench on which experiments can take place. It also serves as a useful tool for exploring the design space of a tracing garbage collector in HHVM and PHP.

Chapter 6

# Conclusion

This thesis has outlined the topic of garbage collection and explored its use and misuse in the dynamic web language PHP. PHP implementations have been tied to using naive reference counting for their garbage collection since the inception of the language. Demand for a high performance PHP virtual machine has risen, primarily led by Facebook in order to run their extremely high traffic web servers. In addressing the performance of PHP, garbage collection presents unique challenges and opportunities for improvement.

This thesis has presented three language features of PHP which rely on naive reference counting for efficient implementation. In order to further pursue garbage collection as an area of research, this thesis has presented the one-bit reference count as a replacement for full, naive reference counting. This optimisation has considerable benefits, requiring only one bit of overhead in the object headers whilst providing comparable results when used to inform copy-on-write checks. Changes to the behaviour of destructors introduced are found to be consistent with the PHP specification, and code that relies on old behaviour is found to be easily adapted using alternative design patterns.

With these barriers to tracing garbage collection removed, Chapter 5 proposed a design for a new garbage collection scheme for HHVM and explored the design space for such a collector. The high performance garbage collector Immix was chosen as a base and adapted to fit the peculiarities of HHVM, providing the first PHP virtual machine to behave correctly without the use of naive reference counting. Explicitly managed allocations in HHVM were analysed and found to have quite different properties to other allocations, and a new lazy bump pointer allocator was presented and evaluated.

Many potential improvements to the prototype collector are still possible, and this prototype can act as a basis for further work towards a high performance tracing garbage collector for PHP.

## 6.1 Further Work

### Improve the prototype collector

This thesis presents a prototype tracing garbage collector based on Immix, but it does not reach the performance of HHVM naive reference counting. This implementation has focused on several areas including improving explicitly managed allocations and achieving logical correctness, particularly in conservative scanning. There are still many opportunities to focus on different areas of the collector and HHVM to improve memory management, including removing some operations that do not integrate well with Immix, improving how collection is triggered and removing costly whole-heap operations during collection.

### Investigate reference counting Immix

Reference counting Immix embraces reference counting and presents some novel optimisations that allow the combination of high performance reference counting with tracing garbage collection. Given PHP's reliance on reference counting, RC Immix may be able to provide high performance garbage collection for HHVM. Such an implementation could use highly optimised reference counts or examine integrating the one-bit reference count with RC Immix.

### Explore reference demotion semantics

This thesis has not focused on resolving the issue of PHP's reference demotion semantics because these do not present as much of a performance barrier as copy-on-write does. It may be the case that reference demotion behaviour is not relied on very often in typical PHP code. If it does prove necessary, there may be a range of implementation options that can provide these semantics without the cost of naive reference counting.

# Bibliography


BACON, D. F. AND RAJAN, V. 2001. Concurrent cycle collection in reference counted systems. In J. L. KNUDSEN Ed., *15th European Conference on Object-Oriented Programming*, Volume 2072 of *Lecture Notes in Computer Science* (Budapest, Hungary, June 2001), pp. 207–235. Springer-Verlag. (p. 10)

BISSONNETTE, P. 2015. Lockdown results and hhvm performance. `http://hhvm.com/blog/9293/lockdown-results-and-hhvm-performance`. Accessed: 2015-08-28. (p. 21)

BLACKBURN, S. AND MCKINLEY, K. S. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In R. GUPTA AND S. P. AMARASINGHE Eds., *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 43(6) (Tucson, AZ, June 2008), pp. 22–32. ACM Press. (pp. 2, 11)

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ACM SIGMETRICS Performance Evaluation Review 32(1) (June 2004), pp. 25–36. ACM Press. (p. 6)

BOEHM, H.-J., ATKINSON, R., AND PLASS, M. 1995. Ropes: An alternative to strings. *Softw. Pract. Exper. 25*, 12 (Dec.), 1315–1330. (p. 25)

CHENEY, C. J. 1970. A non-recursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov.), 677–8. (p. 11)

CHIKAYAMA, T. AND KIMURA, Y. 1987. Multiple reference management in Flat GHC. In *4th International Conference on Logic Programming* (1987), pp. 276–293. (pp. 7, 33, 34)

COLLINS, G. E. 1960. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (Dec.), 655–657. (pp. 1, 7)

DEUTSCH, L. P. AND BOBROW, D. G. 1976. An efficient incremental automatic garbage collector. *Communications of the ACM 19*, 9 (Sept.), 522–526. (p. 9)

FACEBOOK INC. 2015. HHVM. `https://github.com/facebook/hhvm/commit/9b4363859e7f4156851294f3f79ef2a0f4b58e25`. (p. 18)

GOSLING, J., JOY, B., STEELE, G., BRACHA, G., AND BUCKLEY, A. 2014. *The Java Language Specification* (Java SE 8 ed.). Addison-Wesley. (p. 15)

