Deferred Gratification: Engineering for High Performance Garbage Collection from the Get Go *

Ivan Jibaja† Stephen M. Blackburn‡ Mohammad R. Haghighat* Kathryn S. McKinley†

†The University of Texas at Austin ‡Australian National University *Intel Corporation

ivan@cs.utexas.edu, steve.blackburn@anu.edu.au, mohammad.r.haghighat@intel.com, mckinley@cs.utexas.edu

Abstract

Implementing a new programming language system is a daunting task. A common trap is to punt on the design and engineering of exact garbage collection and instead opt for reference counting or conservative garbage collection (GC). For example, AppleScript™, Perl, Python, and PHP implementers chose reference counting (RC) and Ruby chose conservative GC. Although easier to get working, reference counting has terrible performance and conservative GC is inflexible and performs poorly when allocation rates are high. However, high performance GC is central to performance for managed languages and only becoming more critical due to relatively lower memory bandwidth and higher memory latency of modern architectures. Unfortunately, retrofitting support for high performance collectors later is a formidable software engineering task due to their exact nature. Whether they realize it or not, implementers have three routes: (1) forge ahead with reference counting or conservative GC, and worry about the consequences later; (2) build the language on top of an existing managed runtime with exact GC, and tune the GC to scripting language workloads; or (3) engineer exact GC from the ground up and enjoy the correctness and performance benefits sooner rather than later.

We explore this conundrum using PHP, the most popular server side scripting language. PHP implements reference counting, mirroring scripting languages before it. Because reference counting is incomplete, the implementors must (a) also implement tracing to detect cyclic garbage, or (b) prohibit cyclic data structures, or (c) never reclaim cyclic garbage. PHP chose (a), AppleScript chose (b), and Perl chose (c). We characterize the memory behavior of five typical PHP programs to determine whether their implementation choice was a good one in light of the growing demand for high performance PHP. The memory behavior of these PHP programs is similar to other managed languages, such as Java™ —they allocate many short lived objects, a large variety of object sizes, and the average allocated object size is small. These characteristics suggest copying generational GC will attain high performance.

Language implementers who are serious about correctness and performance need to understand deferred gratification: paying the software engineering cost of exact GC up front will deliver correctness and memory system performance later.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); Optimization

General Terms Design, Experimentation, Performance, Management, Measurement, Languages

Keywords Heap, PHP, Scripting Languages, Garbage Collection

1. Introduction

Programmers are increasingly choosing managed languages, which provide a high level of abstraction with automatic memory management, safe pointer disciplines, typing disciplines, and sometimes object-orientation. For example, PHP and JavaScript™ are the languages of choice for server and client-side web programming. Scripting languages provide some of the highest levels of abstraction and concision, which greatly improve programmer productivity. While scripting languages typically emerge to assist with specific tasks, over time programmers use them to perform more complex and general tasks. For example, programmers are now using some of today's most popular scripting languages, such as Python, Perl, JavaScript, Ruby, and PHP, for a large variety of tasks—long and short, simple and complex. As the use of managed scripting languages grows, so do the demands of their applications and consequently demand for correctness and high performance.

Implementing a new scripting language is a daunting task and is often initiated without anticipating that the language may explode in popularity. Consequently, the complexity and sophistication of the applications will grow and demand much from the implementation. Implementors choose to build from scratch or to target an existing Virtual Machine (VM) language, such as Java or .NET. For example, JVM languages now include JRuby, Jython, JTcl, as well as JavaScript variants. These implementations enjoy the high performance optimizing compilers and garbage collector of the host JVM, since JVM technology is now mature. However, mismatches between language features can introduce systemic inefficiencies and restrict the programs in the guest language. Furthermore, as far as we are aware, none of the existing VMs have tuned their collectors for these scripting languages. Our results suggest that this is a significant opportunity.

Memory management is central to the performance of scripting languages. JRuby core member, Charles Nutter states that 'you simply can't have a high-performance Ruby without a high-performance GC' [17]. We can expect the impact of memory management on performance to grow in the future because memory bandwidth on future chip multiprocessors (CMPs) is struggling to keep pace with the demands from large numbers of processes with

* This work is supported by NSF CSR-0917191, NSF SHF-0910818, NSF CCF-1018271, Intel, and Google. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

MSPC'11, June 5, 2011, San Jose, California, USA.

multiple threads of executions [11, 15, 21, 26]. Furthermore, Ruby programs have a very high mortality rate of young objects, so the language benefits from copying and generational collection. Copying bestows good locality and offers cheap en mass reclamation while generational collection focuses effort on the easy-to-collect, high mortality young objects [7, 8]. However, copying collection requires exact GC and generational collection requires support for barriers. Thus high performance for such languages is predicated on strong support for GC from the runtime.

