Biased Allocator for Generational Garbage Collector

Hyung-Kyu Choi, Seoul National University
HyukWoo Park, Seoul National University
Soo-Mook Moon, Seoul National University

Virtual machines often employ garbage collection (GC) to reclaim dead objects automatically. Various GC techniques have been used, but the generational GC is known to show the best performance. It divides the memory into the young area and the old area, and objects are allocated to the young area only such that some of the objects that survive from a few GCs on the young area can be promoted to the old area. Since young objects are likely die soon, we can perform GC on the young area more often than the old area, which improves the overall GC efficiency. However, we observed that some objects are likely to be immortal inherently, so it would be desirable to allocate them directly to the old area. We propose such a memory allocator called biased allocator, which analyzes the code to identify objects to be allocated to the old area. Our preliminary evaluation shows that the biased allocator can reduce an average of 4.1% of the GC pause time of the generational GC while reducing the number of promotions significantly.

Categories and Subject Descriptors: D.3.4 [**Processors**]: Memory management (garbage collection)

General Terms: Algorithm, Performance, Experimentation

Additional Key Words and Phrases: object allocation, generational garbage collector, memory management

ACM Reference Format:
DCE 2014 V, N, Article A (January 2014), 10 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Virtual machine adopts automatic memory management to manage the heap. Automatic memory management reclaims objects which are not used anymore automatically from the heap, although object allocation is requested explicitly by a program. Garbage collection is a famous approach to find unnecessary objects, i.e. dead object, and reclaim them [Jones and Lins 1996]. Allocation strategies and garbage collection should be considerate each other, since garbage collector is responsible for securing and managing free space which is used by allocator later to allocate objects. We can consider garbage collector a producer of free space and then allocator can be a consumer of free space. Therefore some garbage collectors enforce allocation methods considering fragmentation, performance and throughput. In vice versa, some allocation strategies are more efficient with specific garbage collectors. Various garbage collectors have been proposed by many researchers [Appel 1989; Jones and Lins 1996; Hertz et al. 2005; Xian et al. 2007; Reames and Necula 2013]. In detail, there have been different approaches to find dead object and also there are various ways to secure free space. One of simplest way to find unnecessary objects is traversing pointers recursively from always live objects, which are called roots, to find reachable objects which are live and necessary. Another approach maintains counters for incoming references to each object at runtime to determine a liveness of object [Levanoni and Petrank 2001]. There are also several ways to secure free space after identifying dead objects. A simplest way maintains a list of free space by reclaiming dead objects. Another approach secures free space by moving live objects to different area, as a result previous area contains only dead objects and whole previous area can be considered to be free space. There are so many garbage collectors depending on how they identify dead objects and how they secure free space. Among them, a generational garbage collector is

This work was supported by the Ministry of Trade, Industry & Energy (MOTIE) through the Electronics and Telecommunications Research Institute (ETRI) (Project No. 10045344)

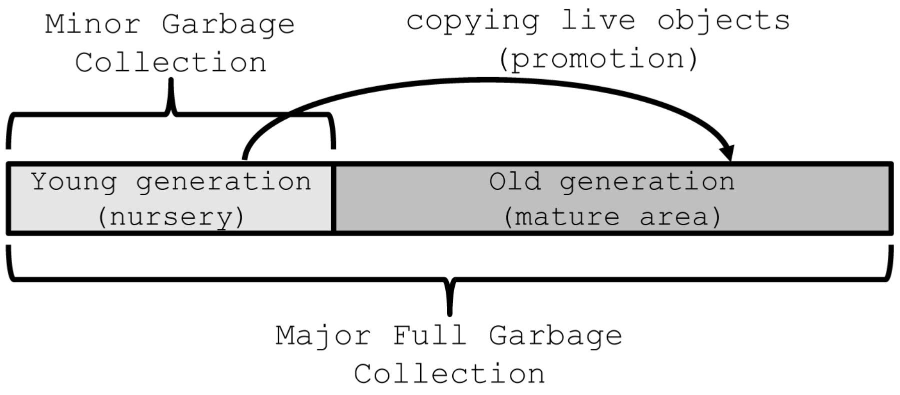
Minor Garbage Collection
copying live objects (promotion)
Young generation (nursery)
Old generation (mature area)
Major Full Garbage Collection

Fig. 1. A generational garbage collector with two generations.

famous and widely adopted in virtual machines, e.g. Java Virtual Machine from Oracle [HotspotMM 2006].

