Improving Memory Management Within the HipHop Virtual Machine

Timothy Sergeant

A thesis submitted in partial fulfilment of the degree of
Bachelor of Advanced Computing (Honours)
The Australian National University

October 2014

© Timothy Sergeant 2014

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

Timothy Sergeant
24 October 2014

To my parents,
Evan and Elizabeth

Acknowledgments

Firstly, an enormous thank you to my supervisor, Steve Blackburn. Steve's wealth of knowledge in the area has been endlessly valuable for this project. He was always able to provide useful advice for the problems I was facing, and talking with Steve always left me with renewed vigor for the project.

Thank you to the members of the HHVM team at Facebook who I have talked to throughout the year, especially Bert Maher, Guilherme Ottoni and Joel Pobar, who have provided useful insight and speedy answers to my questions. A particular thanks to Joel for giving me the opportunity to experience Silicon Valley.

To everyone else in the #hhvm IRC channel: Sorry for spamming you with endless failed build notifications.

Thank you to Wayne Tsai and Peter Marshall for providing valuable feedback on draft copies of this thesis, as well as to everyone else in the honours cohort who has helped me throughout the year.

Thank you to my housemates, Tom and Wayne, who somehow put up with the decreasing quality of my jokes as my stress level increased. Thank you also to the rest of my friends for being there for me throughout the year.

Finally, thank you to my family, Evan, Elizabeth and Steph. None of this would have been possible without your support.

Abstract

PHP is a popular dynamic scripting language for webpages, known for ease of development and wide availability. However, little work has been done to optimise PHP's memory management. Two major PHP implementations, PHP5 and HHVM, both use naive reference counting for memory management, an algorithm which is known to be slow and expensive. However, the semantics of PHP loosely tie the language to naive reference counting.

This thesis argues that high performance memory management is within reach for PHP.

We perform analysis of the memory characteristics of the HipHop Virtual Machine to determine how it compares to similar virtual machines. We describe and experimentally evaluate the changes required to remove reference counting from HHVM. Finally, we provide a design for a proof-of-concept mark-region tracing collector for HHVM, with a discussion of the issues faced when implementing such a collector.

We find that HHVM has similar memory demographics to PHP5 and Java, and would be well suited to high performance garbage collection algorithms such as Immix. However, we encounter a performance tradeoff associated with removing reference counting, due to the need to weaken the copy-on-write optimisation for arrays. Our proposed alternative, blind copy-on-write, was found to be ineffective for production use, we propose avenues for future work to reduce this tradeoff.

We are confident that these challenges can be overcome to create a high performance garbage collector for HHVM. While PHP is widely used on small webpages where performance is not critical, it is also used at scale by companies such as Facebook and Wikimedia. In this context, even a small performance gain can have have a significant cost impact. This result has impact beyond just HHVM, as the techniques described in this thesis could be adapted to PHP5 and other PHP runtimes, making a large part of the web faster by improving memory management.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction **1**
1.1 Thesis Statement 1
1.2 Contributions 2
1.3 Meaning 2
1.4 Thesis Outline 3

2 Background and Related Work **5**
2.1 Overview of garbage collection 5
2.1.1 Naive reference counting 5
2.1.2 Reference counting optimisations 6
2.1.3 Tracing garbage collection 7
2.1.4 Generational garbage collection 8
2.1.5 Conservative garbage collection 9
2.2 Reference counting in PHP 9
2.2.1 Pass-by-value for arrays 10
2.2.2 PHP references 11
2.2.3 Precise destruction 14
2.3 The HipHop Virtual Machine 14
2.4 Related work 15
2.4.1 PHP compilers 15
2.4.2 Targeting existing virtual machines 16
2.5 Summary 17

3 Experimental Methodology **19**
3.1 Software platform 19
3.2 Benchmarks 19
3.2.1 Compiling and running 20
3.3 Hardware platform 20

4 Memory Management in HHVM **21**
4.1 Instrumented HHVM build . . . 21
4.2 Lifetime of memory-managed objects . . . 22
4.3 Size of objects . . . 22
4.4 Maximum value of reference counts . . . 26
4.5 Heap usage . . . 27
4.6 Reference count eliding . . . 31
4.7 Summary . . . 31

5 Eliminating Reference Counting from HHVM **33**
5.1 Copy-on-Write . . . 33
5.1.1 Blind Copy-on-Write . . . 33
5.1.2 Experimental evaluation . . . 34
5.1.3 Discussion and alternative optimisations . . . 34
5.2 Removing reference counting operations . . . 36
5.2.1 Method . . . 36
5.2.2 Experimental evaluation . . . 37
5.3 HHVM without reference counting . . . 37
5.4 Summary . . . 40

6 Design of a Tracing Garbage Collector for HHVM **41**
6.1 Block-based allocation . . . 41
6.1.1 Memory allocation in HHVM . . . 41
6.1.2 blockMalloc . . . 42
6.1.3 PHP extensions . . . 42
6.1.4 Experimental evaluation . . . 43
6.2 Collection and recycling . . . 43
6.2.1 Collection algorithm . . . 43
6.2.2 Triggering garbage collection . . . 45
6.3 Summary . . . 46

7 Conclusion **47**
7.1 Future work . . . 48

Appendix A HHVM Patches **49**
A.1 Obtaining patches . . . 49
A.2 Applying patches . . . 49
A.3 Index of available patches . . . 50

References **53**

List of Figures

2.1 Demonstration of pass-by-value semantics for PHP arrays 10
2.2 Demonstration of reference demoting in PHP 12
2.3 Final states of the programs in Figure 2.2 13

4.1 Lifetime of objects within HHVM 23
4.2 Object size demographics within HHVM 24
4.3 Cumulative frequency of object sizes in open source applications 26
4.4 Maximum size of refcounts within HHVM 27
4.5 Bits required to store reference counts. A 3-bit reference count field would suffice for 99.7% of objects 28
4.6 Heap usage of HHVM over the course of a WordPress and MediaWiki request 29
4.7 Heap usage of HHVM over the course of a WordPress and MediaWiki request, with each type split into header and data sections to show the amount of heap used by headers 30
4.8 Performance of HHVM-3.1 without refcount eliding compared to default HHVM-3.1 32

5.1 Performance evaluation of blind-cow compared to HHVM-3.1 35
5.2 Performance evaluation of no-refcount compared to no-collect 38

6.1 Performance evaluation of block-malloc compared to HHVM-3.1 44

A.1 Recommended application order for patches. Starting at the root of the tree, apply each of the patches in order until the desired version of HHVM is reached 50

List of Tables

4.1 Mean and median object size for open source applications 25

5.1 Summary of the behaviour of different array optimisations 34
5.2 Confidence in the no reference counting microbenchmark results of Figure 5.2(a) 39
5.3 T-test confidence in the no reference counting open source application results of Figure 5.2(b) 39

Chapter 1

Introduction

This thesis examines and evaluates an area of the PHP programming language that has seen little attention previously: memory management. PHP continues to use outdated garbage collection techniques, despite recent innovations in the field providing far greater performance. It is my thesis that high performance garbage collection is achievable for the PHP language and for the HipHop Virtual Machine.

1.1 Thesis Statement

PHP is something of an anomaly in the programming languages community. After beginning life as a simple tool for laying out dynamic webpages, PHP has grown organically to become a powerful language with an expansive standard library. Additionally, PHP is very popular for web programming – according to one survey, PHP is used for 82% of web pages [Q-Success, 2014], and is used by numerous high-traffic websites including Facebook and Wikipedia. However, despite this popularity, many recent advances in programming language technology have not been implemented in PHP.

One such advancement is garbage collection. The reference implementation of PHP, which we refer to as PHP5, uses naive reference counting to implement garbage collection. This algorithm was devised in the 1960s for use in early Lisp systems, and is now widely considered unacceptable for high performance systems, having been superseded by generational tracing garbage collectors and high performance reference counting collectors such as RC Immix.

Furthermore, it is perceived that PHP is tied to reference counting to allow for high performance. While the newly-written draft specification for PHP does not specifically require the use of reference counting, it recommends that implementations use a copy-on-write optimisation for arrays, which requires naive reference counting to be implemented effectively. PHP's reference variables also have semantics which are most easily implemented using naive reference counting. Additionally, PHP5 has long been considered a de-facto standard for PHP. Therefore, any new PHP implementation which wants to reach full compatibility with existing PHP applica-

tions must behave in the same way as PHP5, regardless of what is allowed by the draft specification.

Recently, Facebook has released an open-source, high performance PHP virtual machine, the HipHop Virtual Machine (HHVM). HHVM has almost full compatibility with PHP5 and includes a number of performance improvements, including a JIT compiler, which allow it to reach performance that is several times better than PHP5. HHVM also includes support for Hack, a new PHP-like language which includes a static type checker. However, HHVM and the Hack language are still tied to naive reference counting in the same way as PHP5.

This thesis addresses the problem of implementing a tracing garbage collector for PHP, targeting the HipHop Virtual Machine. By doing so, I aim to address the perception that naive reference counting is required for performant PHP, and open the door for further optimisation using advanced garbage collection and memory management algorithms.

1.2 Contributions

This thesis has three main contributions:

Evaluation of memory management in HHVM We perform a detailed analysis of the current state of memory management within HHVM. We analyse the demographics of objects and compare them to PHP5 and Java. We find that HHVM has memory characteristics which indicate that it would be well suited to high performance garbage collectors such as Immix.

Eliminating barriers to high performance garbage collection We describe and evaluate changes to HHVM to allow it to successfully run PHP programs without reference counting. We find that our proposed replacement for array copy-on-write, "Blind Copy-On-Write" has unacceptable performance for production use, and propose alternative solutions.

Design of a tracing collector We provide a detailed description of the design of a Mark-Region collector suitable for implementation within HHVM. We expect that this collector could be competitive with HHVM's existing naive reference counting collector. The collector is designed to provide the infrastructure for future implementation of an Immix collector for HHVM.

1.3 Meaning

This thesis intends to open the gates for further work to improve PHP's memory management. We create a 'clean slate' upon which future garbage collection work can be performed, and provide explanation and analysis of problems which may be

faced by implementors of these garbage collectors. In effect, we aim to provide an opportunity for PHP to 'catch up' to modern garbage collection research.

The creation of a high performance tracing collector for HHVM would have a significant impact on the PHP community. HHVM is used at large scale by Facebook, while other PHP-based companies are beginning to migrate to HHVM. For example, at the time of writing the Wikimedia Foundation is beginning a rollout of HHVM to the servers which power its applications, including Wikipedia [Wikimedia Foundation, 2014]. Optimising memory usage is a high priority for the Facebook team working on HHVM. At the scale at which Facebook operates, small reductions in response time, memory usage or memory bandwidth utilisation can translate into significant cost savings.

In the longer term, if PHP garbage collection techniques can be improved to be sufficiently performant, stable and compatible, integrating high performance garbage collection into PHP5 would provide great improvements for the wider PHP community. Many PHP websites are run in a zero-configuration environment managed by a web hosting provider. For example, WordPress.com provides hosting for millions of WordPress sites, with 16.3 billion monthly pageviews [WordPress.com, 2014]. An improvement to PHP's memory management would be able to improve the response time and throughput of all of these sites without any action from individual users.

1.4 Thesis Outline

This thesis is split into 6 chapters as follows. In Chapter 2, we provide an overview of garbage collection techniques, background information on PHP's semantics, a brief introduction to HHVM and an overview of other PHP implementations. In Chapter 3, we describe the experimental methodology which is used in the remainder of the thesis.

In Chapter 4, we perform a experimental evaluation of HHVM's memory characteristics to determine its suitability for different garbage collection algorithms. Chapter 5 describes and experimentally evaluates the steps that must be taken to remove reference counting from HHVM. In Chapter 6 we describe the implementation of a proof-of-concept tracing garbage collector for PHP.

Finally, in Chapter 7, we conclude the thesis and identify possible future work in the area.

Chapter 2

Background and Related Work

We now discuss the technical background behind garbage collection and the HipHop Virtual Machine. In Section 2.1 we start by reviewing the major performance innovations made to garbage collectors. Then, in Section 2.2 we explain how the semantics of PHP affect what reference counting optimisations are possible. In Section 2.3 we introduce the HipHop Virtual Machine and provide an overview of its operation. Finally, Section 2.4 describes other attempts to create high performance PHP compilers and virtual machines.

2.1 Overview of garbage collection

In this section, we provide an overview of the major innovations in garbage collection algorithms since their introduction in 1960.

2.1.1 Naive reference counting

One of the major archetypes of garbage collection algorithm is reference counting, which was first introduced by Collins [1960]. Collins' work concerns early LISP runtimes, where it was desirable to 'overlap' lists in order to save memory. Collins suggests incrementing a reference count field every time a list is borrowed to be used elsewhere, and decrementing the count when that reference is no longer used. By carefully maintaining correct reference counts every time lists are mutated, it is possible to know exactly when a list is no longer in use and can be collected.

Exact reference counting is able to recover memory as soon as it is no longer required, and does not require pausing mutator activity for garbage collection to take place. However, it does this at the cost of requiring small amounts of activity every time a pointer is modified. Reference counting is also simple to implement, and is used by several popular language runtimes, including PHP5, HHVM and CPython [Shahriyar et al., 2012].

Cycle collection

Reference counting is not complete, because it is unable to recover cyclic data by itself. A cycle occurs when a group of objects reference each other in such a way that all the objects have a non-zero reference count, even when they are not accessible from outside the cycle. Language implementers have a choice between three options for handling cycles in a reference counting system:

1. Do not attempt to collect cyclic garbage
2. Prohibit cyclic data from being created
3. Implement an additional cycle collector

[Shahriyar et al., 2014]

Currently, HHVM uses option 1 (as described in Section 2.3, all memory is reclaimed at the end of each request, thus the impact of cyclic garbage is low). There are two primary methods of collecting cyclic garbage. Firstly, a backup tracing collector can be implemented to periodically trace the entire heap and delete objects which cannot be reached. Alternatively, Bacon and Rajan [2001] describe a 'trial deletion' method for deleting cycles. This method maintains a list of possible cycle roots, periodically checking each subgraph to see whether it actually is a cycle, and removing those which it finds.

2.1.2 Reference counting optimisations

Many optimisations are available for naive reference counting. The most common of these, deferred and coalescing reference counting, involve making reference counts inaccurate in certain circumstances. RC Immix is an extension of this which is able to opportunistically move objects to improve locality. We also consider Nondeferred reference counting, which improves reference count performance without making reference counts inaccurate.

Deferred reference counting

Deferred reference counting makes use of the idea that reference counts only need to be accurate when objects are about to be released. Thus, deferred reference counting ignores changes to reference counts in very active areas of memory (such as registers and stacks). Instead, as reference counts are changed within the rest of the heap, a Zero Count Table (ZCT) is maintained, containing all objects which are *believed* to have a reference count of zero. Periodically, mutator activity is paused for the ignored areas of memory to be scanned. At this point, objects in the ZCT which are not found in the stacks or registers are unused and can be released [Deutsch and Bobrow, 1976].

Deferred reference counting ignores large numbers of reference count updates, but requires occasional pauses of the mutator, and is no longer able to collect memory as soon as it is unused.

Coalescing reference counting

Coalescing reference counting further improves deferred reference counting by observing that between collections, only the first and last pointer mutations are required to maintain correct reference counts. All the intermediate changes create a chain of mutations which cancel each other out. Levanoni and Petrank [2001, 2006] implement coalescing for Java as follows:

1. When an object is modified for the first time, it is marked as dirty and all outgoing references are recorded.

2. All further modifications to objects marked as dirty are ignored by the write barrier.

3. During the next collection, decrement the recorded outgoing references and increment the current outgoing references of each dirty object.

By eliminating additional reference counting operations, coalescing reference counting further reduces the overhead of the mutator write barrier.