Unfortunately support for copying and generational collection requires pervasive predicates throughout the VM implementation. The virtual machine must be able to: (1) precisely enumerate all live pointers into the heap, (2) enumerate all pointers within each live heap object, and (3) intercept with a write barrier every change to a pointer field of a heap object. In principle, meeting these requirements is not difficult, but in practice these requirements have a significant impact on the JVM design. Requirement (1) is the most pervasive—the VM must meticulously maintain information that identifies all pointers into the heap: on the stack, in registers, and within native code. These invariants about pointers and offsets require a rigorous software engineering discipline in the runtime (e.g. in the compiler). Because they are systemic, the predicates must be understood by virtually all implementers of the VM, even though most are not directly involved in developing the memory manager. Mistakes almost always manifest as hard to diagnose application crashes. It is easy to understand the cultural pressure within a development team to leave the problem of exact GC for another day.

On the other hand, it is enormously easier for programmers to incrementally maintain and test such predicates than to attempt to retrofit them to an existing design. Unfortunately, the worst of this burden falls to the compiler, which as it happens, is also the source of the most rewarding short term performance gains. Thus the deferred gratification of designing for good memory management performance is directly at odds with the tempting rewards of quickly getting a JIT compiler working. It is easy to understand why memory management performance is often an afterthought.

Rather than face this software engineering challenge, scripting language implementors have often chosen reference counting or conservative garbage collection (GC). For example, AppleScript, Perl, Python, and PHP all use reference counting and Ruby initially used conservative GC. Reference counting is appealing because it is relatively simple to engineer. However, because reference counting is incomplete, implementors must (a) also implement tracing to detect cyclic garbage, or (b) prohibit cyclic data structures, or (c) never reclaim cyclic garbage. PHP chose (a), AppleScript chose (b), and Perl chose (c). Unfortunately, reference counting has abysmal performance [6, 8, 16]. Conservative GC performs better, but it cannot support copying which is critical to high performance GC [7, 8]. If the language implementors decide later to get serious about performance, they have a conundrum since retrofitting support for high performance collectors is typically very hard, if not impossible. *If the compiler implementation does not build on a foundation that uses the discipline of exact GC, fixing it later is a nightmare.*

When developers face enormous pressure to get a first implementation working quickly, they could commit to an iterative implementation strategy that starts with a throwaway prototype, adding support for exact GC in a later design. Although this approach is sound in principle, in practice the temptation to hold on to the prototype is too great for most development teams. This conundrum is vivid in the example of Perl, which has been intending to move to an entirely new engine (Perl 6) for over ten years now. We argue that the software engineering discipline of exact GC is not burdensome when followed from the beginning of VM development and actually improves compiler correctness as well [2, 18]. The development history of scripting languages such as Ruby, AppleScript, and Perl make a compelling argument for early investment and deferred gratification.

In this paper, we perform a case study of PHP memory characteristics to determine whether they warrant a high performance exact collector. Perhaps some scripting languages have simple allocation patterns that will not benefit from generational and other high performance collectors. The PHP scripting language was designed for web programming and it is the most used server-side scripting language for web development in use today. PHP programs create dynamic web content in response to user requests. Nonetheless, no study has examined the memory performance of the PHP language in depth. We use five benchmarks obtained from the PHP Group as a representative set of real workloads. We analyze the memory behavior of these benchmarks through heap composition graphs and object size demographics. The results show that there are a variety of object sizes, but most objects are small. Although the memory footprint of the PHP programs is small, many objects are short lived and there is a very high ratio of allocation to live objects. All of these characteristics and the expected growth in size and complexity of PHP programs suggest the need for some timely investment in a high-performance garbage collector.

2. Background and Related Work

Of the three routes to memory management we describe, the vast majority of scripting languages have taken routes (1) punting and going with either reference counting or conservative GC, or (2), using an existing managed runtime that supports exact garbage collection.

Reference counting and conservative GC When Perl and AppleScript were initially implemented, reference counting was their choice for GC. However, as mentioned above, reference counting is incomplete. For AppleScript, the decision to forbid cyclic data structures solved that issue. On the other hand, Perl still has an incomplete GC solution as of today [1].

The native implementation of Python used reference counting; only in 2000—about 10 years after it was developed—a cycle detection algorithm was added to make the GC complete. Unfortunately, if the developers of Python want to add a high-performing GC, some serious rework of the implementation would need to be done. For example, Python can never fully determine the root set [22], thus cannot to perform exact GC.

Similarly, Lua is a scripting language with extensibility as its main goal. Lua and classic Ruby originally implemented the conservative approach for GC—simple mark and sweep—in order to avoid the pitfalls of reference counting. However, as the demands on these languages have increased, the developers did attempt to tune the GC. Regrettably, because of the lack of write barriers in the original implementation, a change to a highly-tuned GC would require a deep change in the design.