A generational garbage collector manages the heap by splitting whole heap into several generations from young to older. With a generational garbage collector, a new object is always allocated from a nursery area which is one of generations and considered to contain young objects. Then later if a nursery is overpopulated and there is no room for new objects, a generational garbage collector secures free space from a nursery by moving live objects to older generations. We call this object copying a promotion, since an object is promoted to old generations. Such garbage collection on a nursery is called minor garbage collection. Later we have to reclaim all dead object in young and old generations too when old generations are also overpopulated. We call it a major garbage collection or a full garbage collection. Figure 1 depicts how a simple two generational garbage collector works. Due to various advantages, a generational GC is adopted in many virtual machines for various environments. First, a garbage collection can be completed in a short time when minor GC is requested instead of full GC, because minor GC performs only on a nursery which is relatively smaller than whole heap. Although number of garbage collections increases relatively, each pause time caused by garbage collection is reduced and responsiveness of virtual machine is improved when compared to a garbage collector with only full GC. Furthermore secured free space from a nursery is continuous and fragmentation free since whole young generation is empty after all live objects are promoted. There are many variations of generational garbage collector depending on number of generations, size of young and old generations and etc. [Appel 1989; Doligez and Leroy 1993; Xian et al. 2007] However a generational garbage collection has unavoidable runtime overhead and it shows undesired behaviors in some cases.

A generational garbage collector has to promote live objects to older generations to clear up a nursery. Each promotion contains not only copying an object but also updating pointers which refer to the object just moved to a new location. A generational garbage collector is beneficial when only few objects are live and most of objects in young generation are dead. However when many of objects in a nursery are live and is going to be promoted, the overhead of promotion increases to hide advantages of a generational GC. In worst case when every object in a nursery is live, we have to promote all objects and minor GC does not reclaim dead object at all with overhead of minor GC and promotions. We suggest that such overhead can be avoided if we place promoted objects to old generations instead of young generation when those objects are being allocated at first. We are going to segregate objects in various ways to reduce number of objects allocated to a nursery. Rest of the paper is composed as follows. In the next Section 2, we address the problem in detail and propose an approach to exploit biased allocators to improve a generational garbage collector. Then we propose a way to invoke biased allocator and describe three analyses to identify objects to be allocated with biased allocators in Section 3. We describe how to combine proposed analyses and how we implemented proposed approaches in real environment in Section 4. In Section 5, proposed approaches are evaluated on a real embedded device. Section 6 summarizes the paper and discusses future works.

2. MOTIVATION

As we discuss in the previous section, a generational garbage collector itself suffer from inherent overhead of promotion. As a result pause time of each garbage collection can be increased to compensate advantages of a generational GC. There have been many researches to improve a generational GC [Jones and Lins 1996; Xian et al. 2007; Reames and Necula 2013] , but most of them require modifying a garbage collector itself and a garbage collector is getting more complicated which is hard to predict the effect modification in various situation.

We propose an approach to exploit an allocation instead of a generational GC to overcome the undesired overhead of a generational GC. We already address that such an undesirable behavior of a generational GC is due to promotion of many objects in a nursery. In other words, such objects live long to the time when minor GC is requested to reclaim dead objects. We are going to avoid the situation by simply locating such objects in old generations instead of a nursery when objects are allocated. Simply we can allocate all objects in old generations, but then it is not a generational GC anymore and may suffer from long pause time of full GC instead. Therefore we have to choose a set of objects and allocate them to old generation using biased allocators. In the following section, we propose a way to make use of biased allocators and describe how to identify objects to be biased in detail.

3. BIASED ALLOCATOR

With biased allocators, an object can be allocated to heap in different ways depending on various properties to improve the performance of heap management with a generational garbage collector. On the other hand, traditional virtual machine with a generational GC allocates an object to a nursery area of heap with a single same allocator. We propose that we affect the performance of heap management in a beneficial way by reducing copying overhead of generational garbage collection if we allocate an object to other than a nursery carefully with different object allocators. In this section, we will discuss when to choose an allocator and propose a way to make a decision with less runtime overhead. Then we will describe three analyses to select an allocator.

3.1. When to choose an allocator

We can choose an allocator every time when an object is being allocated to the heap. It would be best if we can perform fine-grain analysis for each object and decide allocation area for each object. However it is not easy to predict lifetime of each object precisely and there will be extra overhead if we choose an allocator every time an object is being allocated. Usually an object allocation occurs very frequently and an additional computation could harm overall performance. Therefore it would beneficial to runtime performance if we can choose an allocator without extra overhead of an allocation itself.

A *new* bytecode in Java Virtual Machine always knows a type of an object to be allocated [Lindholm and Yellin 1999] and the *new* bytecode allocate objects of same type. Therefore we are going to exploit the property that each *new* bytecode always allocates isomorphic type of objects at runtime.