RC-Immix

Shahriyar et al. [2013] identify that the major reason that high performance reference counting garbage collectors still perform significantly worse than their tracing counterparts is ineffective heap layout causing poor cache locality. They propose RC Immix, a reference counting collector which uses Immix's block and line heap layout (see section 2.1.3) and opportunistic copying to improve the cache locality of reference counted systems. RC Immix is able to outperform production tracing garbage collectors, making reference counting a viable choice for production systems.

Nondeferred

It is possible, although less common, to optimise reference counting garbage collection without deferring reference count updates. This can be achieved by statically analysing the program and determining what updates can be safely ignored without changing the correctness of the program. A simple example of this is the coalescing optimisation described in Joisha [2006], which erases pairs of reference count increment and decrement operations on the same object where this provably will not impact correctness. A version of this optimisation is performed by HHVM's JIT compiler. In Joisha [2007, 2008] this idea is expanded into a framework which uses a variety of optimisations to elide a greater number of reference count operations.

2.1.3 Tracing garbage collection

The second major archetype of garbage collection is tracing, first suggested by McCarthy [1960] (only months before the publication of Collins [1960]). Tracing works occasionally pausing program execution to recursively scan the heap for live objects.

In McCarthy's implementation, list nodes are scanned starting from the roots (stacks and registers), marking nodes as seen by making their sign bit negative. Then, the entire heap is stepped through from start to finish. Each node with a negative sign is flipped back to positive, while nodes with a positive sign bit are returned to the allocator.

Many refinements to this basic algorithm have been made to make it suitable for use in modern memory management systems. We briefly describe three of these algorithms: SemiSpace, Mark-Region and Immix.

SemiSpace

SemiSpace collection works by dividing the heap into equal halves. Objects are allocated using bump-pointer allocation into the first half of the heap. That is, objects are allocated contiguously by incrementing a pointer by the size of the object to allocate. Once this half of the heap is full, a collection is triggered. Starting at the roots, objects are copied across into the empty heap space, before the two spaces are swapped and allocation continues in the partially full semispace. This algorithm provides compaction and good locality, at the cost of repeated copying of long-lived objects. [Cheney, 1970; Blackburn et al., 2004]

Mark-Region

In a mark-region collector, memory is split into fixed-size regions into which objects are bump allocated. Objects are not permitted to cross region boundaries. Collection is performed using a basic tracing algorithm, where each region is marked as either alive or empty. Empty regions can then be returned to the allocator. Mark-region is not directly used in production collectors – rather, it is a theoretical foundation for the Immix collector [Blackburn and McKinley, 2008].

Immix

Immix splits memory into coarse-grained blocks and fine-grained lines. Objects are allowed to span multiple lines, but not multiple blocks. In order to minimise fragmentation and improve locality, Immix performs evacuation of old objects into new blocks, leaving behind a forwarding pointer. This pointer can then be used to correct any objects which have a reference to the moved object. The algorithm is designed so that evacuation is always performed the first time a reference to the object is seen during a collection, ensuring that all references can be corrected during the remainder of the collection [Blackburn and McKinley, 2008].

2.1.4 Generational garbage collection

The generational hypothesis states that most objects will die young. Therefore, it is desirable to avoid intensive memory management operations for newly allocated objects. Generational garbage collection [Lieberman and Hewitt, 1983; Ungar, 1984] seeks to collect short-lived objects as quickly as possible, while avoiding repeatedly

tracing long-lived objects. This is achieved by splitting memory into distinct generations, through which objects can be promoted as they age. Different allocators and collectors can be used for each generation to exploit the properties of different objects.

Blackburn and McKinley [2003] evaluate a generational collector which uses a bounded copying nursery (a tracing collector for young objects) with a reference counting mature space. They find that this provides high performance with good responsiveness.

2.1.5 Conservative garbage collection

Generally, implementing an exact tracing or deferred reference counting garbage collector is a significant engineering task. In order to enumerate all live objects from the roots, the compiler must know the type of all values in the stack at all points where a collection can occur, so that references to objects in the heap can be distinguished from values. This information must be propagated through all stages of compilation for all live objects, which is a difficult task which many language implementations, such as PHP, Objective C, Perl and several Javascript VMs do not attempt [Shahriyar et al., 2014].

As an alternative, some implementations use conservative garbage collection. When scanning the roots, conservative garbage collection identifies values that *might* be valid references to objects in the heap. For the purpose of liveness, these are treated as valid references, and the objects they point to will not be collected. However, these references must be treated carefully – for example, in a compacting collector, the referent object must not be moved, as this would modify the ambiguous reference. Thus, conservative collection significantly reduces engineering effort at the cost of increased fragmentation and possible retention of dead objects [Bartlett, 1988].

Conservative RC-Immix [Shahriyar et al., 2014] is a conservative version of the RC-Immix collector (discussed in 2.1.2) with slightly better performance than exact generational collectors. Conservative RC-Immix allows ambiguous references to pin objects with very fine granularity (using Immix's block and line heap layout) while retaining RC-Immix's compaction and fast collection of objects.

2.2 Reference counting in PHP

PHP is unusual in that the choice of garbage collector is heavily influenced by the semantics of the language. In this section, we explain why PHP5 and HHVM continue to implement naive reference counting for their garbage collection.

It is important to note that PHP has had no formal specification for most of its life. In July 2014, a draft specification for PHP was made available to the community. The initial draft was written by members of the HHVM team, before being adopted and open-sourced by the PHP group [Marcey, 2014; The PHP Group, 2014]. Prior to this specification, the behaviour of PHP5 was considered the de-facto standard, which has lead to unusual design choices and bugs introduced early in PHP's life

```
1  <?php
2
3  function array_modify($b) {
4      $b['hello'] = 'everyone';
5      return $b;
6  }
7
8  $a = array('hello'=>'world');
9  $b = array_modify($a);
10
11 var_dump($a); // array(1) { ["hello"]=> string(5) "world" }
12 var_dump($b); // array(1) { ["hello"]=> string(8) "everyone" }
```

Figure 2.1: Demonstration of pass-by-value semantics for PHP arrays

being adopted as features.[1]

2.2.1 Pass-by-value for arrays

One major motivation for naive reference counting is PHP's pass-by-value semantics for arrays. Consider the code snippet Figure 2.1, in which two copies of an array are created, one is modified, and then both are printed. In most programming languages (eg, C, Java, Python, Javascript) the single array object would be shared between both variables. In PHP, each variable has its own copy of the array which can be modified independently. The same semantics apply to function parameters and any other places arrays are used.

In order to optimise for the case where the array is copied but not modified, PHP engines will generally implement copy-on-write for arrays. In this case, the copy on line 8 of Figure 2.1 is delayed for as long as possible (in this case, until line 4), when the array is written to and the copies differ. If the array is not written to, then the copy will not be performed, saving both time and space.

In order to implement copy-on-write effectively, an accurate reference count for each array must be available at all times. That way, when an array is written to we can check the reference count to determine if it should be copied. If the reference count is two or more, we copy before the write, otherwise no copy is required. Many optimisations for reference counting collection (see Section 2.1.2) rely on making reference counts inaccurate some of the time, and thus are incompatible with this exact copy-on-write optimisation.

Prior to the writing of the draft PHP specification, Tozawa et al. [2009] attempted to formally specify PHP's copying semantics, and found that PHP's copying behaviour was internally inconsistent. The draft PHP specification makes explicit reference to deferred array copying as an optional optimisation, stating that:

[1] An example of this is PHP Bug #20993 [The PHP Group, 2002]:

> "We have discussed this issue and it will put a considerable slowdown on PHP's performance to fix this properly. Therefore this behavior will be documented."

> In practice an application written in PHP may rely on value assignment of arrays being relatively inexpensive for the common case (in order to deliver acceptable performance), and as such it is common for an implementation to use a deferred array copy mechanism in order to reduce the cost of value assignment for arrays. [The PHP Group, 2014]

Additionally, the specification allows for several choices to be made by implementations which can result in observably different results when performing deferred array copying, both as compared to eager array copying and as compared to other conforming implementations of deferred array copying.

In summary, PHP's array copy-on-write optimisation has implications for both performance and correctness, and must be considered when attempting to remove naive reference counting from PHP.

2.2.2 PHP references

PHP's userspace reference variables are also designed around the presence of precise reference counts. Consider the example in Figure 2.2. In program 2.2(a), the following sequence of events occurs:

1. Array `$a` is created
2. Variable `$v` is created, which turns `$a[1]` into a reference variable.
3. Array `$b` is created, and creates a copy of the array elements in `$a`. Therefore, `$b[1]` is *also* a reference to the original 'world' string.
4. When `$b[1]` is modified, the change can be seen in both `$a` and `$v`

Program 2.2(b) differs as follows:

1. On line 6, `$v` is unset, which causes the reference count of the 'world' string to drop to 1. Therefore, PHP is able to demote this reference back to a value.
2. On line 11, `$b[1]` is a value, not a reference. Therefore, the change made here is not visible to `$a`.

The final states of these programs as seen by the PHP interpreter is shown seen in Figure 2.3.

Note that the difference in behaviour between the two programs is entirely due to the interpreter knowing that `$a[1]` has a reference count of 1 after line 6 of program 2.2(b). If exact reference counts were not available, this demotion would not be possible, and program 2.2(b) would have the same behaviour as program 2.2(a).

The draft PHP language specification agrees that this behaviour is a part of the PHP language. However, the abstract memory model described by the specification does not have a notion of a reference variable in the same way that PHP5 or HHVM do.

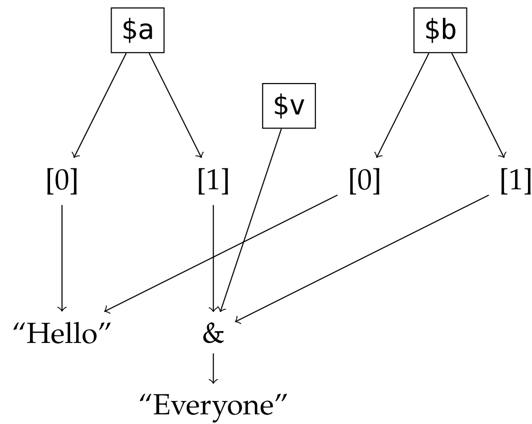
```php
<?php

$a = array("hello", "world");
$v =& $a[1];

xdebug_debug_zval_stdout('a');

$b = $a;
$b[1] = 'everyone';
xdebug_debug_zval_stdout('a');

// Output:
// a: (refcount=1, is_ref=0)=array (
//     0 => (refcount=1, is_ref=0)='hello',
//     1 => (refcount=2, is_ref=1)='world'
//    )(103)
// a: (refcount=1, is_ref=0)=array (
//     0 => (refcount=2, is_ref=0)='hello',
//     1 => (refcount=3, is_ref=1)='everyone'
//    )(106)
```

(a) The existence of `$v` means that modifications to `$b` are visible to `$a`

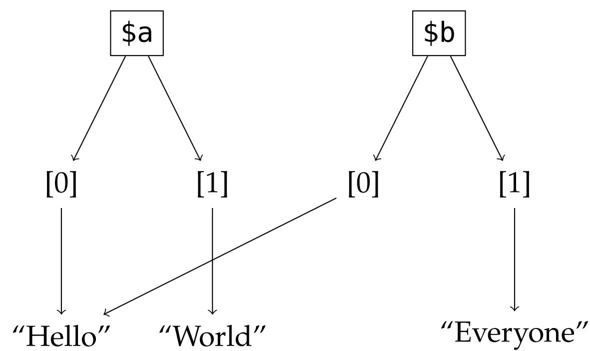
```php
<?php

$a = array("hello", "world");
$v =& $a[1];

unset($v);

xdebug_debug_zval_stdout('a');

$b = $a;
$b[1] = 'everyone';
xdebug_debug_zval_stdout('a');

// Output:
// a: (refcount=1, is_ref=0)=array (
//     0 => (refcount=1, is_ref=0)='hello',
//     1 => (refcount=1, is_ref=0)='world'
//    )(103)
// a: (refcount=1, is_ref=0)=array (
//     0 => (refcount=2, is_ref=0)='hello',
//     1 => (refcount=1, is_ref=0)='world'
//    )(103)
```

(b) By unsetting `$v`, `$a[1]` is demoted to a value, and thus modifications to `$b` are not visible to `$a`

Figure 2.2: Demonstration of reference demoting in PHP



(a) Final state of program 2.2(a). Note that `$a[1]` is a reference to a string, and has a refcount of 3.



(b) Final state of program 2.2(b). Note that `$a[1]` and `$b[1]` point to different values.

Figure 2.3: Final states of the programs in Figure 2.2

Internally, PHP5 marks reference variables using the `is_ref` flag, while HHVM uses a separate `RefData` data structure for these variables. Because of this explicit difference in representation between values and references, demotion must be explicitly performed.

However, in the draft specification's abstract memory model, a reference is any `VStore` (stack storage location) aliased by two or more `VSlots` (variable slots). When the number of referring `VSlots` drops to 1, the variable is naturally no longer treated as a reference, since only one variable can see the `VStore`. While this interpretation of PHP's memory model is not tied to naive reference counting, common implementations of this feature (both PHP5 and HHVM) are. Therefore, reference demotion must also be considered when removing reference counting from PHP.

2.2.3 Precise destruction

One benefit of using naive reference counting for garbage collection is that garbage can be collected immediately after it is no longer in use. In the case of PHP objects, this means that their destructor will be run immediately, at a time that may be predicted by the application programmer. The draft PHP specification explicitly states that memory does not need to be reclaimed immediately:

> The engine may reclaim a VStore or HStore at any time between when it becomes eligible for reclamation and when the script exits. [The PHP Group, 2014]

However, PHP5's long-standing status as the de-facto standard for PHP means that precise destruction of PHP objects (which PHP5 provides) may be relied upon by existing PHP programs. Therefore, any PHP implementation wishing to reach full compatibility with PHP5 must consider this issue.

2.3 The HipHop Virtual Machine

HHVM provides both a high performance interpreter and a JIT compiler for PHP. A major goal for the project is to reach complete parity with the canonical PHP5 runtime. Exact parity is difficult to define, as HHVM must replicate all bugs and other unusual behaviour from PHP5. At the time of writing, 26 open source PHP frameworks pass 100% of their tests under HHVM, with an overall pass rate of 98% across all frameworks tested [Facebook, 2014].

HHVM's compilation pipeline consists of two phases. First, in the ahead-of-time phase (either at the start of a request or during deployment to a production server), PHP is parsed into HipHop bytecode (HHBC), and some global optimizations are performed. In production mode, this bytecode can be written to a bytecode repository for reuse between requests. Then, in the just-in-time phase, this bytecode is used by the interpreter and the JIT compiler to execute a request.

The JIT compiler operates on the concept of a 'Tracelet', which is a single-entry, multiple-exit piece of code, typically representing a small number of lines of PHP,

rather than an entire method. PHP's type inference is undecidable, and therefore HHVM must be conservative about what types are able to pass through each tracelet. HHVM initially compiles each tracelet using currently available type information. If the tracelet is later executed with different input types, the tracelet can be compiled again, with a chain of compiled tracelets forming for each combination of input types. The actual JIT compilation is performed using a machine-independent, static single assignment (SSA) intermediate representation (HHIR), on which machine-independent optimisations can be performed, followed by further optimisations and machine code generation for the required architecture [Adams et al., 2014].

HHVM uses a region-heap (reap) approach to memory management, where request-local objects are allocated onto a region of memory (using a free-list allocator). These objects can then be individually released, or left until the end of the request, at which point the entire region is released. Garbage collection is performed using naive reference counting, with an optional optimisation pass to eliminate unnecessary reference count increment and decrement operations during JIT compilation. There is currently no cycle collector – instead, cyclic objects are left to be collected by the reap at the end of each request.