GUYER, S. Z., MCKINLEY, K. S., AND FRAMPTON, D. 2006. Free-Me: A static analysis for automatic individual object reclamation. In M. I. SCHWARTZBACH AND

T. BALL Eds., *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 41(6) (Ottawa, Canada, June 2006), pp. 364–375. ACM Press. (p. 40)

INAMURA, Y., NOBUYUKI, I., KAZUAKI, R., AND KATSUTO, N. 1989. Optimization techniques using the MRB and their evaluation on the Multi-PSI/V2. In *North American Conference on Logic Programming, 1989*, Volume 2 (1989), pp. 907–921. MIT Press.

ISO/IEC. 2014. Programming language c++. ISO/IEC 14882:2014, International Organization for Standardization, Geneva, Switzerland. (p. 15)

JIBAJA, I., BLACKBURN, S. M., HAGHIGHAT, M. R., AND MCKINLEY, K. S. 2011. Deferred gratification: Engineering for high performance garbage collection from the get go. In J. VETTER, M. MUSUVATHI, AND X. SHEN Eds., *Workshop on Memory System Performance and Correctness* (San Jose, CA, June 2011). (p. 2)

LEVANONI, Y. AND PETRANK, E. 2001. An on-the-fly reference counting garbage collector for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 36(11) (Tampa, FL, Nov. 2001), pp. 367–380. ACM Press. (p. 36)

MCCARTHY, J. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM 3*, 4 (April), 184–195. (pp. 1, 8, 10)

NISHIDA, K., KIMURA, Y., MATSUMOTO, A., AND GOTO, A. 1990. Evaluation of MRB garbage collection on parallel logic programming architectures. In *7th International Conference on Logic Programming, Jerusalem* (June 1990), pp. 83–95. MIT Press.

ROTH, D. J. AND WISE, D. S. 1998. One-bit counts between unique and sticky. In S. L. PEYTON JONES AND R. JONES Eds., *1st International Symposium on Memory Management*, ACM SIGPLAN Notices 34(3) (Vancouver, Canada, Oct. 1998), pp. 49–56. ACM Press. (p. 33)

SERGEANT, T. 2014. Improving memory management within the hiphop virtual machine. Australian National University Honours Thesis. (pp. 14, 18, 26)

SHAHRIYAR, R., BLACKBURN, S. M., AND FRAMPTON, D. 2012. Down for the count? getting reference counting back in the ring. In K. MCKINLEY AND M. VECHEV Eds., *11th International Symposium on Memory Management* (Beijing, China, June 2012), pp. 73–84. ACM Press. (pp. 1, 7)

SHAHRIYAR, R., BLACKBURN, S. M., AND MCKINLEY, K. S. 2014. Fast conservative garbage collection. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, OR, Oct. 2014), pp. 121–139. ACM Press. (pp. 9, 38)

SHAHRIYAR, R., BLACKBURN, S. M., YANG, X., AND MCKINLEY, K. S. 2013. Taking off the gloves with reference counting Immix. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Indianapolis, IN, Oct. 2013), pp. 93–110. ACM Press. (pp. 10, 13, 18, 27)

STOYE, W. R., CLARKE, T. J. W., AND NORMAN, A. C. 1984. Some practical methods for rapid combinator reduction. In G. L. STEELE Ed., *ACM Conference on LISP and Functional Programming* (Austin, TX, Aug. 1984), pp. 159–166. ACM Press.

STYGAR, P. 1967. LISP 2 garbage collector specifications. Technical Report TN-3417/500/00 (April), System Development Corporation. (p. 11)

THE PHP GROUP. 2015. The PHP language specification. `http://git.php.net/?p=php-langspec.git`. Accessed: 2015-08-28. (p. 15)

TOZAWA, A., TATSUBORI, M., ONODERA, T., AND MINAMIDE, Y. 2009. Copy-on-write in the php language. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09 (New York, NY, USA, 2009), pp. 200–212. ACM. (pp. 2, 19, 24)

UGAWA, T., IWASAKI, H., AND YUASA, T. 2010. Improved replication-based incremental garbage collection for embedded systems. In J. VITEK AND D. LEA Eds., *9th International Symposium on Memory Management* (Toronto, Canada, June 2010), pp. 73–82. ACM Press. (p. 7)

UNGAR, D. M. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices 19(5) (Pittsburgh, PA, April 1984), pp. 157–167. ACM Press. (pp. 8, 10)

WISE, D. AND FRIEDMAN, D. 1977. The one-bit reference count. *BIT Numerical Mathematics 17*, 3, 351–359. (p. 7)

YAO, M. 2013. Vector rope example. `https://commons.wikimedia.org/wiki/File:Vector_Rope_example.svg`. Accessed: 2015-09-07. (p. 26)