Building on top of a Virtual Machine The Java Virtual Machine (JVM) and .NET's Common Language Runtime (CLR) are increasingly used as the implementation target for scripting languages. Perhaps best known among these are JRuby, Jython, IronPython, and IronRuby. The Parrot VM is designed specifically as a host for scripting languages [14], and is best known as the target for the long-awaited Perl 6. All three of these VMs support exact garbage collection. Although the JVM and CLR offer exact garbage collection, as far as we know, their collectors have not been tuned to the relatively short-lived use patterns of scripting languages.

PHP Memory Behavior Only Inoue et al. have studied PHP memory behavior [15]. Their work replaces the free-list allocator

in the PHP runtime with a region-based allocator and a variation of a region-based memory manager which they call *defrag-dodging*. Their PHP workloads are well suited to region based allocation because region memory management semantics map nicely to the transactional nature of PHP activities. Region collectors, however, are not general purpose, and as the demands and memory footprint of PHP applications grow they cannot adapt and will needlessly exhaust memory. Our results confirm Inoue et al.'s findings, but characterize the benchmarks' memory behavior rather than propose and measure a particular collector.

A recent effort has been made to improve the performance of PHP by retrofitting a well-optimized Just-In-Time (JIT) compiler for a Java Virtual Machine to PHP. Tatsubori et al. evaluated this JIT compiler-based implementation of a PHP runtime and showed their JIT-based acceleration of PHP yields good performance improvements [23]. Whereas they show PHP benefits from modern JIT technology, we show the performance of PHP programs will likely also benefit from modern garbage collection technology.

Trent et al. presented performance experiments that compare PHP and JSP, which is also a server-side scripting language [24]. Similar to Tatsubori et al., they show the code that these VMs produce impacts runtime, although often the web server performance itself dominates. They do not examine memory behavior or garbage collection performance.

3. Methodology

We use PHP as a case study for scripting language memory performance. This section describes the current PHP Zend™ virtual machine, the version we use, how we modify it to collect statistics, and the PHP benchmarks.

3.1 PHP memory management

PHP is a general-purpose scripting language that is mostly used for server-side web development to produce dynamic web content. We use the Zend Engine 2, which is the 'scripting engine' (i.e., a Virtual Machine) for the PHP 5 platform.

PHP is a garbage collected language. The Zend Engine 2 implements garbage collection with reference counting [12] and a general purpose memory allocator that supports single-object allocation, and single-object and bulk deallocation. At the end of a request, the collector performs bulk deallocation to clear the heap. During a request, the garbage collector reclaims individual single-object whenever it determines the object reference count is zero.

Every PHP variable is stored in a container called a *zval*. This *zval* container typically includes four fields: *type*, *value*, *isRef*, and *refCount*. The *isRef* field denotes whether the *zval* holds a reference or a value. The *refCount* field counts the number of incoming references to the *zval*. Whenever a *refCount* reaches zero, the garbage collector reclaims the container and reuses the memory.

An important limitation of reference counting is that it cannot reclaim objects that form a self-sustaining cycle of references, even if that structure becomes unreachable. Because such structures can be common, reference counting garbage collectors usually employ a back-up strategy to collect cycles. The Zend Engine 2 collects cycles using trial deletion, as specified by Bacon and Rajan [5]. Trial deletion adds possible roots of cycles to a buffer when a reference count to an object is decremented, but does not reach zero. Zend Engine 2 uses a fixed size buffer and when this buffer is filled, it performs the trial deletion algorithm.

3.2 Zend modifications for instrumentation

We modified the Zend Engine 2 to track allocation and object liveness. We modified the allocator to track the number and size of each object it allocates. To gather accurate lifetime statics, we modified the memory manager to reclaim the memory promptly. In the standard version, when the reference count decrements a *refCount* value to zero, the collector pushes the deallocation in to a buffer and later processes the buffer when it is full or the allocator is out of memory. In our modified version, the system promptly reclaims the memory after each reference count decrement.

Similarly, for cyclic data structures, instead of inserting all possible roots to cycles in a buffer, we perform trial deletion on each pointer installation that could be eliminating a reference to a cycle. We implemented this version by simply reducing the size of the trial deletion buffer to one. This change forces the memory manager to attempt to reclaim cyclic data structures as soon as possible.

Together, these modifications produce an accurate memory profile. At any time in the trace, we know exactly how many objects are allocated, when they were allocated, and whether they are live.

3.3 Configuration

We modified PHP 5.3.4, which is the latest stable branch of the PHP development trunk. We used Apache 2.2.14 for the HTTP server, and mysql 5.1.41-3 for the database server. We used http_load, an open source program by ACME labs, to fetch the URLs in order to test the throughput of the web server. Any execution of each the benchmarks will fetch more than a single URL.