2.4 Related work

We now investigate other similar attempts to create high performance PHP runtimes. Where possible we identify what garbage collection techniques are used by these runtimes, however, we are not aware of any previous work which focuses on improving garbage collection for PHP.

2.4.1 PHP compilers

A common approach to high performance PHP has been to compile PHP scripts to lower-level languages, or directly to machine code.

HPHPc HHVM was built upon a related project from Facebook, the HPHPc compiler. HPHPc takes a very different approach to high performance PHP, by compiling a complete PHP codebase into C++, which can then be compiled and optimised by `gcc`. Areas of the code which can be statically analysed are almost directly translated to C++, while PHP's dynamic features are simulated by a number of symbol tables which are built by HPHPc [Zhao et al., 2012]. Garbage collection is performed using naive reference counting. Estimates performed using Facebook workloads indicated that HPHPc achieved 5.8 times more throughput than PHP5. HPHPc has now been superseded by HHVM, which has higher performance and greater compatibility with PHP5.

`phc` `phc` [de Vries et al., 2011] takes a similar approach of compiling PHP to C, which is then compiled by `gcc`. The major difference is that `phc` takes advantage of the PHP5 C API at both compile and run time, to provide types, memory management,

symbol tables and the PHP standard library in a manner consistent with PHP5. This significantly reduces the engineering effort required as many features do not need to be reimplemented. Garbage collection is performed using PHP5's naive reference counting API.

In 2009, `phc` had achieved 1.5 to 2x performance increase compared to PHP5. However, it is no longer actively developed, and is incapable of handling all modern PHP features [Biggar and Gregg, 2009].

Roadsend Roadsend [Roadsend, 2014] was a compiler written in Scheme that translated PHP directly into machine code. The project has been unmaintained since 2010, and performance and compatibility are unclear.

Summary This pre-compilation approach is able to provide good performance, at the cost of slowing PHP's quick development cycle and loss of some features (such as `eval`).

2.4.2 Targeting existing virtual machines

PHP's nature as a highly dynamic language lends itself to just-in-time (JIT) compilation. JIT compilers are able to make use of additional information that cannot always be determined statically (for example, dynamic loading of source files). A common approach has been to build a PHP front-end for existing virtual machines and JIT compilers, in order to reduce the engineering effort required.

HappyJIT HappyJIT [Homescu and Şuhan, 2011] is a JIT-compiled interpreter written using RPython as a replacement front-end for PyPy. Work is saved by reading in PHP5 bytecode saved by the Advanced PHP Cache instead of parsing the PHP source from scratch. Garbage collection is performed using PyPy's builtin algorithms, with reference counting for PHP references implemented on top of this. Overall, they report a moderate performance improvement compared to PHP5 across a suite of microbenchmarks, but are unable to support full PHP web applications.

P9 The P9 PHP processor Tatsubori et al. [2010] replaces the Java-specific areas of the IBM TR JIT compiler framework with PHP parsers, code generators and optimisers. Garbage collection is performed using naive reference counting. A small number of PHP extensions were also ported to P9, to allow interaction with web servers and databases. Overall, the approach gives favourable results, with a 20–30% increase in concurrent sessions compared to PHP5 in the SPECweb2005 benchmark.

Phalanger Similarly, Phalanger [Benda et al., 2006] is a compiler for PHP targeting the .NET Common Language Runtime. Phalanger supports most PHP 5.3/5.4 features, as well as existing PHP extensions through the PHP5 C API. Additionally, Phalanger supports interaction with other code running on the .NET platform (C#,

VB.NET, etc). This is achieved with a performance improvement of approximately 2x compared to PHP5.

Summary Creating a JIT compiler by targeting an existing virtual machine reduces the engineering effort required, but makes PHP-specific considerations more difficult. Many of these runtimes address the issues with PHP's semantics identified in Section 2.2 by either using naive reference counting for garbage collection, or adding reference counting where required on top of an existing garbage collector (which will incur a significant overhead). In addition, the vast PHP standard library and extension framework make reaching total compatiblility with PHP5 a difficult task.

2.5 Summary

In this chapter, we have outlined several modern approaches to memory management, discussed the semantics of PHP which tie it to naive reference counting, and given a high level overview of HHVM and other similar PHP runtimes. In Chapter 4, we build upon this by investigating how suited naive reference counting is to the memory characteristics of HHVM and PHP applications. First, in the next chapter we describe the experimental methodology used throughout the remainder of the thesis.

Chapter 3

Experimental Methodology

This chapter describes the software, hardware and benchmarks used to experimentally evaluate the changes made to HHVM in the remainder of this thesis.

3.1 Software platform

The experiments performed in the remainder of this thesis are built upon HHVM version 3.1, released May 2014 [Tarjan, 2014]. A small number of changes were backported from more recent versions to fix bugs and compiler issues encountered during implementation. Further details on obtaining and compiling this version of HHVM (`HHVM-3.1-update`), as well as all of the other patches described in the remainder of this thesis, can be found in Appendix A.

We use the server version of Ubuntu 14.04 LTS with version 3.13.0-32 of the Linux kernel.

3.2 Benchmarks

We use two main sets of benchmarks to experimentally evaluate the changes made to HHVM. Firstly, we use a set of 37 artificial microbenchmarks. This is the same suite used in Adams et al. [2014], which includes the Programming Language Shootout benchmarks (`shootout`), and a `vm-perf` suite of benchmarks specifically designed to test scenarios encountered by large PHP programs.

Secondly, we test against the popular open-source PHP applications WordPress and MediaWiki. Both have full compatibility with HHVM, and are representative of the real-world use of PHP.

WordPress We test against WordPress 4.0, installed using default settings. We import a dataset ([BetterWP.net, 2011]) containing 2000 randomly generated blog posts and pages to simulate a medium-sized WordPress website. We test the index page, individual posts and search functionality.

MediaWiki We test against MediaWiki 1.23. We use a recent database dump of the English language WikiNews to test performance with a real-world dataset. We disable caching and test the `Special:Random/Category` page, as we find this to be a better indicator of performance (particularly with arrays) than rendering a simple wiki page.

For both applications, we use the `siege` HTTP benchmarking tool [Fullmer, 2012] to measure the end-user response time and throughput of the HHVM server.

3.2.1 Compiling and running

We run tests using HHVM's production (`Repo.Authoritative`) mode. In this mode, an initial compilation stage is used to transform the PHP source into an optimised bytecode repository. Then, when running the program, this repository is treated as a single, unchanging source of code – no checks will be performed at runtime to see if the PHP source has changed, improving performance.

We run each benchmark four times (once as an unmeasured warmup run) and report the minimum of the final three runs. Benchmarks are run with a 12 GB memory limit and 15 minute time limit, enforced using `runlim` [Biere and Jussila, 2011].

A further difference between the methodology used for benchmarks and open source applications is that the applications are run using a single, warmed-up server process. This means that JIT code is generated once during the warm-up run, and then is reused for the test runs of the application. On the other hand, microbenchmarks are run in CLI mode, and perform JIT compilation on every run of the benchmark.

3.3 Hardware platform

Experiments are performed on a 3.7GHz AMD A10-7850K processor. This is a quad-core processor with four 16KB L1 data caches, two shared 96KB L1 instruction caches and two shared 2MB L2 caches.

Chapter 4

Memory Management in HHVM

In Chapter 2, we describe the state of memory management research as a whole, along with a discussion of how memory management fits into the semantics of PHP. We now explore the memory management approach taken by HHVM in more detail. We perform a variety of experiments on HHVM to determine how effective reference counting is, and whether other garbage collection algorithms are likely to provide increased performance.

Different garbage collection algorithms perform differently depending on the memory usage patterns of the runtime. For example, generational garbage collectors perform well when objects have a short lifetime. Therefore, it is desirable to understand the general memory characteristics of HHVM.

In order to achieve this, we created an instrumented build of HHVM (described in Section 4.1), and use it to measure the lifetime (Section 4.2), size (Section 4.3), and maximum reference count (Section 4.4) of objects within HHVM. In Section 4.5 we discuss the heap composition of HHVM applications, and in Section 4.6 evaluate HHVM's reference count eliding optimisation before summarising in Section 4.7.

Note that the experiments performed in this section are conceptually similar to those performed by Jibaja et al. [2011], who evaluate the memory characteristics of PHP5. This chapter can be considered a verification that HHVM shares many of the same characteristics as PHP5.

4.1 Instrumented HHVM build

We have created an instrumented build of HHVM named `refcount-survey` which records each of the following actions:

- Memory is allocated through the smart allocator
- Memory is freed through the smart allocator
- A reference count is changed

We are then able to track reference count mutations across the lifetime of an object. We collect a range of statistics about object sizes, lifetimes and reference counts in-memory, and output per-request and cumulative reports at the end of each request.

4.2 Lifetime of memory-managed objects

By tracking the difference between when an object was allocated, and when it was freed, we are able to measure the lifetime of objects within HHVM. The results of this measurement across typical WordPress and MediaWiki requests are shown in Figure 4.1. Note that the x-axis of these graphs uses bytes of memory allocated as an abstraction of time, as clock time is difficult to measure accurately at this scale. Figure 4.1(a) shows the percentage of objects which survive for up to 256 kB of allocation, while Figure 4.1(b) zooms in to show objects which survive for less than 1 kB of allocation. Both applications allocate significantly more than 256 kB to serve a request: WordPress requests average approximately 27 MB and MediaWiki requests average approximately 38 MB.

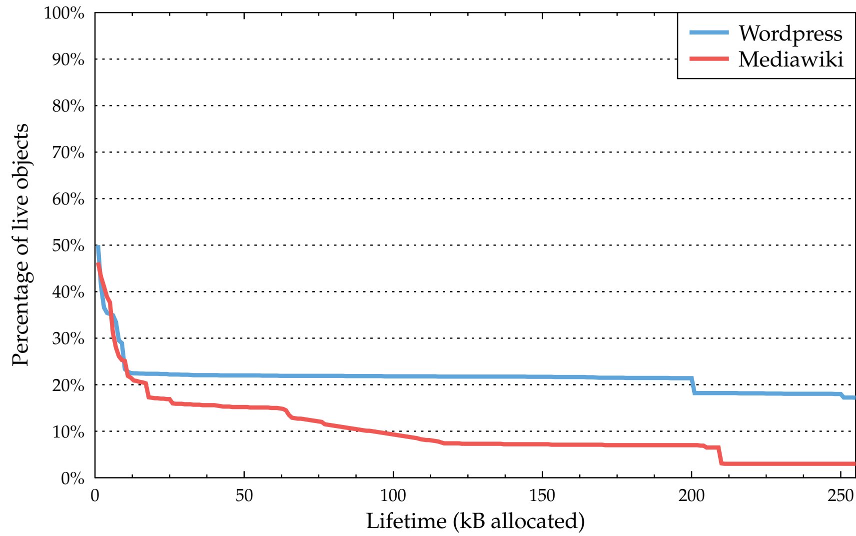
From this data, it is evident that many objects in HHVM are short-lived. 15–20% of objects have a lifetime of less than 30 bytes, indicating that they are collected before any other object is allocated. Approximately 50% of objects survive for less than 1kB of allocation. Additionally, 20% of WordPress and 3% of MediaWiki objects survive for more than 256 kB of allocation. It is likely these objects are global and static variables which will never be collected.

As naive reference counting is able to collect these objects immediately, these short-lived objects will have their destructors run and memory claimed as soon as they are no longer used. This behaviour means that HHVM is able to run with a reduced memory footprint. However, this means that HHVM is doing additional work maintaining reference counts in the common case where an object needs to be collected. Alternative collectors, such as SemiSpace, do work proportional to the number of survivors, and may provide better performance (at the cost of additional memory usage) for HHVM, where the number of survivors is low.

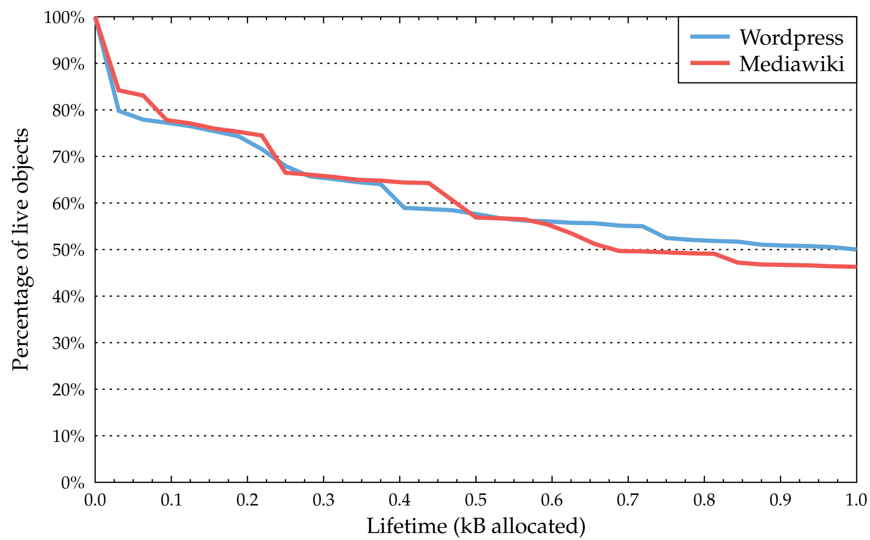
4.3 Size of objects

The refcount-survey build allows us to track the size of objects allocated within HHVM. We show the frequency of different object sizes for WordPress and MediaWiki in Figure 4.2, with a cumulative frequency plot in Figure 4.3. We have also calculated the mean and median for each benchmark, shown in Table 4.1. Note that we track object sizes in 16 byte buckets (in line with HHVM's 16 byte alignment for allocations), and consider all objects larger than 2 kB as a single bucket.