Also we try to reduce the overhead of decision making by making a decision once and use the same decision later. To achieve these, we choose an allocator when bytecode are analyzed and being translated into native machine code to improve overall performance. A Just-in-time compiler (JITC) which translates bytecode into machine code at runtime is a famous acceleration technique [Krall 1998; Yang et al. 1999; Arnold et al. 2000; Aycock 2003] as well as an Ahead-of-time compiler (AOTC) where the translation is occurred before bytecode is being executed [Proebsting et al. 1997; Varma and Bhattacharyya 2004; Nilsson and Robertz 2005] .

A biased allocator is chosen when a *new* bytecode is being translated into machine code depending on the type of object to be allocated. Then the *new* bytecode is translated into a machine code which allocates an object with the selected allocator. In this way, we make a decision once and an allocation is done without additional overhead other than that the allocator allocates an object in a different way.

3.2. How to choose an allocator

Even though that a specific *new* bytecode accepts an isomorphic type, it is not easy to exploit the information to select an allocator wisely. We need whole type analysis on a Java program to make a correct decision and it is not eligible for a JITC or AOTC, because whole type analysis including class hierarchy analysis [Dean et al. 1995; Snelting and Tip 2000] is not a simple problem and it takes much time. Therefore we consider three information as well as simple type information, i.e. class information which is known directly from the *new* bytecode itself.

First we identify a location where local-scoped objects are allocated. Also an allocation site within a loop is identified and being chosen to use a biased allocator. Finally we analyze the use of an allocated object which is assigned to static fields and identify locations where the object is allocated. Since an object can be allocated from multiple locations depending on control flow, we exploit traditional iterative data follow analysis. Of course, type information is always considered together with three properties.

3.2.1. Local-scoped objects. An object is known to be locally scoped if an object is live only within a specific scope. A scope can be anything such as a basic block, a super block, a trace, a method or even a program. There have been many researches to identify locally scoped objects and escape analysis is one of famous technique to identify locally scoped objects. Escape analysis has been used in Java to make use of stack allocation [Choi et al. 1999; Gay and Steensgaard 1999] to relieve memory pressure on the heap and adopted in various JVM such as Java Standard Edition 6 [HotspotVM 2013]. We use an escape analysis to identify an allocation site where objects being allocated are locally scoped. We expect that such an allocation site can make use of traditional allocator or even stack allocator which uses a stack instead of the heap, because locally scope objects are only live within a specific scope and liveness is limited to the scope which can be consider being relatively shorter than other objects which escape the scope. As a result, we don't have to consider such allocation sites for being a candidate for biased allocation to improve the performance of heap management with garbage collection.

3.2.2. Objects allocated inside loops. Loops have been a famous target for an optimization, because many programs spends most of the time in loops and small improvement in a loop can be result in large runtime improvement of the performance due to its repetition. We also look into loops, because an allocation in loops will continue allocate same type of objects until loop stops and quite large amount of objects are allocated inside of the loops.

We expect that objects allocate inside loops are relatively short lived compared to objects allocated outside of loops, because loops usually perform same computation repetitively and many objects allocated within loops are for temporary use. We decide objects allocated inside loop to be possibly short-lived at first. However we find that some objects, which are allocated in a loop but have relatively small size, are long-lived. Therefore allocation sites within loops are chosen when smaller objects are allocated. We can easily compute the size of objects, because the type of object being allocated is identified directly from *new* bytecode as we described before. We don't have to worry about leaving large objects behind in young area, because promotion overhead is more dominated by number of objects being promoted than size of objects as we discussed in previous sections.

3.2.3. Objects assigned to static fields. There are two types of objects in Java, i.e. an instance object and a class object. An instance object is an instance of a specific class which are usually allocated with *new* keyword of Java language and object we talked before in this paper are all instance objects. A class object is a unique object of a specific class and they are usually created implicitly by Java virtual machine when the class is being resolved. A static field is a field not related to an instance object but class object itself. Since a static field looks like a global variable, researches have shown that an object assigned to a static field tend to be immortal, i.e. never dead till the program ends [Hirzel et al. 2002].

We decide to make use of this property and use biased allocators for such allocation sites where any object allocated can be assigned to static fields. We make use of traditional analysis of reaching definition to identify allocation sites on the compilation unit. Candidate allocations sites can be one or more and even we can't find a site, because we perform analysis only within the compilation unit.

Of course, some candidate allocation sites can be duplicated with the previous analysis, i.e. allocation sites within loop. We will discuss how we arrange three analyses we discussed here to make a decision for biased allocation in the following section.