3.4 Benchmarks

We use a set of benchmarks suggested to us by members of the PHP Group as a suite of real workloads that are representative of applications that run PHP to create dynamic web content. All of these applications are widely used on the Internet. They all create dynamic web pages on demand.

We use popular open-source content management systems: Drupal™ [3], Typo3 [4], and Xoops [19]. They vary in complexity and features, which differentiate their executions and memory characteristics. We also use: Qdig [20] a PHP script that dynamically presents digital image files as an online gallery or set of galleries, and Wordpress™ [25], a personal publishing platform.

4. Experimental Results

This section explores the memory behavior of the PHP benchmarks through three characterizations—gross allocation behavior, heap composition, and object size demographics. We use this analysis to characterize the benchmarks and contrast them with the demographics of SPECjvm98 benchmarks [8, 9]. We choose SPECjvm98, as opposed to the real-world more sophisticated Java DaCapo [10] benchmarks, because today's PHP benchmarks and VM technology are at a similar, early point in their development, as were Java benchmarks and VM technology when SPECjvm98 was introduced.

4.1 Gross Allocation Characteristics

Table 1 shows the total allocation and maximum live object size, expressed both in KB and number of objects. The table shows the mean and median allocated object size. Compared to SPECjvm98, these PHP benchmarks have substantially smaller total allocation and maximum live heap size. Column four shows the ratio of total allocation and maximum live size, also known as heap *turnover*. These PHP benchmarks as a whole have a very high turnover that ranges from 7:1 up to 54:1. This ratio is higher than SPECjvm98 benchmarks, which range from just 1:1 to 17:1. As the following sections show, the PHP heap is entirely empty at the end of each transaction. If we consider turnover ratios per transaction rather than for the entire execution, the ratios are reduced by around a factor of five, since most benchmarks run five transactions. These findings are consistent with Inoue et al.'s finding that PHP is well suited to region allocation [15].

Benchmark	Heap Volume (KB)			Heap Objects		Object Size (Bytes)	
	Allocated	Max Live	Ratio	Allocated	Max Live	Mean	Median
drupal	2,740	366	7:1	27,506	2,935	101	24
wordpress	3,688	207	17:1	23,184	831	162	24
typo3	74,514	4,949	15:1	684,164	37,512	111	20
qdig	39,415	725	54:1	216,268	11,744	186	28
xoops	114,249	3,125	36:1	1,085,267	40,312	107	24

Table 1. Allocation and maximum live in KB and objects, ratio of allocation to maximum live, average and median object allocation size.

4.2 Heap Composition Graphs

Figures 1(b) through 5(b) each show the heap composition in lines of allocation as a function of time which is measured in bytes of allocation. These graphs illustrate object lifetime behaviors visually with respect to allocation time. Each line in the heap composition graph represents a cohort of contemporaneously allocated objects. Cohort sizes vary depending on the program; we choose sizes as a power of two (2^n) bytes of allocation so that each graph contains between 100 and 200 cohorts. The cohort with the oldest objects allocated in the heap is represented by the top line. The size of the cohorts in bytes of allocation corresponds to the area between each of the lines. As objects in a cohort die, the area becomes smaller and the lines move closer together, or if all objects in a cohort die, the lines merge.

From these heap composition graphs, the first observation is that they all exhibit the expected transactional nature of PHP applications. This behavior stems from their on-demand creation of dynamic web pages. The sharp drop-offs following the peaks in all of the heap composition graphs are indicative that most objects do not live past each web page request. We did observe however that *drupal* and *wordpress* allocate longer lived data at the beginning of their execution. The graphs indicate this behavior with the cohorts at the top of the graph that have a thicker band. It is also important to note that there are thick and thin bands present for the entire duration of each of the phases, which indicate varying lifetimes within a transaction. For example, a long running transaction has some objects that do not adhere to a region lifetime model within the transaction, but at the end of each transaction, most objects are dead.

Although Java benchmarks represent a much wider class of applications, a few workloads in SPECjvm98 and the DaCapo Java benchmarks have very similar behavior. For example, Blackburn et al. show the heap composition graphs for the SPECjvm98 *javac* benchmark exhibits the same behavior [9]. Moreover, more sophisticated benchmarks, such as *antlr* from the DaCapo Java benchmark suite, have a very similar behavior as well [10].