Both WordPress and MediaWiki are skewed towards small objects, with a median size bucket of 48 bytes (that is, objects between 33 and 48 bytes). Futher, over 70% of objects are less than 96 bytes in size for both applications. The remaining 30% is a

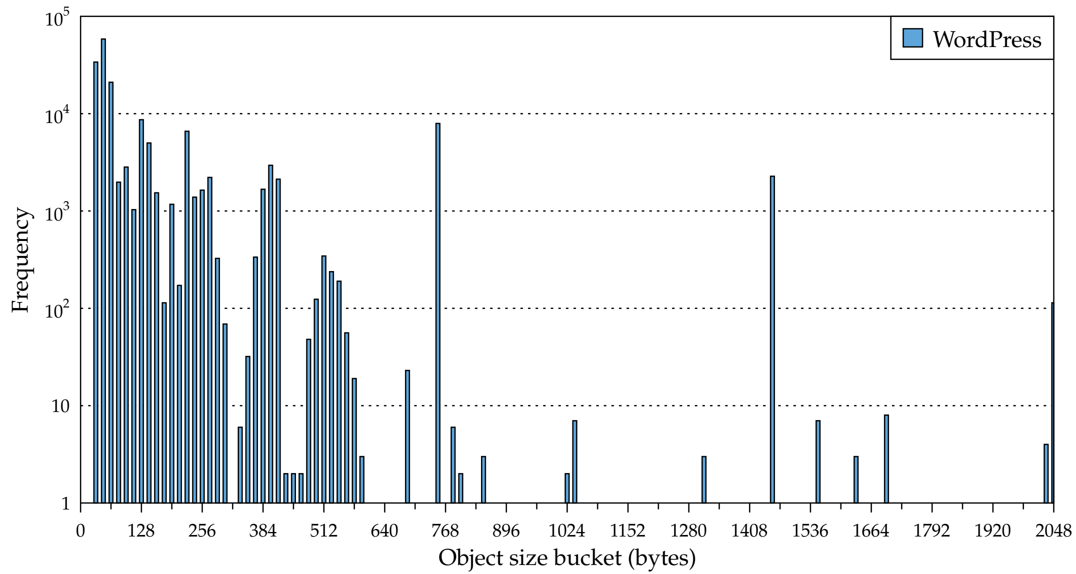


(a) Percentage of objects which live for a certain amount of time after their allocation

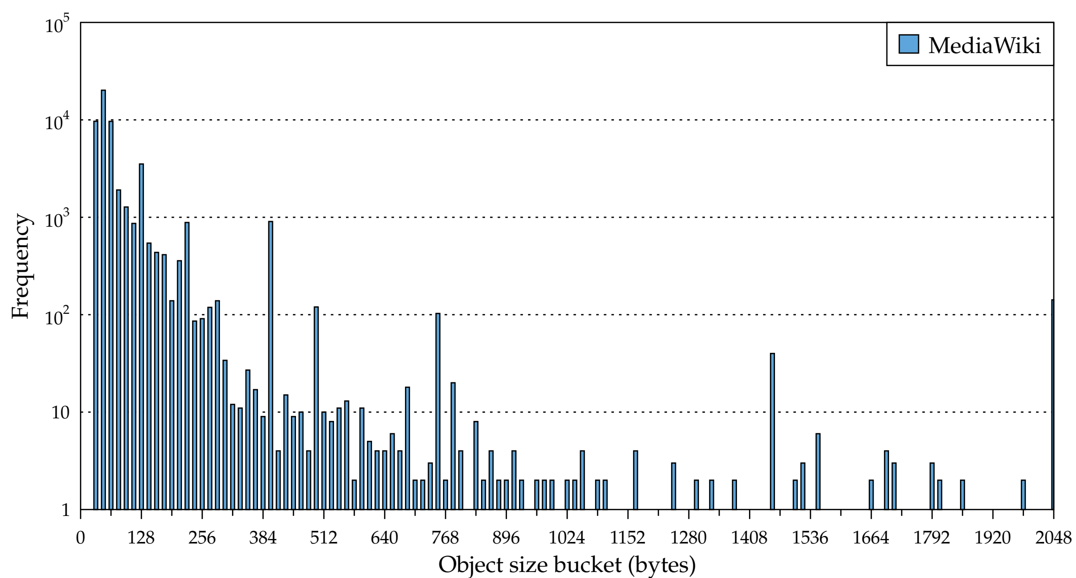


(b) Percentage of objects which live for a certain amount of time after their allocation, zoomed in to display very short lifetimes

Figure 4.1: Lifetime of objects within HHVM



(a) Object sizes for a typical WordPress request



(b) Object sizes for a typical MediaWiki request

Figure 4.2: Object size demographics within HHVM

Benchmark	Mean object size (bytes)	Median object size bucket (bytes)
Wordpress	86	48
Mediawiki	144	48
shootout/binarytrees_1.php	80	80
shootout/binarytrees_2.php	80	80
shootout/binarytrees_3.php	64	64
shootout/fannkuchredux_2.php	206	208
shootout/fasta_2.php	65	64
shootout/fasta_3.php	32	32
shootout/fasta_4.php	65	64
shootout/knucleotide_1.php	86	96
shootout/knucleotide_4.php	39	32
shootout/nbody_3.php	56	32
shootout/nbody_3a.php	56	32
shootout/pidigits_1.php	820	32
shootout/regexdna_1.php	56	32
shootout/regexdna_2.php	56	48
shootout/regexdna_3.php	56	48
shootout/regexdna_4.php	57	32
shootout/revcomp_1.php	95	96
shootout/revcomp_2.php	95	96
shootout/spectralnorm_2.php	58	32
shootout/spectralnorm_3.php	51	32
vm-perf/big.php	198	128
vm-perf/cache_get_scb.php	89	48
vm-perf/center-of-mass.php	80	80
vm-perf/dyncall.php	56	32
vm-perf/fib.php	56	32
vm-perf/fibr.php	56	32
vm-perf/hopt_preparable.php	56	32
vm-perf/mixedbag_loop.php	57	32
vm-perf/nbody.php	56	32
vm-perf/obj-fib.php	56	32
vm-perf/perf-ad-hoc-hack.php	56	32
vm-perf/spectral-norm.php	58	32
vm-perf/t-test.php	76	32

Table 4.1: Mean and median object size for open source applications

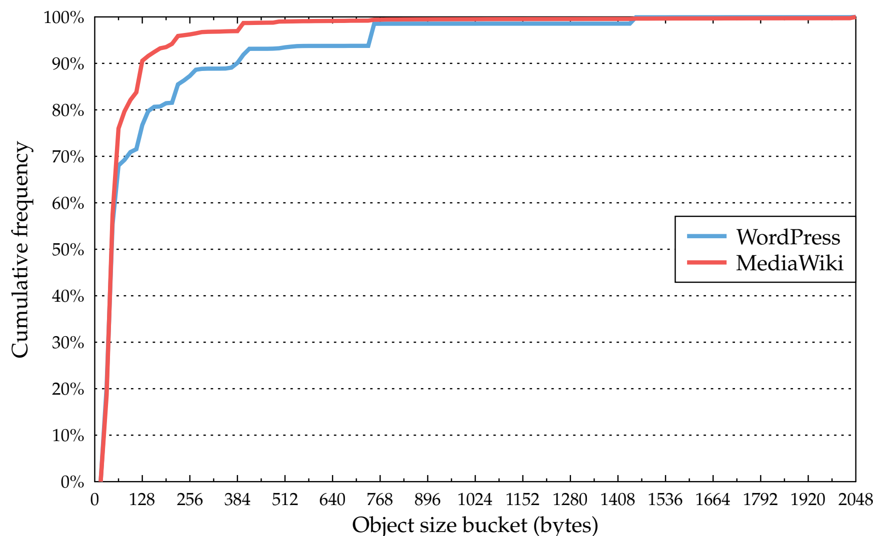


Figure 4.3: Cumulative frequency of object sizes in open source applications

long tail of larger objects. Jibaja et al. [2011] perform a similar evaluation for PHP5, and find a median object size for a number of benchmarks (including WordPress) of 20–28 bytes. Similarly, the DaCapo Java benchmark suite has mean object sizes of 20–60 bytes, excluding outlier benchmarks which allocate particularly large objects [Blackburn et al., 2006]. Only two benchmarks we test have a mean object size under 40 bytes, while most have a mean object size of 50–80 bytes. While these results aren't directly comparable, they indicate that HHVM has larger objects than PHP5 and Java.

Part of this difference in object size between HHVM and PHP5/Java can be attributed to CPU architecture. Our evaluation is performed on a 64-bit virtual machine, while the evaluations in Jibaja et al. [2011] and Blackburn et al. [2006] were both performed on 32-bit virtual machines. Venstermans et al. [2006] show that objects in 64-bit Java Virtual Machines are on average 40% larger than those in 32-bit Virtual Machines. It is likely that a similar result holds for PHP5.

Small objects are desirable when using garbage collectors such as Immix, since many objects can fit into a single block, and they can be compacted effectively into small gaps in the heap. HHVM's larger mean object size means that experimentation may be required to find effective block and line sizes for Immix.

4.4 Maximum value of reference counts

HHVM uses a 32-bit integer to hold reference counts, with a maximum possible value of $2^{30} - 1$ (the additional bits are used to determine whether an object is static or uncounted). It is effectively impossible for an object to reach this value, as the

referring objects would fill gigabytes of heap space. For comparison, Java garbage collectors have seen significant performance gains from limiting reference counts to just 3 bits [Shahriyar et al., 2012].

We used refcount-survey to measure the maximum size of reference counts in HHVM. We count the frequency of different reference count sizes across requests to WordPress and MediaWiki. The results of this experiment are displayed in Figure 4.4. Figure 4.5 shows that almost all (99.7% for MediaWiki, 99.99% for WordPress) of objects can be accounted for with a reference count of just 3 bits. Overflowing reference counts can be fixed using a backup hashtable to store accurate counts, by occasionally tracing the heap to correct and/or collect these objects, or by ignoring them until they are collected automatically at the end of the request.

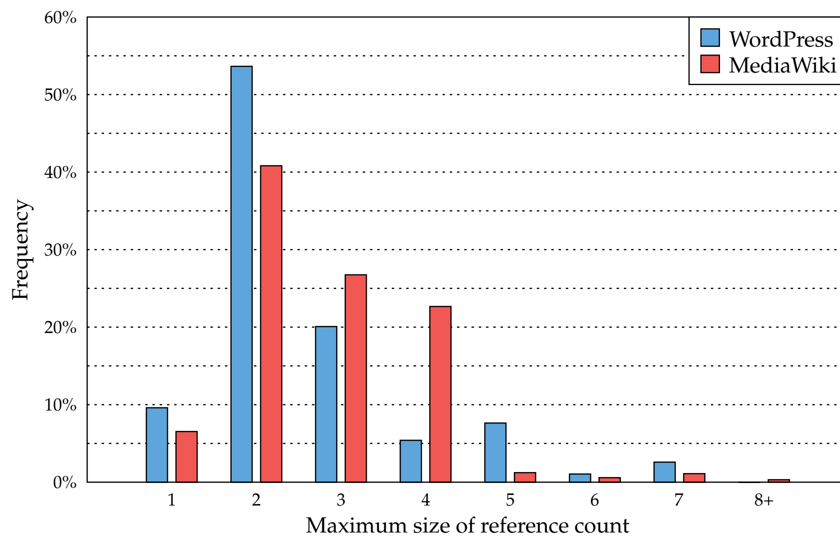


Figure 4.4: Maximum size of refcounts within HHVM

As mentioned below in Section 4.5, a large proportion of HHVM's heap is used by object headers. Shrinking these object headers would potentially provide a significant reduction in memory usage and response time. However, HHVM aligns all allocations to 16 byte boundaries. Therefore, although it is possible to save almost 4 bytes by reducing the size of the reference count field, this would need to be paired with several other significant gains in order to have any performance benefit.

4.5 Heap usage

HHVM has five primary reference counted datatypes: Array, Object, String, Reference and Resource. It is useful to understand how the heap of a typical PHP program is broken down in terms of these types, as it gives us an understanding of what

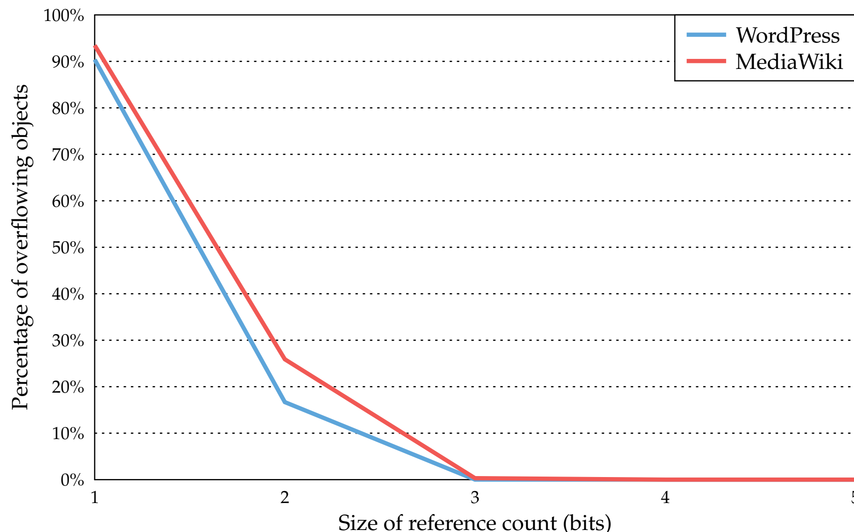
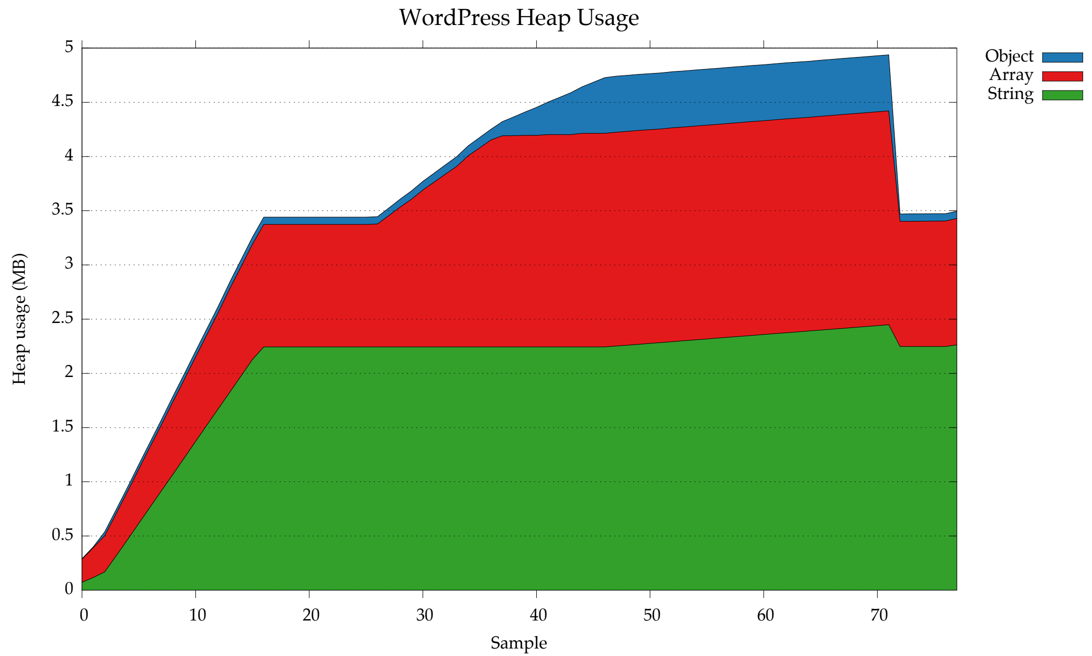


Figure 4.5: Bits required to store reference counts. A 3-bit reference count field would suffice for 99.7% of objects

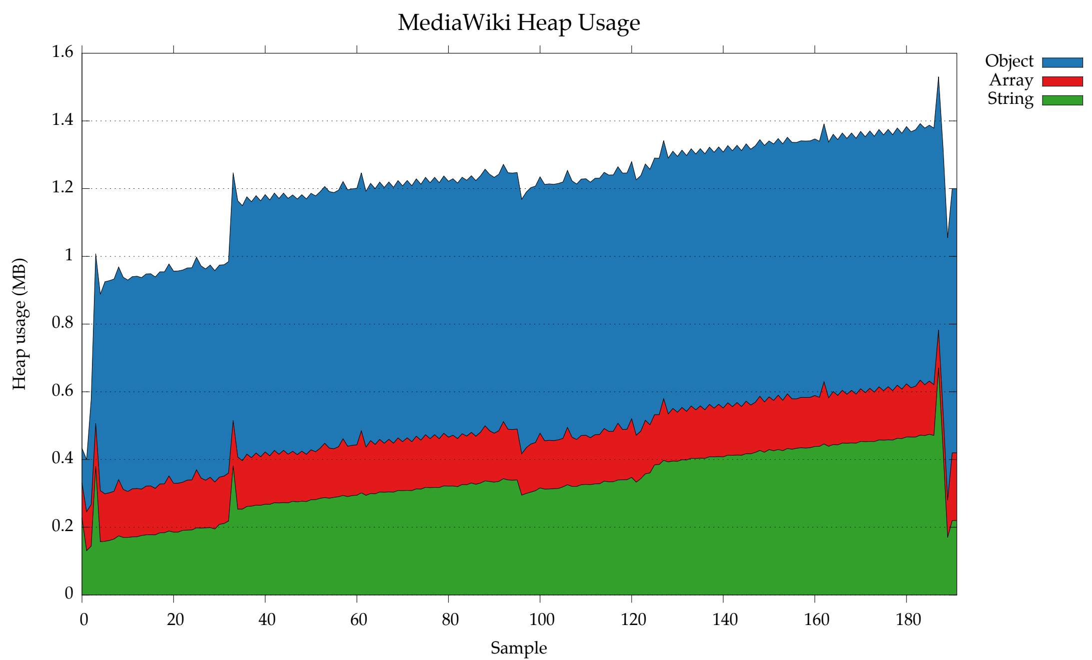
optimization techniques will be the most effective. We created a build of HHVM (heap-stats) which periodically (shortly after every new slab of memory is allocated – see Section 6.2.2 for further details) traces the heap to determine the type breakdown of active memory. In this way, we are able to plot memory usage over the course of a request. We run this build over several requests to MediaWiki and WordPress, and plot the results in Figures 4.6 and 4.7.