4. ANALYSIS AND IMPLEMENTATION

Each allocation site can have three properties, i.e. local, loop and static. Local means this allocation site allocates objects which are live only within the scope. Loop means this allocation site is located within loops and size of allocation is larger than threshold. Static means this allocation site allocate objects which can be possibly assigned to static fields. Only allocation sites which is neither local nor loop are selected for biased allocation. Then we find allocation sites with static property and add them to candidates and we are going to describe how we make use of three analyses.

At first, we assume that all allocation sites are candidate for biased allocation. We find locally scoped object with escape analysis. After we identify allocation sites which only allocate locally scoped objects, we remove those sites from candidates. We do not discard the list of allocation sites that are local and keep the list for later use.

We continue to identify allocation sites within loop and this analysis can be done with other traditional loop optimizations as well. However this analysis should be done after any control flow changes or code motions are made to loop, because the location of an allocation site can be changed with those optimizations and even allocation sites can be eliminated after optimizations. Furthermore we do not analyze and skip allocation sites which are already identified to allocate only locally scoped objects from previous escape analysis. Then we reduce candidate allocation sites with results of loop analysis. We find out that some allocation sites within loop allocate only locally scoped objects and it is obvious that these objects have relatively shorter lifetime than other objects which escape the same scope.

$$\begin{aligned}\texttt{Candidate allocation sites} = \{\ &\texttt{all allocation sites}\\ &-\ \texttt{allocation sites}_{\texttt{local}}\\ &-\ \texttt{allocation sites}_{\texttt{loop}}\ \}\\ +\ &\texttt{allocation sites}_{\texttt{static}}\end{aligned}$$

Fig. 2. Candidate selection with three analyses

Finally we look into every assignment of an object to static fields and try to identify one or more allocation sites where the object was allocated. This analysis should be done just before the code generation, because any control and data flow changes can affect the result of this analysis. After we identify allocation sites, we add those allocation sites to candidates for biased allocation. In short, we can formulate above sequences as in Figure 2. We should keep the order of local, loop and static analysis, because there can be an allocation site which reside in loop and allocate large objects, but allocates objects which can be assigned to static fields. Of course there is no allocation site which allocates locally scoped object and allocates objects assigned to static fields, because an object is not locally scoped if there is any assignments of an object to static fields.

We implemented these analyses on Oracle's phoneME Advanced MR2 version where ahead-of-time compiler (AOTC) [Proebsting et al. 1997; Varma and Bhattacharyya 2004; Nilsson and Robertz 2005] is available for translating Java bytecode to native machine code with optimizations where proposed approaches had been inserted. Since a translation unit of the AOTC is a method, our analyses were also done within a method scope.

5. EVALUATION

We evaluate our proposed analysis on phoneME Advanced MR2 [PhoneME] with digital TV (DTV)[Interactive TV] set-top box which includes MIPS based core with 128MB main memory. We make use of AOTC to perform proposed optimization and observed the effect of biased allocation without runtime overhead of analysis. Java applications have been translated by AOTC before running and stored in set-top box for evaluation. We use six micro benchmarks from specjvm98 [SPECjvm98 1999] to evaluate our approaches. We choose a generic generational garbage collector with two generations in phoneME Advanced MR2 to reclaim objects while running specjvm98. Since total pause time due to garbage collections is relatively small compared to total running time, we compared total pause time separately instead of total running time and measured the amount of promotions occurred in generation garbage collections.

5.1. Total pause time of garbage collections

We measured total pause time of garbage collections before and after applying proposed approaches and compared them in Figure 3. About up to 12.2% of total pause time caused by garbage collection has been reduced and about 4.1% of pause time is removed in average. Figure 4 depicts the size of biased objects compared to total size of objects allocated. We identify lots of objects from _209_db where pause time has been reduced most. However even we biased more than 10% of objects from _228_jack, total pause time is not reduced much as we expected compared to other programs and we can't find direct correlations between the size of biased objects and total pause time. After careful examination, we find out that total pause time of generational garbage collector is affected by various factors and it is very hard to predict. For example,

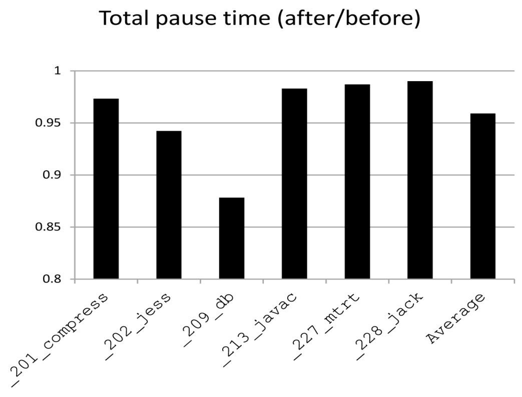

Fig. 3. Ratio of total pause time after applying biased allocation compared to non-biased allocation