4.3 Object Size Demographics

Figures 1(c) through 5(c) each show allocation histograms of object size demographics of each benchmark. The histograms are expressed as the percentage of total objects allocated (y-axis) by each workload for the object size classes (x-axis) through its entire execution. These histograms show a wide range of object sizes allocated by all workloads. The highest percentage of object sizes are between the sizes of 16 and 64 bytes. These numbers are very similar to those of Java programs [9, 10, 13]

Figures 1(c) through 5(c) present a different view on object size data. Rather than aggregating over the entire execution, the graphs present the object size demographics as a time series. We create this graph by frequently measuring the object size demographics of the program (just as in Figures 1(c) through 5(c)). We take a large number of samples throughout the execution of the program, rather than just measuring this data at the end of the execution. Each size class is represented by a line stacked upon the next smallest size class. The smallest size classes are at the bottom. The distance between the lines indicate the number of live objects allocated of the corresponding size as a function of time in bytes of allocation.

Together, the histogram and time series graphs indicate the predominance of the small object size classes, both as a function of allocated objects and as a function of the number of live objects in the heap at any given time.

In summary, these object size and lifetime demographics are similar to other managed languages and are therefore very likely to benefit from high performance memory management approaches, such as region and generational collection.

5. Discussion and Conclusions

Efficient memory management is crucial to high performance languages. Moreover, current trends in hardware and software indicate that memory management will further increase in importance over time. Meanwhile as scripting languages such as Ruby, Python, JavaScript, and PHP take on more critical roles, performance is becoming increasingly important. We argue that if scripting language developers have aspirations of high performance, they must opt for deferred gratification and take memory performance seriously, engineering it in from the get-go. This discipline is not burdensome for developers and will yield a better engineered and flexible system. The alternative is a well worn path that starts down the easy road of reference counting or conservative GC and ends with a system that has a good compiler but is hamstrung by poor memory performance.

We examine this problem through a case study using PHP. We present a thorough evaluation of the memory profiles of a number of popular PHP applications. These applications have modest total memory allocation but higher turnover rates compared to the early SPECjvm98 benchmarks. The average object size and heap profiles, however, have many similarities to those of Java applications. These characteristics indicate that PHP will benefit significantly from modern garbage collection techniques, including copying and generational collection.

We argue that implementers with an interest in performance should, but generally do not, take memory management very seriously when they start building a new language implementation. We recommend designing for exact GC from the get-go or targeting an existing VM. The former enforces a software engineering discipline on language implementers that is exacting, but not onerous, while the later suggests an opportunity to tune VM memory management to scripting language workloads.

6. Acknowledgements

We thank Andi Gutmans and Zeev Suraski, co-creators of PHP 3 and the Zend Engine, and Dmitry Stogov of Zend Technologies Ltd. for their assistance, particularly with the PHP benchmarks in this study.

References

[1] J. Allen. Perl documentation, 2010. URL http://perldoc.perl.org/perlobj.html.

[2] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño — a compiler-supported Java Virtual Machine for servers. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software*, Atlanta, GA, May 1999. ACM.

[3] D. Association. Drupal, 2011. URL http://drupal.org/.

[4] T. Association. Typo3, 2011. URL http://typo3.com/.

[5] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *European Conference on Object-Oriented Programming, ECOOP 2001, Budapest, Hungary, June 18-22*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235. Springer-Verlag, 2001.

[6] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, CA, Oct. 2003.

[7] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality. In *ACM Conference on Programming Language Design and Implementation*, pages 22–32, Tuscon, AZ, June 2008.

[8] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, Dept. of Computer Science, Australian National University, 2006. http://www.dacapobench.org.

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[11] D. Burger, A. Kägi, and J. R. Goodman. Memory bandwidth limitations of future microprocessors. In *ACM/IEEE International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[12] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3:655–657, December 1960. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367487.367501. URL http://doi.acm.org/10.1145/367487.367501.

[13] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference on Object-Oriented Programming*, June 1999.

[14] P. Foundation. Parrot virtual machine, 2011. URL http://www.parrot.org.

[15] H. Inoue, H. Komatsu, and T. Nakatani. A study of memory management for web-based applications on multicore processors. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 386–396, 2009.

[16] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 367–380, Tampa, FL, Oct. 2001. ACM.

[17] C. Nutter. Interview: What makes jruby tick in 2010?, 2010. URL http://www.rubyinside.com/in-depth-jruby-qa-what-makes-jruby-tick-in-20%10-2971.html.

[18] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 1–12, April 2001.

[19] T. X. Projects. Xoops, 2011. URL http://xoops.org/.

[20] Qdig. Quick digital image gallery, 2011. URL http://qdig.sourceforge.net/.

[21] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *ACM/IEEE International Symposium on Computer Architecture*, pages 371–382, 2009.

[22] N. Schemenauer. Garbage collection for python, 2000. URL http://arctrix.com/nas/python/gc/.

[23] M. Tatsubori, A. Tozawa, T. Suzumura, S. Trent, and T. Onodera. Evaluation of a just-in-time compiler retrofitted for php. In *ACM International Conference on Virtual Execution Environments*, pages 121–132, 2010.