Figures 4.6(a) and 4.6(b) show the amount of memory used by strings, arrays and objects over the course of a request to WordPress and MediaWiki (respectively). Note that reference and resource objects were also measured, but are not displayed due to very small counts. Wordpress consistently has approximately 100 live references and 2 live resources (less than 1% of the heap), while Mediawiki has between 250 and 300 live references and 1 live resource (slightly more than 1% of the heap). We can see that WordPress makes heavier use of arrays than objects, and has a larger working set than MediaWiki. This highlights that objects, strings and arrays are all common within HHVM programs. These three types have very different semantics, and we must be careful to optimise for all three cases.

Figures 4.7(a) and 4.7(b) break each of the types into header and data sections, plotting as a percentage of the entire heap. At any point in time, headers take up 20–30% of the heap. Finding ways to shrink this overhead could potentially provide a significant performance benefit.

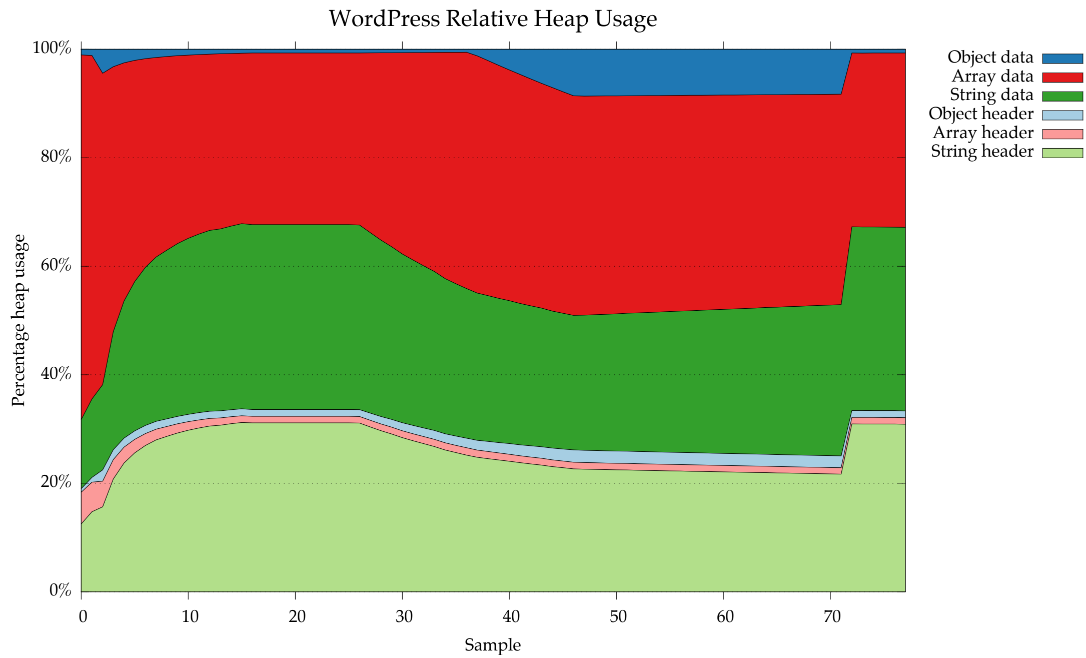


(a)

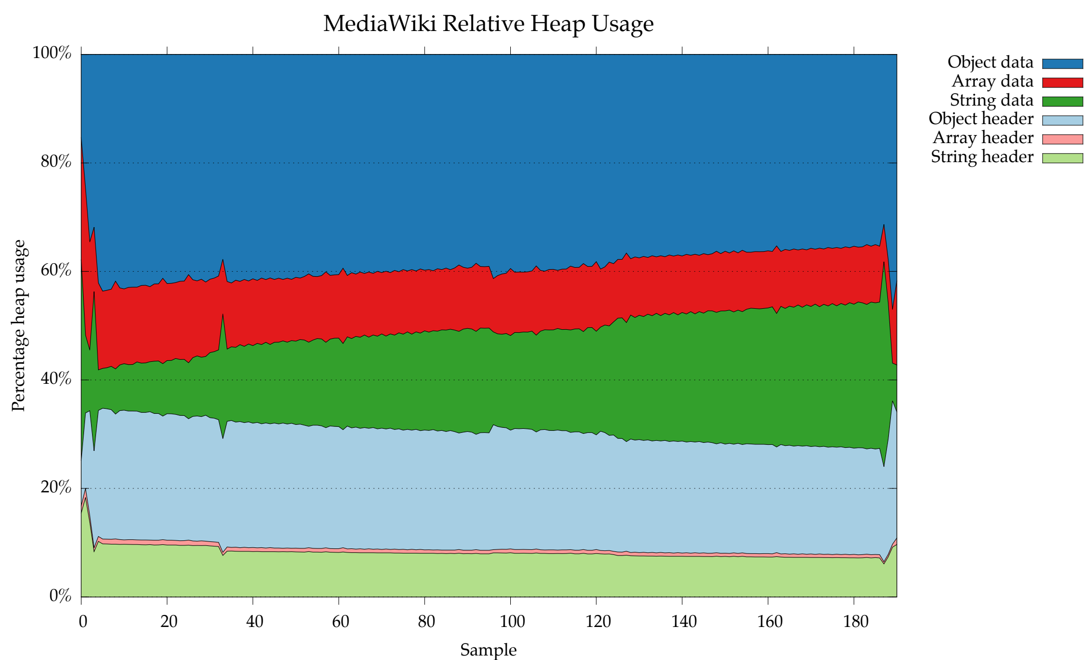


(b)

Figure 4.6: Heap usage of HHVM over the course of a WordPress and MediaWiki request



(a)



(b)

Figure 4.7: Heap usage of HHVM over the course of a WordPress and MediaWiki request, with each type split into header and data sections to show the amount of heap used by headers

4.6 Reference count eliding

HHVM includes an optional (but enabled by default) optimisation pass during JIT compilation which attempts to elide corresponding pairs of `IncRef` and `DecRef` instructions. This optimisation is particularly useful for removing reference count mutations caused by manipulating the evaluation stack (which is a primary purpose of deferred reference counting) [Simmers, 2014]. This is a basic version of the nondeferred reference counting described in Section 2.1.2. We experimentally evaluate the effect of this optimisation by comparing `HHVM-3.1` without reference count eliding (a simple configuration change) to the default `HHVM-3.1` configuration. The results of this experiment are shown in Figure 4.8.

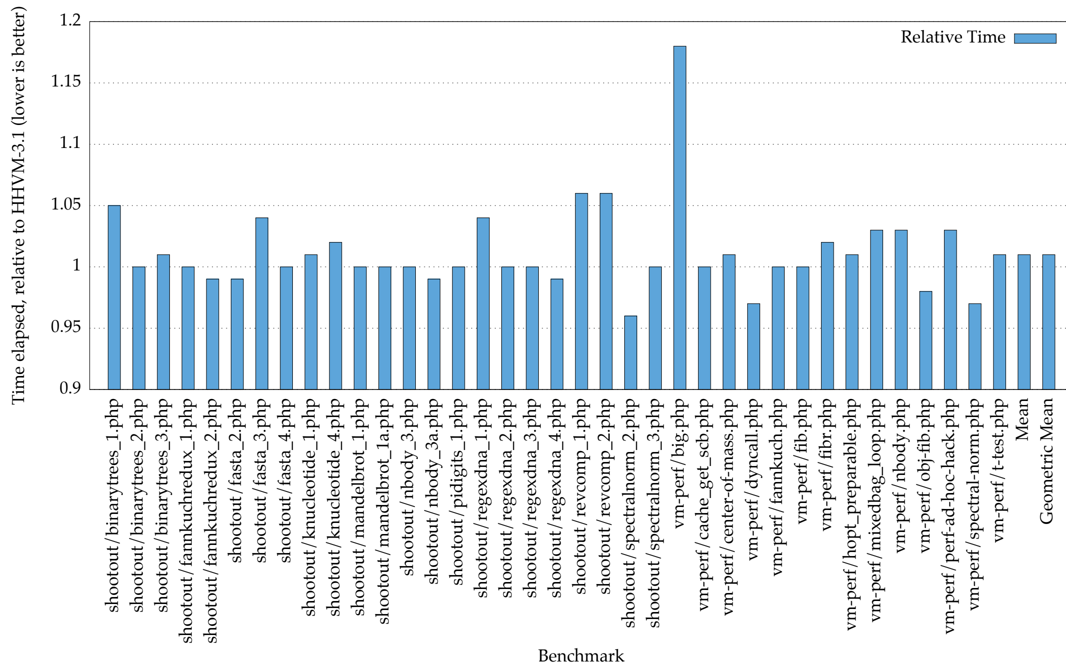
These results are somewhat mixed. Several benchmarks run slightly faster when reference count eliding is turned off. This indicates that the increase in JIT compilation time outweighs the benefit of removing reference count mutations. On the other hand, several benchmarks run slightly slower without reference count eliding, while `vm-perf/big.php` sees a particularly large performance drop. This benchmark involves passing a single String object through a three-level hierarchy of function calls. Each test execution involves over 1000 function calls with the string as an argument, and this process is repeated many times in a tight loop. This involves repeatedly pushing and popping the string from the evaluation stack, which (as mentioned above) is a case which refcount eliding is particularly good at optimising.

MediaWiki loses approximately 2% of its throughput when the refcount optimisations are disabled, while the throughput for WordPress does not change significantly. Note that unlike for the microbenchmarks, when running the application benchmarks we are able to warm up the server so that JIT compilation is performed before measurements are taken.

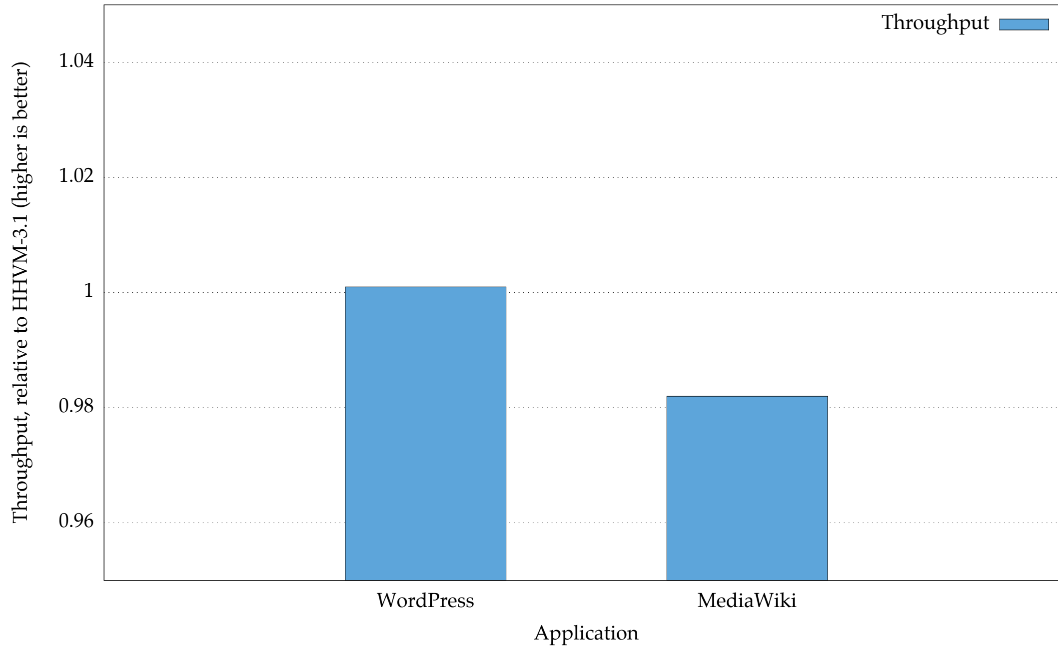
Overall, we have found that the reference count eliding optimisation is capable of providing a measurable performance benefit, particularly in the case of a long-running server application such as MediaWiki. This optimisation is able to partly account for the lack of deferred or coalescing reference counting in HHVM.

4.7 Summary

In this chapter, we have found that HHVM allocates many small, short-lived objects. These demographics are somewhat similar to PHP5 and Java programs [Blackburn et al., 2006; Jibaja et al., 2011; Shahriyar et al., 2012]. The use of exact reference counting allows HHVM to collect these objects immediately, and therefore run with a reduced memory footprint. However, languages such as Java have found great success by moving to high performance generational tracing garbage collectors. These results suggest that such a change might be viable for HHVM. In the next chapter, we explore the other side of this proposed change, by eliminating the areas in which HHVM depends on naive reference counting.



(a) Performance of HHVM-3.1 without refcount eliding on microbenchmarks



(b) Performance of HHVM-3.1 without refcount eliding on open source applications

Figure 4.8: Performance of HHVM-3.1 without refcount eliding compared to default HHVM-3.1

Chapter 5

Eliminating Reference Counting from HHVM

In Chapter 4, we identify that HHVM has memory characteristics which are well suited to advanced tracing garbage collection algorithms. However, the memory characteristics are only half of the story. In Section 2.2 we explained the semantics of PHP which tie HHVM to naive reference counting. We now explore the options which are available to break this dependency and remove reference counting from HHVM.

In Section 5.1, we describe and evaluate an alternative to copy-on-write ("blind copy-on-write") which does not depend on reference counts. Then, in Section 5.2 we remove reference counting operations and measure the associated performance gain. Finally, in Section 5.3 we describe the remaining compatibility issues with this version of HHVM.

5.1 Copy-on-Write

As discussed in Section 2.2.1, PHP implementations commonly use Copy-on-Write to reduce needless copying caused by pass-by-value semantics for arrays. However, copy-on-write relies on the presence of exact reference counts to know when to copy an array. If reference counts are incorrect or unavailable, false positives (copying when it is not necessary) or false negatives (failing to copy when it is required, resulting in unintended pass-by-reference semantics) will be introduced.

We propose an alternative form of copy-on-write, known as *Blind Copy-on-Write* which does not require reference counts in Section 5.1.1. We then experimentally evaluate this weakened optimisation in Section 5.1.2.

5.1.1 Blind Copy-on-Write

Blind Copy-on-Write is a weakened copy-on-write (COW) optimisation which does not require reference counts. As in regular COW, copying an array is deferred from when the array is assigned to when it is written. Blind COW, however, will *always*

Action	COW	Blind COW	COA
Array assignment	Point new variable to existing data and increment refcount	Point new variable to existing data	Copy existing data and point new variable to the copy
Write to array	Copy before writing if reference count is two or more	Always copy before writing	Never copy
Read from array	Read from array	Read from array	Read from array

Table 5.1: Summary of the behaviour of different array optimisations

copy when a write operation is performed, even if there is only one reference to it. Therefore, Blind COW will have poor performance when a single array is written to repeatedly, but will still optimise for the case where an array is assigned but never written to. Table 5.1 describes the difference between COW, Blind COW and unoptimised copy-on-assignment (COA).

5.1.2 Experimental evaluation

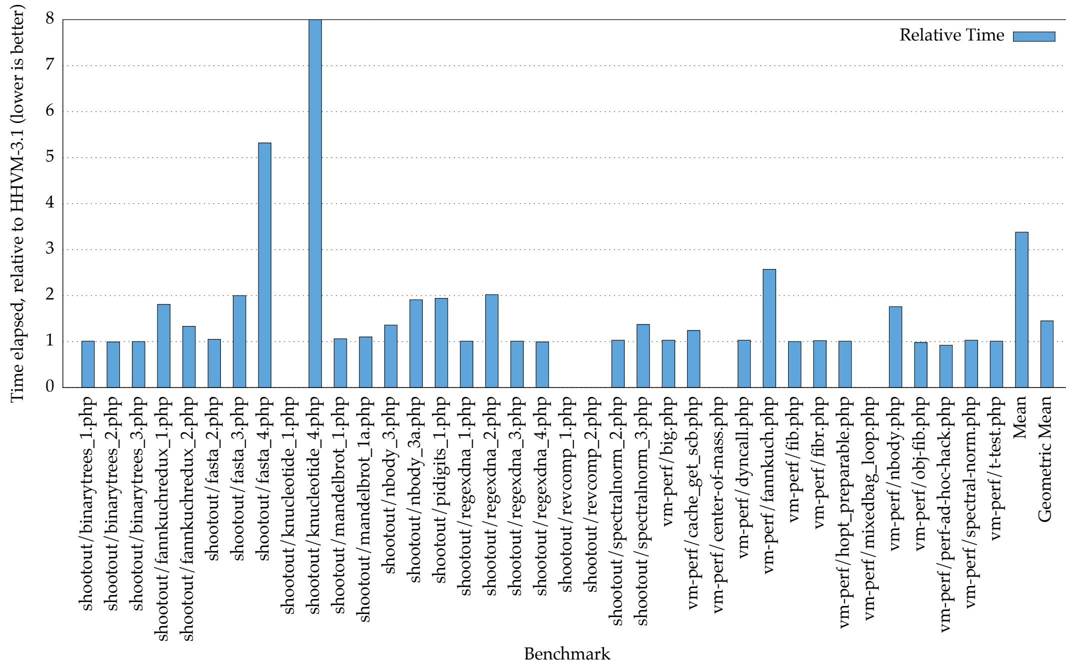
We implement a build of HHVM (`blind-cow`) which uses Blind COW for array copying, but leaves all other reference counting operations in place. This allows us to compare the performance of Blind COW to COW independent of any other changes.

We present the performance of `blind-cow` on the microbenchmark suite and PHP applications in Figure 5.1.

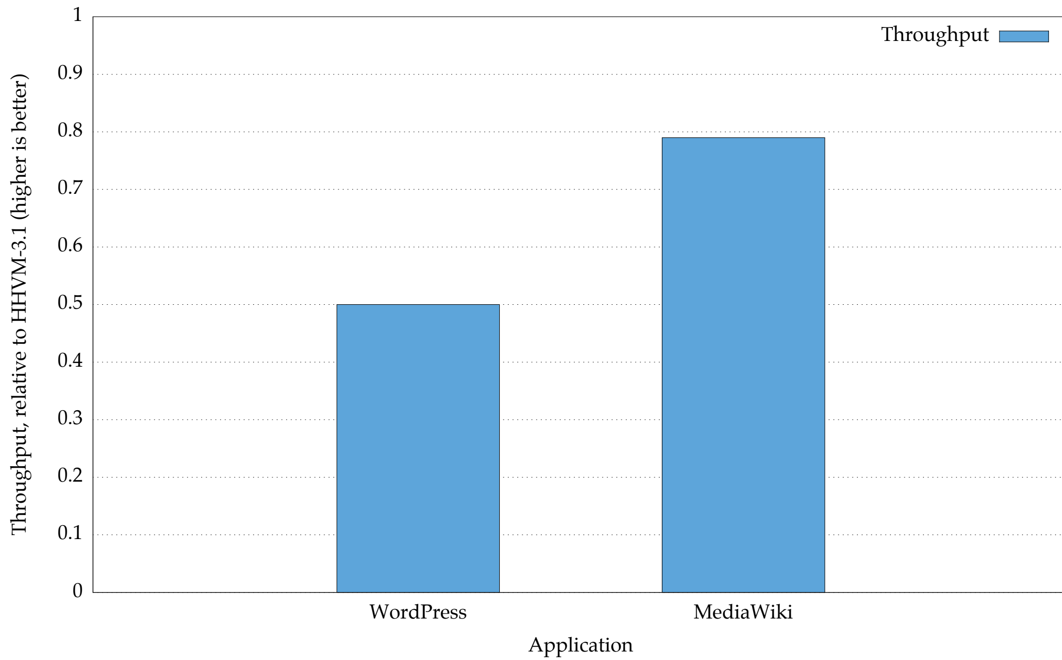
It is clear that performance suffers significantly due to this change. Performance degrades significantly in the pathological case (such as writing to an array within a tight loop), to the point where several benchmarks are unable to complete within the 15 minute time limit. For the real-world applications in Figure 5.1(b), throughout is halved for WordPress, and reduces by 20% for MediaWiki. A likely explanation for the difference in performance between these applications is that, as shown by the results of Section 4.5, WordPress makes heavier use of arrays than MediaWiki.

5.1.3 Discussion and alternative optimisations

This result is clearly unacceptable for production environments. PHP arrays are heavy data structures, and optimising for all available array operations is difficult. Further, the “one size fits all” nature of the data structure means that many PHP programs make heavy use of arrays. However, copy-on-write as it exists currently is reliant on always having exact reference counts available. Therefore, a performant alternative to copy-on-write must be found before replacing reference counting with tracing is viable in a production environment.



(a) Performance of blind-cow on the microbenchmark suite, normalised to HHVM-3.1. Five benchmarks could not complete within the time limit



(b) Throughput of blind-cow for PHP applications, normalised to HHVM-3.1

Figure 5.1: Performance evaluation of blind-cow compared to HHVM-3.1

A simple alternative would be to entirely disable the copy-on-write optimisation, resulting in 'copy-on-assignment'. However, this is unlikely to produce an effective result. As mentioned in Section 2.2.1, the draft PHP specification specifically recommends the use of copy-on-write to provide effective performance. Additionally, Homescu and Şuhan [2011] find that some benchmarks are heavily affected by copy-on-assignment. For example, `shootout/binarytrees-3.php` experiences a 4.18x slowdown using copy-on-assignment compared to copy-on-write. This benchmark makes heavy use of array assignment without writing, and therefore performs well under Blind COW.

A further alternative is to attempt static analysis to eliminate unnecessary copy operations. This copy-on-write problem can be seen as a sequence of assignment, write and copy operations targeting specific array variables. Once an array has been copied and written to once, we can eliminate future copies on that array so long as we can prove that no assignment operations have been performed. This is conceptually similar to the nondeferred reference count operations described in Section 2.1.2.

5.2 Removing reference counting operations

The major drawback of reference counting is that maintaining accurate reference counts incurs a cost at every variable mutation. In this section, we measure the cost of this write barrier to determine whether moving to a tracing collector is likely to have a positive performance impact. At the same time, removing reference counting provides a clean slate for further garbage collection work.

5.2.1 Method

Measuring the performance impact of reference counting independent of other factors requires care, as the garbage collection scheme has far-reaching performance implications for the program as a whole. In order to ensure that we are only measuring the impact of reference count mutations, we produce two builds of HHVM:

- `no-collect`, which performs all refcount operations (including recursive decrementing when an object is due to be collected), but never collects memory. Arrays are copied using Blind COW.
- `no-refcount`, which performs no reference counting operations for types other than references, does not collect objects, and copies arrays using Blind COW.

  Note that a version of HHVM which does not reference count reference objects was considered, however, the correctness of HHVM programs was found to rely too heavily on having these refcounts in place (for example, the PHP5 extension compatibility layer makes heavy use of reference counts [Paroski, 2013]). In the common case, we are able to generate no JIT code for `IncRef` and `DecRef` instructions, and thus the performance penalty of reference counting reference objects is limited.

The only difference between these two builds is that no-refcount does not perform reference counting operations. Both will copy arrays at the same time, and since neither build collects objects, both will have the same memory allocation patterns.

5.2.2 Experimental evaluation

We evaluate the performance of no-refcount relative to no-collect, with results shown in Figure 5.2. Note that many of the microbenchmarks are unable to complete under a 12GB memory limit with no collection, we ignore these benchmarks for the purposes of this test. This may introduce selection bias since the benchmarks with the most memory activity are likely to benefit the most from the removal of reference count mutations.

These results are mixed, with some benchmarks showing increased performance without reference counting, while others show decreased performance. We perform a hypothesis test on the results using a two-tailed t test in order to determine whether the difference between the two builds is statistically significant. The results of this test are shown in Table 5.2 and Table 5.3. The final column of these tables shows the percentage probability that no-refcount and no-collect have the same mean performance, based on the results for that benchmark. We have highlighted the benchmarks that are outside the 95% confidence interval.

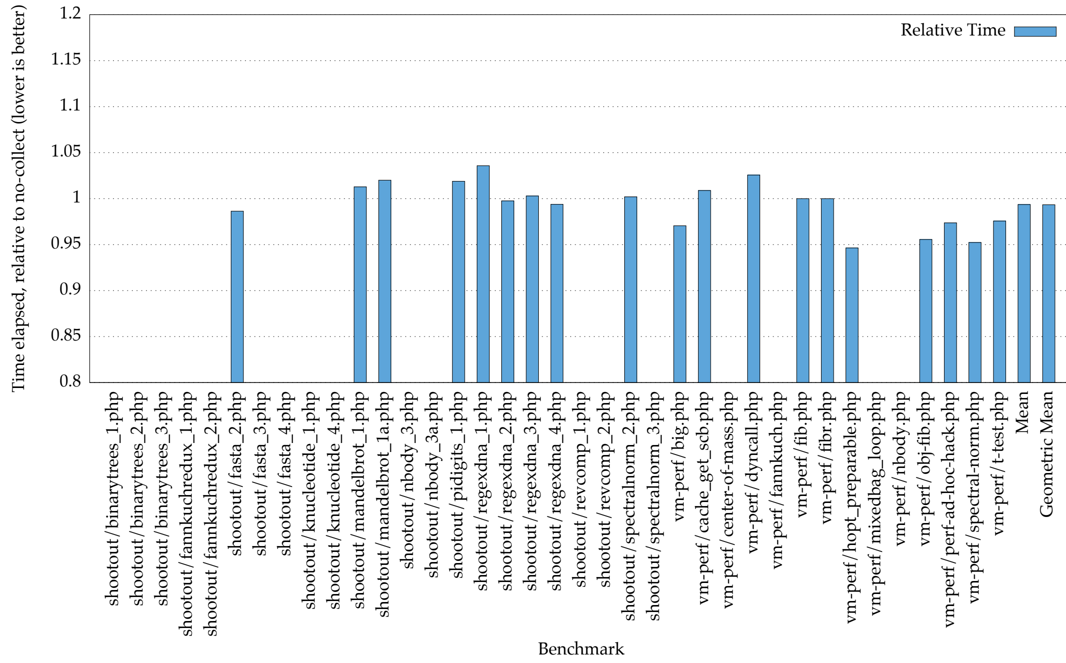
We can see that of the eight benchmarks whose run time decreases for no-refcount, six have a statistically significant difference. On the other hand, of the five benchmarks whose run time increases, only one is statistically significant. For the open source applications, WordPress' result is on the very boundary of the 5% confidence interval, while MediaWiki's result is not statistically significant.

These results show that removing reference counting from HHVM has either no affect, or a 1-5% performance gain. However, measuring this effect accurately requires completely disabling garbage collection, which means that HHVM is operating under adverse memory conditions. The cost of working with a blown-out, fragmented heap may be hiding the actual cost of reference counting.

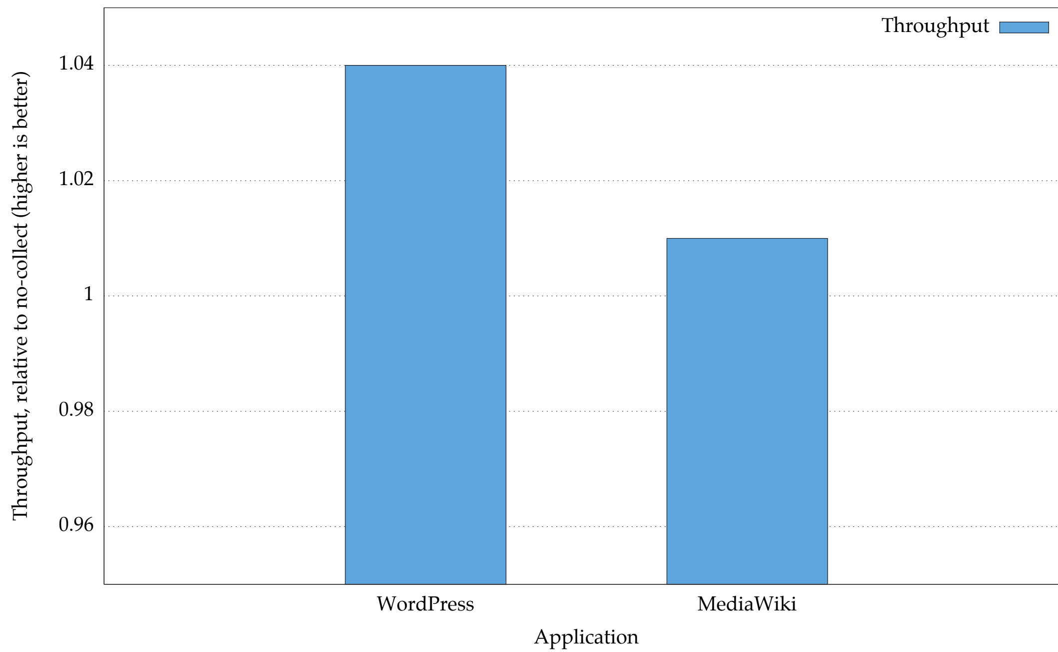
5.3 HHVM without reference counting

We now consider the compatibility of no-refcount with respect to HHVM-3.1 and PHP5. A primary measure of compatibility for HHVM is the test suite, which contains 10,000 test cases which compare output from HHVM to expected output from PHP5. We ran no-refcount through the complete test suite and found the following compatibility issues.

Incompatible extensions A small number of extensions contain C++ functions which are incompatible with blind-cow. Generally, this involves using pointers to array structures in a way that assumes an array will never be copied during a block of code. When Blind COW copies the array, the pointer becomes invalid. These issues are generally simple to solve on a case-by-case basis.



(a) Performance of no-refcount on the microbenchmark suite, normalised to no-collect. Several benchmarks could not complete within the 12GB memory limit



(b) Throughput of no-refcount for PHP applications, normalised to no-collect

Figure 5.2: Performance evaluation of no-refcount compared to no-collect

Benchmark	Relative time	T Test
shootout/fasta_2.php	99%	**0.4%**
shootout/mandelbrot_1.php	101%	4.3%
shootout/mandelbrot_1a.php	102%	25.7%
shootout/pidigits_1.php	102%	**0.0%**
shootout/regexdna_1.php	104%	63.4%
shootout/regexdna_2.php	100%	46.2%
shootout/regexdna_3.php	100%	39.8%
shootout/regexdna_4.php	99%	54.9%
shootout/spectralnorm_2.php	100%	41.0%
vm-perf/big.php	97%	**0.4%**
vm-perf/cache_get_scb.php	101%	10.8%
vm-perf/dyncall.php	103%	10.1%
vm-perf/fib.php	100%	100.0%
vm-perf/fibr.php	100%	100.0%
vm-perf/hopt_preparable.php	95%	**0.1%**
vm-perf/obj-fib.php	96%	**0.2%**
vm-perf/perf-ad-hoc-hack.php	97%	100.0%
vm-perf/spectral-norm.php	95%	**0.1%**
vm-perf/t-test.php	98%	**0.2%**

Table 5.2: Confidence in the no reference counting microbenchmark results of Figure 5.2(a)

Benchmark	Relative throughput	T Test
WordPress	104%	**2.6%**
MediaWiki	101%	73.6%

Table 5.3: T-test confidence in the no reference counting open source application results of Figure 5.2(b)

Precise destruction no-refcount disables precise destruction of objects (Specifically, destructors are called for every object at the very end of the request). While this rarely affects the correctness of programs, many test cases rely on destructors being called in a particular order to correctly match the expected output.