[24] S. Trent, M. Tatsubori, T. Suzumura, A. Tozawa, and T. Onodera. Performance comparison of PHP and JSP as server-side scripting languages. In *ACM/IFIP/USENIX International Conference on Middleware*, Middleware, pages 164–182, 2008.

[25] Wordpress. Wordpress web software, 2011. URL http://wordpress.org/.

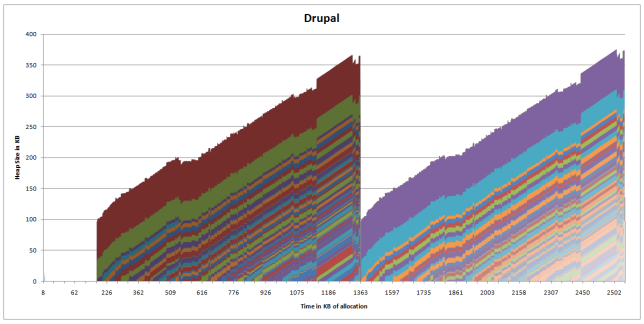
[26] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 361–376, 2009.

Notes on Trademarks

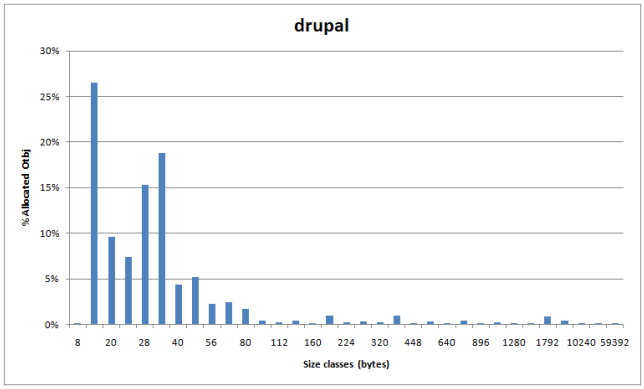
We add the trademark symbol on the first reference in the text to a trademark name and enumerate them again here. AppleScript™ is a trademark of Apple Computer, Inc. Oracle™, Java™, and JavaScript™ are registered trademarks of Oracle and/or its affiliates. Zend™ is a trademark of Zend Technologies Ltd. Drupal™ is a registered trademark of Dries Buytaert. WordPress™ is a trademark of WordPress Foundation, Inc.

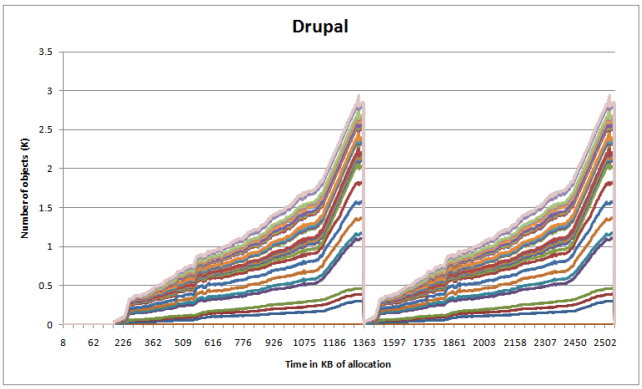
Total allocation	(KB)	2,740
	(Obj)	27,506
Maximum Live	(KB)	366
	(Obj)	2,935

(a) Benchmark Characteristics



(b) Heap Composition Graph



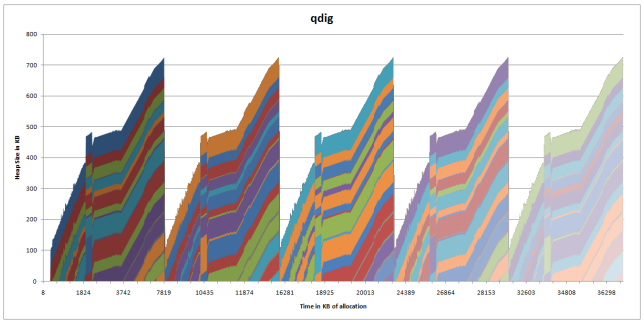


(e) Object Size Demographics

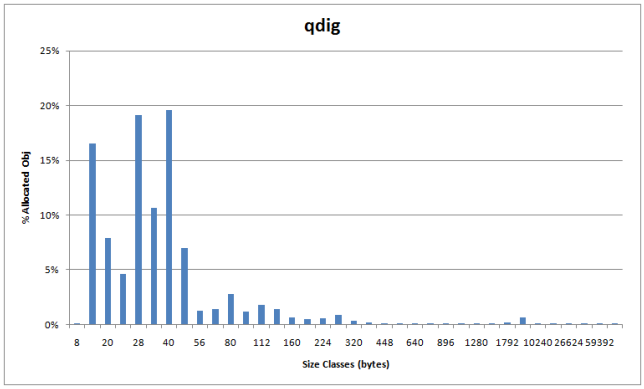
Figure 1. Benchmark Characteristics: drupal.