Incorrect object IDs PHP labels each object with a integer ID, which is only used for serialisation (including debug printing). In some cases, PHP5 will reuse an object ID after the original object was collected. As we do not have precise destruction, no-refcount is unable to reuse IDs in the same way. This is purely a cosmetic difference, and has no impact on correctness.

Failure to demote references In Section 2.2.2, we discuss how PHP demotes reference objects with a refcount of 1 back to values. However, the lack of precise destruction means that reference objects may have incorrect reference counts, and therefore will not be demoted when they should be. A tracing collector could reduce the impact of this change by performing demotion of eligible arrays while tracing the heap. This change *could* affect the correctness of programs, but does not affect our benchmark suite, MediaWiki or WordPress.

Summary These test results show that there are incompatibility issues with the no-refcount version of HHVM. However, we have still been able to successfully run a variety of benchmarks and open source PHP applications using this build. Futhermore, we are confident that any incompatibility issues which are encountered with this build could be solved with only a small amount of developer effort, either by modifying HHVM extensions or the PHP application itself. Therefore, we believe these compatibility issues should not block further development of garbage collection algorithms for PHP.

5.4 Summary

This chapter performs key work to eliminate reference counting from HHVM and open the door for tracing reference counting in HHVM. Although our Blind COW optimisation was found to have insufficient performance, we suggest alternative methods which may produce better results. We remove reference counting operations from HHVM in such a way that PHP's reference semantics are preserved, while also improving performance for some benchmarks. The result is a version of HHVM with no reference counting and only minor incompatibilities with PHP5. In the next chapter, we use this version of HHVM as the starting point for the design of a mark-region garbage collector for HHVM.

Chapter 6

Design of a Tracing Garbage Collector for HHVM

In Chapter 5, we describe and evaluate `no-refcount`, a build of HHVM with no reference counting, which despite performance and minor compatibility issues, provides a clean slate for further garbage collection to be performed. In this chapter, we describe the design of a proof-of-concept tracing garbage collector for HHVM. Specifically, we have chosen to design a mark-region collector (see Section 2.1.3 for details), as it is relatively simple to implement, but forms a base for the future implementation of an Immix collector.

In Section 6.1 we describe the changes to object allocation required for the implementation of mark-region. Then, Section 6.2 describes the method used to trace the heap and recycle unused memory. Finally, we conclude in Section 6.3 by describing the remaining work required to successfully implement this collector.

6.1 Block-based allocation

6.1.1 Memory allocation in HHVM

HHVM currently uses size-segregated free-lists for allocation. When allocating an object, the free-list of the appropriate size is checked. If no memory is available in this free-list, a bump-pointer is used to contiguously allocate from 2 MB 'slabs' of memory. When an object is freed, the memory is appended to the appropriate free-list. Therefore, free-lists are used in the common case, with bump-pointer allocation used initially, and while the heap is growing. Objects over 2 kB in size are not handled using this system, but instead are directly allocated and freed using `malloc` and `free`. According to the results of Section 4.3, objects which fall into this category are uncommon, making up 0.1–0.3% of objects allocated by WordPress and MediaWiki.

HHVM version 3.2 includes improvements to this allocator. Firstly, size-classes are logarithmically spaced (compared to linear in HHVM-3.1), which reduces the number of size classes required while limiting internal fragmentation. Secondly, a small amount of preallocation is performed to limit the number of times the bump

allocator is called. This also makes it more likely that objects of the same size will be located next to each other in memory, potentially improving locality. Our experiments are based on HHVM version 3.1, so these improvements have not been evaluated.

6.1.2 blockMalloc

HHVM uses the same 'smart allocator' for both reference-counted PHP objects and a number of internal VM objects. This is an effective strategy when reference counting, but is not suited to performing tracing collection as VM objects will appear as empty holes in the heap to the collector.

Therefore, we propose a new allocation method, `blockMalloc`, to be added *alongside* the existing smart allocator. `blockMalloc` and the existing smart allocator must allocate from separate areas of the heap, therefore we mark each 'slab' of memory requested from the operating system with a boolean to indicate whether it is 'GC enabled'. Upon allocation, GC enabled slabs are split into 8 kB blocks, stored in a queue. This block size is chosen to limit block-level fragmentation to 25% [Blackburn and McKinley, 2008]. Allocation of individual objects involves simply bump-allocating into the currently active block. If this allocation would cross the end of the block, we pop the next block off the queue and allocate into it instead. Finally, if the queue is empty we request a new slab of memory and use it to refill the queue.

We continue to use the existing HHVM strategy of allocating large objects separately using `malloc`. In order to perform garbage collection for these objects, we must record the address of these objects in a set or similar data structure. The number of large objects is small enough that this should not have a significant performance penalty, however, optimisations aimed at large objects should be explored in the future.

6.1.3 PHP extensions

HHVM and PHP5 extensions are a further case which needs to be considered. Extensions may take the form of either PHP or C++ code, and have a large amount of control over the heap. A particular issue is that extensions may allocate PHP objects which are never actually visible from PHP code. Our proposed solution is to use the `blockMalloc` allocator for extensions, so that by default PHP objects created by an extension will be garbage collected. In addition, we provide a method which extensions can use to 'pin' an object so that it is never collected. Specifically, we flip a pin bit on the containing block so that it is marked as active regardless of whether any live objects are seen in it during tracing.

It is also possible to implement a simple debugging tool to allow diagnosis of extensions which are using memory incorrectly. When a block is collected, the debugging tool fills it with a magic number instead of returning it to the allocator. Any extensions which have kept pointers to memory in that block will error the next time they attempt to access it. The root cause of this problem can then be tracked down using `gdb`.

6.1.4 Experimental evaluation

We have implemented this allocation scheme in a build of HHVM (`block-malloc`). While `block-malloc` is compatible with `no-refcount` (and is intended to be used with `no-refcount` in the future), we implement `block-malloc` directly on top of `HHVM-3.1` for evaluation purposes. Note that we have not yet fully implemented the handling of large objects described above – large objects are allocated using `malloc`, but are not logged. Figure 6.1 shows the results of comparing our current version of `block-malloc` to `HHVM-3.1` on the microbenchmarks and open source applications.

This build does not behave in a sensible manner, since it mixes a new allocator with an old garbage collection scheme. As a result, `block-malloc` allocates new memory in blocks, which are then collected by the existing reference counting collector and returned to the free-list allocator. `block-malloc` is therefore unable to reuse memory and will allocate more memory than `HHVM-3.1`. This is reflected in the results of the microbenchmarks (Figure 6.1(a)). Most of the benchmarks have identical performance between both builds of HHVM. However, some of the benchmarks have unfavourable memory usage patterns for this situation. For example, the memory usage of `shootout/binarytrees_3.php` increases from 200 MB to 8400 MB. Likewise, the peak memory usage of the open source applications increases, with MediaWiki increasing from 530 MB to 1700 MB and WordPress increasing from 410 MB to 760 MB.[1]

This problem will no longer occur when a new garbage collector is implemented on top of `block-malloc`. Therefore, the important takeaway from this performance evaluation is that in the case where memory usage does not disrupt results, `block-malloc` does not significantly change the performance of HHVM.

6.2 Collection and recycling

Collection for a mark-region garbage collector is straightforward, and does not differ greatly from simple tracing as described by [McCarthy, 1960]. However, there are some minor HHVM-specific considerations which must be made.

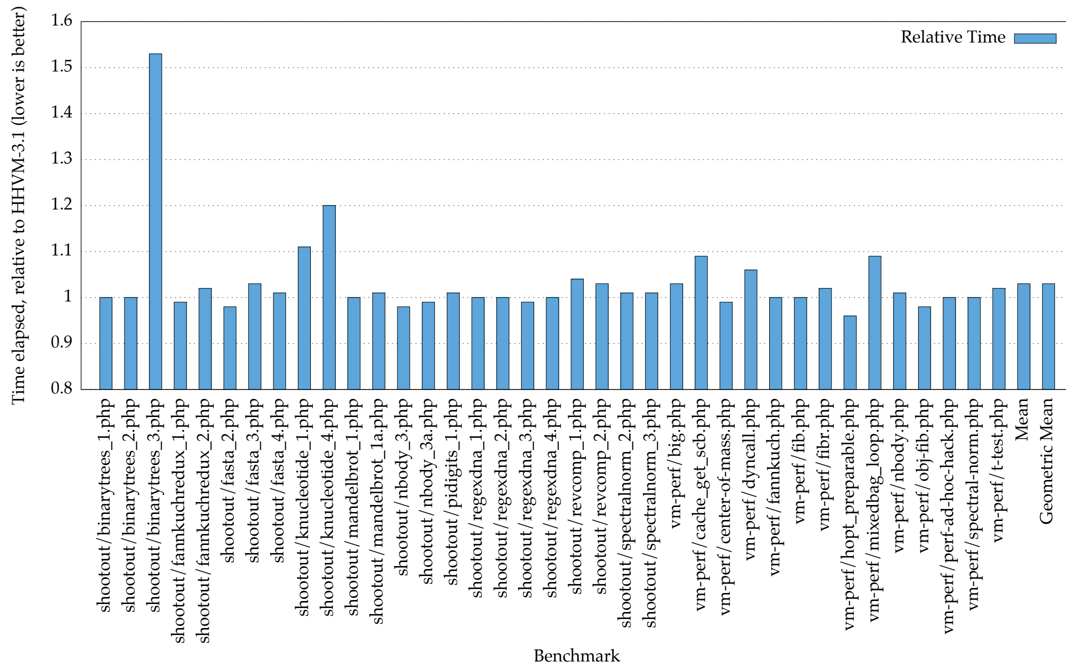
6.2.1 Collection algorithm

The basic tracing and collection algorithm is as follows:

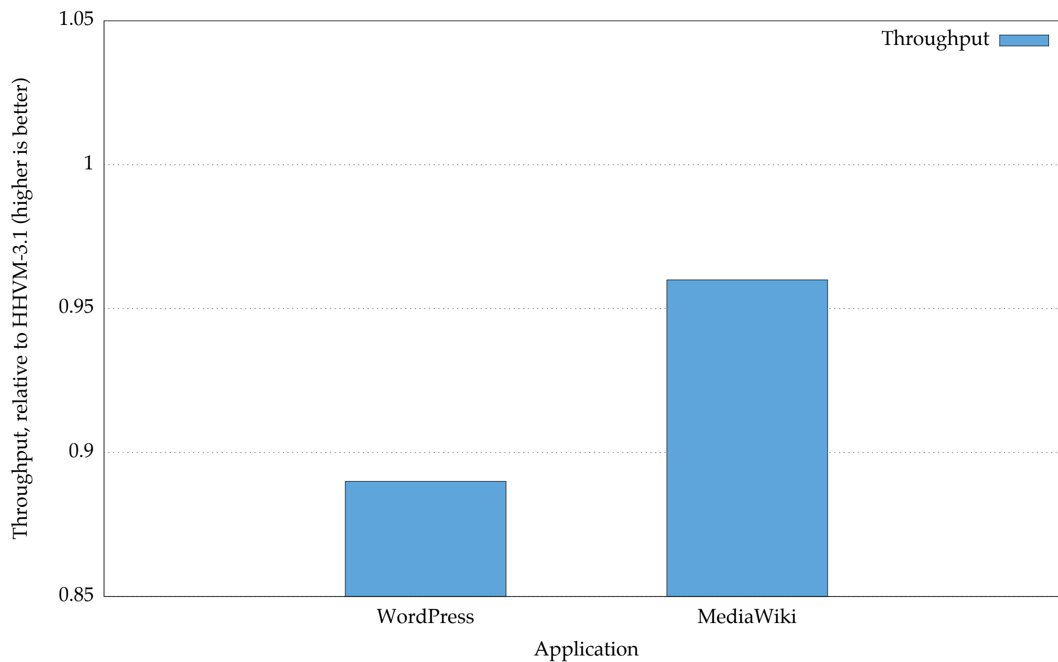
1. During execution of the program, log the allocation of any object which has a destructor which must be called before it is collected. This includes PHP `Object`s whose class has a `__destruct` method, and any PHP `Resource`s.

2. When garbage collection is triggered, the garbage collection module requests a list of all active blocks from the allocator. It creates a bitmap for the liveness

[1] These results indicate that MediaWiki has a greater relative increase in heap usage than WordPress, while WordPress suffers greater performance impact in Figure 6.1(a). However, we test MediaWiki with more concurrent connections than WordPress. WordPress' heap usage per connection has increased by more than MediaWiki's, explaining this discrepancy.



(a) Time taken by block-malloc on the microbenchmark suite, relative to HHVM-3.1



(b) Throughput of block-malloc for WordPress and MediaWiki, relative to HHVM-3.1

Figure 6.1: Performance evaluation of block-malloc compared to HHVM-3.1

of each block and the liveness of each object (512 bits for each block, recalling that objects are aligned to 16-byte boundaries and each block is 8 kB), including large objects. Object-level liveness could also be recorded using a mark bit in object headers, we use a bitmap to keep the design simple.

3. The garbage collection module then finds the set of roots of the virtual machine. This primarily involves stepping through the execution stack and the global variable table.

4. Starting at the roots, perform a search through the heap. At each object, mark the block and the individual object as live. Array, object and reference objects have children which can be searched through.

5. Any blocks and large objects which were not seen during tracing can be collected. Before these blocks can be collected, we must search the set of destructible objects and resources for any objects whose destructors need to be run. Having completed this, the blocks can be returned to the allocator.

6.2.2 Triggering garbage collection

As mentioned in Section 2.1.5, implementing exact tracing collection requires exact stack maps, whose implementation is a formidable engineering challenge. Like many similar virtual machines, HHVM does not implement stack maps, and therefore we are unable to perform garbage collection while executing JIT code. However, as described in Section 2.3, HHVM performs JIT compilation on small 'tracelets' of code. As a result, while we may not be able to perform garbage collection at exactly the time that we want to, we will rarely have to wait long before the end of a JIT tracelet allows us to perform GC.

Further, HHVM does not strictly enforce memory usage limits. It uses a mechanism named 'surprise flags' to enforce memory limits for each request. When the memory limit is reached, a surprise flag is set and the program continues executing. It is not until the next time that a method is entered that the surprise flags are checked and the program is gracefully shut down.

The surprise flags provide a convenient method for requesting and triggering garbage collection, with no additional overhead created by adding a new flag. As a starting point, we simply request garbage collection when we run out of available blocks and allocate a new slab of memory. Triggering garbage collection in this way is clearly suboptimal, as garbage collection can only occur *after* new memory has been allocated. There are a number of options available to improve this, including pre-emptively performing garbage collection when the pool of available blocks is low, and implementing conservative garbage collection (see Section 2.1.5) to allow for garbage collection to be performed at any point.

6.3 Summary

In this chapter, we have described the design of a proof-of-concept tracing collector for HHVM. This design optimises for simplicity of implementation rather than for performance. Therefore, when faced with an implementation choice, we select the option that is easiest to implement. Many of the individual components of this collector have been written, including a new allocator and a basic heap tracing module, but integration has not been completed. Once a basic implementation is complete, we will be able to iterate on this basic design by reviewing these design choices, making informed decisions backed by experimental data.

Chapter 7

Conclusion

Despite the widespread use of the PHP programming language to power webpages, little effort has been made to optimise PHP's use of memory. The standard PHP implementation, PHP5, uses naive reference counting to perform garbage collection. This algorithm was introduced in 1960 and is widely known to be inefficient. However, the language semantics of PHP, including the pass-by-value semantics for arrays create loose ties between PHP and reference counting garbage collection, and make moving away from this algorithm difficult.

The advent of HHVM, a new high performance virtual machine for PHP, provides an opportunity for a solution to this problem to be found. Making more efficient use of memory is a high priority for the HHVM team at Facebook, whose scale means that small improvements to memory usage, response time and memory bandwidth utilisation have a significant impact.

We perform an in-depth analysis of the current memory usage characteristics of HHVM applications, and find that HHVM has similar demographics to PHP5 and Java, with large numbers of small, short-lived objects. In this case, reference counting garbage collection is effective in keeping the heap small, since memory is continually being recycled. However, this is a tradeoff, with additional work being performed in the common case where an object has a short lifetime. Therefore, we conclude that HHVM is likely to be well suited to garbage collection algorithms such as Immix, which perform well in this situation.

We remove reference counting from HHVM, thereby creating a clean slate for further garbage collection work. Performing this work comes with a cost, however, as 'Blind Copy-on-Write', our proposed alternative for the array copy-on-write optimisation, was found to be inefficient and unsuitable for a production environment. Despite this issue, this version of HHVM is still useful for development of garbage collectors, and is compatible with the PHP benchmarks and applications we have tested.

Finally, we describe the design of a proof-of-concept tracing collector for HHVM. We identify several issues which must be considered when implementing such a collector, and provide potential solutions for these issues. While the implementation of this collector has not yet been completed, we believe that with further development,

it will be competitive with the existing naive reference counting collector used by HHVM. From there, a wide range of optimisations can be implemented to further improve HHVM's memory management.

7.1 Future work

Garbage collection is a large field, and therefore there are many avenues for future work related to this project. We identify four such opportunities below.

Complete implementation of proof-of-concept garbage collector Many of the component parts of the tracing garbage collector described in Chapter 6 have been implemented, but the complete collector was left unfinished due to time constraints. Completing this collector would demonstrate that tracing garbage collection is a feasible approach to memory management in PHP.

Improving copy-on-write In Section 5.1.2 we identify that the proposed Blind COW optimisation provides an unacceptable performance loss. Blind COW *always* copies arrays before write operations, even when it is clearly unnecessary to do so. Static analysis may be able to identify and eliminate unnecessary write operations, based on the sequence of read, write and assignment operations to the array. Unoptimised copy-on-assignment may also be a competitive alternative to Blind COW.

Implement Immix The mark-region collector described in Chapter 6 was selected as it is a good basis for an Immix collector. Implementing such a collector is a major step in implementing high performance garbage collection for HHVM.

Implement conservative garbage collection Currently, the times when we are able to perform garbage collection are limited by the absense of exact stack maps in HHVM. Implementing conservative garbage collection would allow us to perform garbage collection at more appropriate times, while having little overall performance impact.

Appendix A

HHVM Patches

Throughout the text of this thesis, we refer to different versions of HHVM using sans-serif text. In this chapter, we give a brief overview of how to download and apply these patches, along with an index of all of the patches mentioned.

Two of the patches, HHVM-3.1-update and mark-region-collector, contain work ported from the main development branch of HHVM. These commits are clearly labelled with the name and email address of the original author.

A.1 Obtaining patches

A tarball of all patch files is available from http://www.timsergeant.com/files/hhvm/thesis-patches.tar.gz. Alternatively, patches are available from my fork of the official HHVM Github repository at https://www.github.com/tgsergeant/hhvm. Each patch is available as a branch of that git repository.

A.2 Applying patches

The patch files are modular, and only contain the changes necessary for that specific feature/version. Therefore, the order in which patches can be applied is somewhat flexible, and allows for some experimentation with mixing-and-matching different features. Discovering the extent to which this is possible due to dependencies and conflicts between the patch files is left as an exercise for the reader – instead, the recommended application order is shown in Figure A.1 below.

For example, to reach block-malloc, run the following git commands:

```
git clone git@github.com:facebook/hhvm.git
cd hhvm
git checkout HHVM-3.1
git am patches/hhvm-3.1-update.patch
git am patches/blind-cow.patch
git am patches/no-refcount.patch
git am patches/block-malloc.patch
```

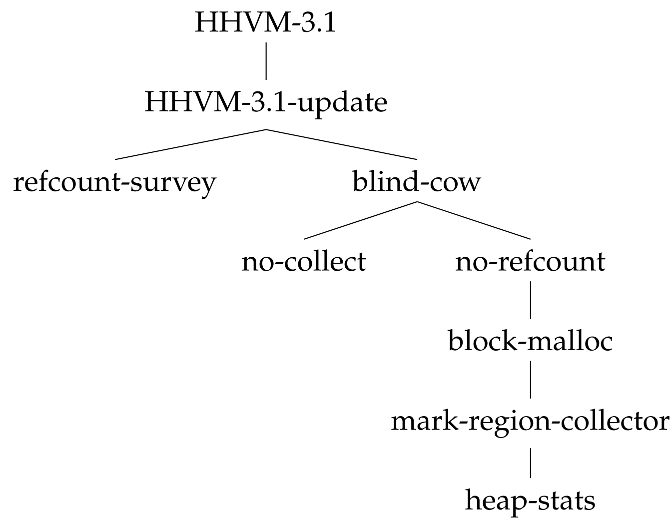


Figure A.1: Recommended application order for patches. Starting at the root of the tree, apply each of the patches in order until the desired version of HHVM is reached

A.3 Index of available patches

HHVM-3.1-update A collection of backported bugfixes and config changes encountered during the development process. May not be necessary on some systems.

refcount-survey Instrumented build of HHVM which collects statistics about reference count operations. To run, compile in debug mode and set the environment variable `TRACE=rcsurvey:3`. The report will be written to `/tmp/hphp.log`.

blind-cow Described in section 5.1.1, removes need for reference counts in copy-on-write.

no-collect Performs all reference counting operations, but never actually collects any memory.

no-refcount Disables reference counting operations for all types other than `RefData`. As a result, no memory is collected.

block-malloc Replaces the default free-list allocator with a bump-pointer allocator. The new allocator is only used for reference counted objects. HHVM still allocates some internal objects and resources using free-lists, these are located in separate slabs of memory so that they do not interfere with garbage collection.

mark-region-collector Implements a module for tracing the heap, and a module which uses this to perform mark-region garbage collection. This is currently incomplete.

heap-stats Traces the heap shortly after each new slab of memory is allocated and measures the number and size of objects of different types. Enable the report with `TRACE=heapstats:3`. This patch depends upon mark-region-collector.

Bibliography

ADAMS, K.; EVANS, J.; MAHER, B.; OTTONI, G.; PAROSKI, A.; SIMMERS, B.; SMITH, E.; AND YAMAUCHI, O., 2014. The HipHop Virtual Machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14 (Portland, Oregon, USA, 2014), 777–790. ACM, New York, NY, USA. doi:10.1145/2660193.2660199. (cited on pages 15 and 19)

BACON, D. AND RAJAN, V., 2001. Concurrent cycle collection in reference counted systems. In *ECOOP 2001 – Object-Oriented Programming* (Ed. J. KNUDSEN), vol. 2072 of *Lecture Notes in Computer Science*, 207–235. Springer Berlin Heidelberg. ISBN 978-3-540-42206-8. doi:10.1007/3-540-45337-7_12. (cited on page 6)

BARTLETT, J. F., 1988. Compacting garbage collection with ambiguous roots. *SIGPLAN Lisp Pointers*, 1, 6 (Apr. 1988), 3–12. doi:10.1145/1317224.1317225. (cited on page 9)

BENDA, J.; MATOUSEK, T.; AND PROSEK, L., 2006. Phalanger: Compiling and running PHP applications on the Microsoft .NET platform. *.NET Technologies 2006*, (2006). (cited on page 16)

BETTERWP.NET, 2011. Dummy data for WordPress. http://betterwp.net/wordpress-dummy-data/. (cited on page 19)

BIERE, A. AND JUSSILA, T., 2011. Runlim. http://fmv.jku.at/runlim/. (cited on page 20)

BIGGAR, P. AND GREGG, D., 2009. Compiling and optimizing scripting languages. http://www.youtube.com/watch?v=kKySEUrP7LA. (cited on page 16)

BLACKBURN, S. M.; CHENG, P.; AND MCKINLEY, K. S., 2004. Myths and realities: The performance impact of garbage collection. In *SIGMETRICS – Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems, New York, NY, USA, June 12–16, 2004*. ACM. doi:10.1145/1005686.1005693. (cited on page 8)

BLACKBURN, S. M.; GARNER, R.; HOFFMANN, C.; KHANG, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A.; JUMP, M.; LEE, H.; ELIOT, J.; MOSS, B.; PHANSALKAR, A.; STEFANOVIC, D.; VANDRUNEN, T.; VON DINCKLAGE, D.; AND WIEDERMANN, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (Portland, Oregon, USA, 2006), 169–190.

ACM Press, New York, NY, USA. doi:10.1145/1167473.1167488. (cited on pages 26 and 31)

Blackburn, S. M. and McKinley, K. S., 2003. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2003), Anaheim, CA, USA, October 26-30, 2003*, vol. 38 of *SIGPLAN Notices*. ACM. doi:10.1145/949305.949336. (cited on page 9)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA, Jun. 2008). ACM. doi:10.1145/1375581.1375586. (cited on pages 8 and 42)

Cheney, C. J., 1970. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13, 11 (Nov. 1970), 677–678. doi:10.1145/362790.362798. (cited on page 8)

Collins, G. E., 1960. A method for overlapping and erasure of lists. *Communications of the ACM*, 3, 12 (Dec. 1960), 655–657. doi:10.1145/367487.367501. (cited on pages 5 and 7)

de Vries, E.; Gilbert, J.; and Biggar, P., 2011. phc – the open source PHP compiler. http://phpcompiler.org/. (cited on page 15)

Deutsch, L. P. and Bobrow, D. G., 1976. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19, 9 (Sep. 1976), 522–526. doi:10.1145/360336.360345. (cited on page 6)

Facebook, 2014. HHVM open source tests. http://www.hhvm.com/frameworks/. (cited on page 14)

Fullmer, J., 2012. Siege manual. http://www.joedog.org/siege-manual/. (cited on page 20)

Homescu, A. and Şuhan, A., 2011. HappyJIT: A tracing JIT compiler for PHP. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11 (Portland, Oregon, USA, 2011), 25–36. ACM, New York, NY, USA. doi:10.1145/2047849.2047854. (cited on pages 16 and 36)

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 *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC 2011), San Jose, CA, June 5, 2011*. ACM. doi:10.1145/1988915.1988930. (cited on pages 21, 26, and 31)

Joisha, P. G., 2006. Compiler optimizations for nondeferred reference counting garbage collection. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06 (Ottawa, Ontario, Canada, 2006), 150–161. ACM, New York, NY, USA. doi:10.1145/1133956.1133976. (cited on page 7)

Joisha, P. G., 2007. Overlooking roots: A framework for making nondeferred reference-counting garbage collection fast. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07 (Montreal, Quebec, Canada, 2007), 141–158. ACM, New York, NY, USA. doi:10.1145/1296907.1296926. (cited on page 7)

Joisha, P. G., 2008. A principled approach to nondeferred reference-counting garbage collection. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08 (Seattle, WA, USA, 2008), 131–140. ACM, New York, NY, USA. doi:10.1145/1346256.1346275. (cited on page 7)

Levanoni, Y. and Petrank, E., 2001. An on-the-fly reference counting garbage collector for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01 (Tampa Bay, FL, USA, 2001), 367–380. ACM, New York, NY, USA. doi:10.1145/504282.504309. (cited on page 7)

Levanoni, Y. and Petrank, E., 2006. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.*, 28, 1 (Jan. 2006), 1–69. doi: 10.1145/1111596.1111597. (cited on page 7)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi: 10.1145/358141.358147. (cited on page 8)

Marcey, J., 2014. Announcing a specification for PHP. http://hhvm.com/blog/5723/announcing-a-specification-for-php. (cited on page 9)

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3, 4 (Apr. 1960), 184–195. doi:10.1145/367177.367199. (cited on pages 7 and 43)

Paroski, D., 2013. PHP extension compatibility layer. https://github.com/facebook/hhvm/blob/5c72501a1aae11274cf9b0c696028e88f3111751/hphp/doc/php.extension.compat.layer. (cited on page 36)

Q-Success, 2014. Usage statistics and market share of PHP for websites. http://w3techs.com/technologies/details/pl-php/all/all. (cited on page 1)

Roadsend, I., 2014. Roadsend PHP. http://roadsend.com/. (cited on page 16)

Shahriyar, R.; Blackburn, S. M.; and Frampton, D., 2012. Down for the count? Getting reference counting back in the ring. In *Proceedings of the Eleventh ACM SIGPLAN International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16* (Beijing, China, Jun. 2012). doi:10.1145/2258996.2259008. (cited on pages 5, 27, and 31)

Shahriyar, R.; Blackburn, S. M.; and McKinley, K. S., 2014. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14 (Portland, Oregon, USA, 2014), 121–139. ACM, New York, NY, USA. doi:10.1145/2660193.2660198. (cited on pages 6 and 9)

Shahriyar, R.; Blackburn, S. M.; Yang, X.; and McKinley, K. M., 2013. Taking off the gloves with reference counting immix. In *OOPSLA '13: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (Indianapolis, IN, USA, Oct. 2013). doi:10.1145/2509136.2509527. (cited on page 7)

Simmers, B., 2014. HHVM JIT optimization passes. https://github.com/facebook/hhvm/blob/2c22fff6dbb65b30f93c764a62369fb96892f437/hphp/doc/hackers-guide/jit-optimizations.md. (cited on page 31)

Tarjan, P., 2014. HHVM 3.1.0. http://hhvm.com/blog/5195/hhvm-3-1-0. (cited on page 19)

Tatsubori, M.; Tozawa, A.; Suzumura, T.; Trent, S.; and Onodera, T., 2010. Evaluation of a just-in-time compiler retrofitted for PHP. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10 (Pittsburgh, Pennsylvania, USA, 2010), 121–132. ACM, New York, NY, USA. doi:10.1145/1735997.1736015. (cited on page 16)

The PHP Group, 2002. Doc bug #20993: Element value changes without asking. https://bugs.php.net/bug.php?id=20993. (cited on page 10)

The PHP Group, 2014. The PHP language specification. http://git.php.net/?p=php-langspec.git. (cited on pages 9, 11, and 14)

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 (Savannah, GA, USA, 2009), 200–212. ACM, New York, NY, USA. doi:10.1145/1480881.1480908. (cited on page 10)

Ungar, D., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, 157–167. ACM, New York, NY, USA. doi:10.1145/800020.808261. (cited on page 8)

Venstermans, K.; Eeckhout, L.; and De Bosschere, K., 2006. 64-bit versus 32-bit virtual machines for Java. *Software: Practice and Experience*, 36, 1 (2006), 1–26. (cited on page 26)

Wikimedia Foundation, 2014. HHVM – MediaWiki. http://www.mediawiki.org/wiki/HHVM. (cited on page 3)

WordPress.com, 2014. Traffic – WordPress.com. http://en.wordpress.com/stats/traffic/. (cited on page 3)

Zhao, H.; Proctor, I.; Yang, M.; Qi, X.; Williams, M.; Gao, Q.; Ottoni, G.; Paroski, A.; MacVicar, S.; Evans, J.; and Tu, S., 2012. The HipHop compiler for PHP. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12 (Tucson, Arizona, USA, 2012), 575–586. ACM, New York, NY, USA. doi:10.1145/2384616.2384658. (cited on page 15)