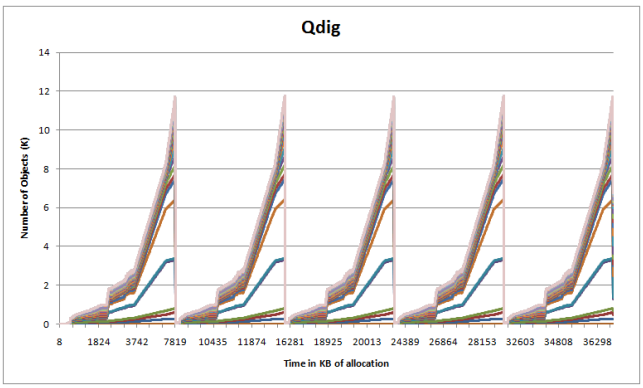
Total allocation	(KB)	39,415
	(Obj)	216,268
Maximum Live	(KB)	725
	(Obj)	11,744

(a) Benchmark Characteristics



(b) Heap Composition Graph



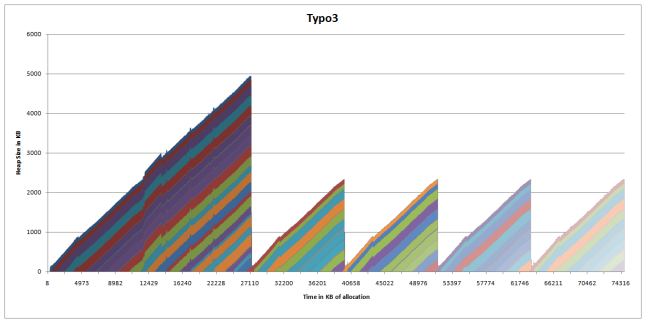


(e) Object Size Demographics

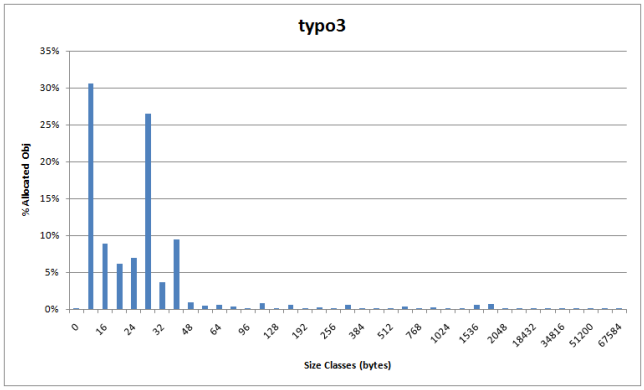
Figure 2. Benchmark Characteristics: qdig.

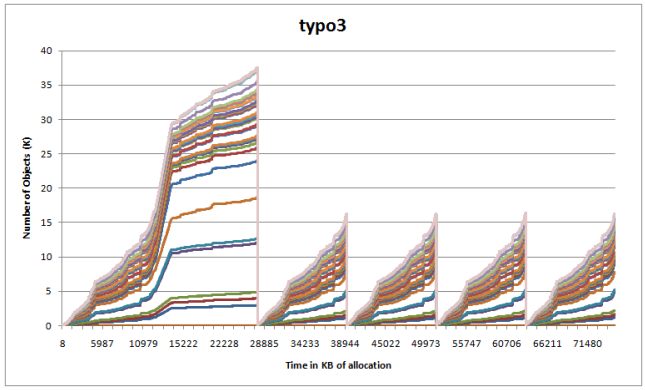
Total allocation	(KB)	74,514
	(Obj)	684,164
Maximum Live	(KB)	4,949
	(Obj)	37,512

(a) Benchmark Characteristics



(b) Heap Composition Graph



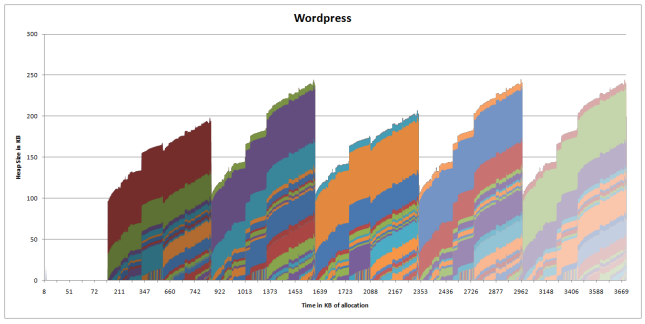


(e) Object Size Demographics

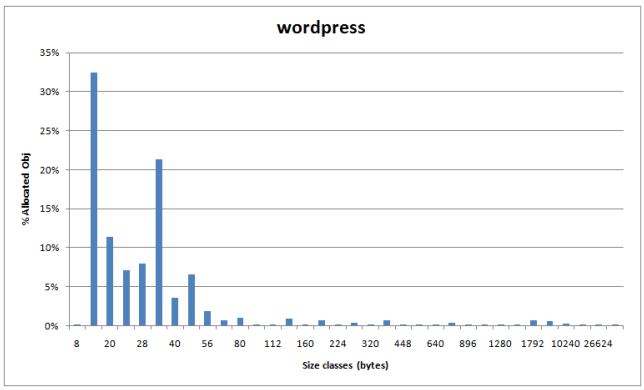
Figure 3. Benchmark Characteristics: typo3.

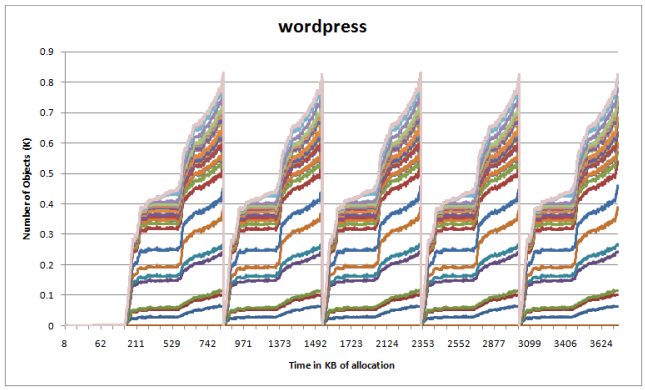
Total allocation	(KB)	3,688
	(Obj)	23,184
Maximum Live	(KB)	207
	(Obj)	831

(a) Benchmark Characteristics



(b) Heap Composition Graph



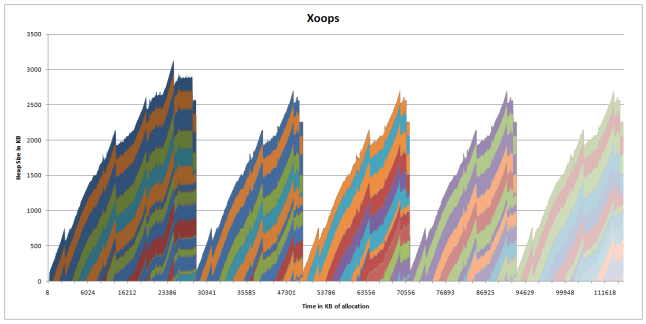


(e) Object Size Demographics

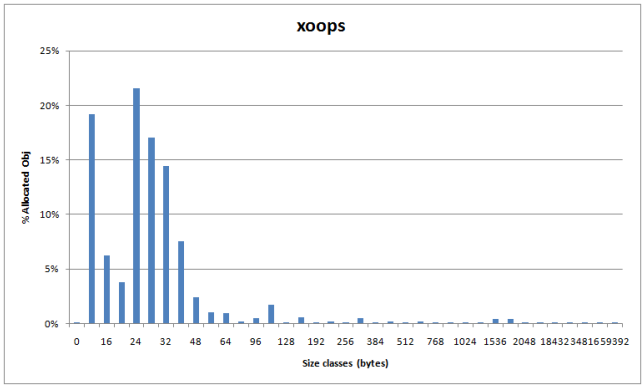
Figure 4. Benchmark Characteristics: wordpress.

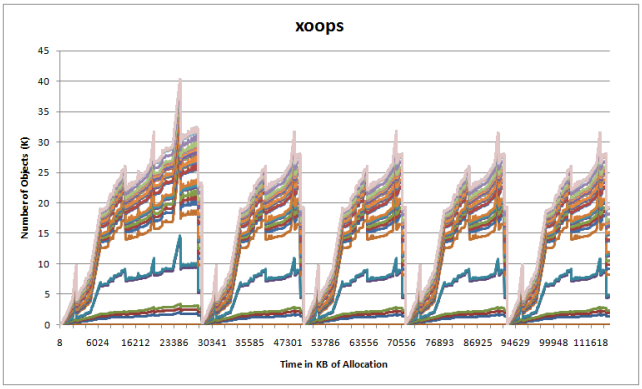
Total allocation	(KB)	114,249
	(Obj)	1,085,267
Maximum Live	(KB)	3,125
	(Obj)	40,312

(a) Benchmark Characteristics



(b) Heap Composition Graph





(e) Object Size Demographics

Figure 5. Benchmark Characteristics: xoops.