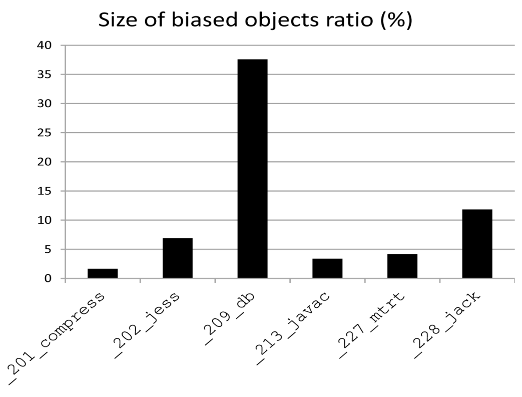

Fig. 4. Ratio of biased objects size compared to total objects

size and number of objects allocated in nursery area affect pause time. When promotion occurs, more factors affect pause time of generational garbage collection, because a promotion includes copying an object and updating pointers to copied object. Even worse promotions may incur a full major garbage collection when there is no sufficient space in a mature area.

On the other hand, our approaches may consume a mature area more aggressively due to false detection. Three analyses we proposed are all based on static analysis without runtime information. Therefore we can't predict exact life time of objects and availability of the heap is not concerned at all. As a result proposed approach may induce side effects in unexpected ways due to exploiting a mature area much more than a nursery area. However it is not easy to calculate lifetime of objects exactly and our research is a start point to exploit different allocation based on analyses. We will discuss these matters in the last section again with future works.

5.2. Effect of each analysis

We also evaluated the effect of each proposed analysis in Figure 5. When we choose objects with an escape analysis, we can't reduce total pause time of garbage collections effectively. We found that total pause time has been reduced much after analyzing loops. Even though we decide to bias objects which are allocated to static fields towards old generation, Figure 5 shows that there is only a little improvement with this optimization. However it is expected, because objects assigned to static fields are rarely overwritten and few allocations are related to static fields. Of course, there are some allocation sites where few objects are assigned to static fields and other objects are discarded soon. A proposed analysis may decide those allocation sites to be candidate for biased allocation but those are not desirable choices, because we want to allocate objects that live long. Therefore those candidates can be false-positive. Nevertheless it reduces pause time slightly in average.

5.3. Pause time of each garbage collection

We also examine each garbage collection to evaluate biased allocation. Since behavior of garbage collections is changed after applying proposed optimizations, it is not reasonable to compare each garbage collections one-to-one. For example, garbage collections are invoked at different phase of a program and each garbage collection may reclaim different objects after applying optimizations. Therefore we choose the first

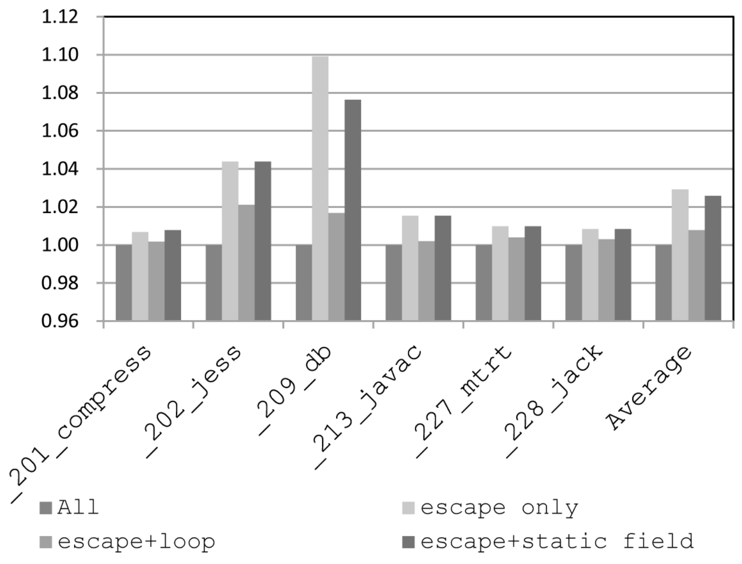

Fig. 5. Ratio of total pause time of garbage collections compared to all analysis enabled. Therefore All is always one.

five garbage collections of _209_db where promotion occurs and compared number of promotions to original garbage collections as in Figure 6. We choose these five garbage collections, because they behaves different but not totally different. Even though it is not fair to compare them one-to-one, it is easily noticed that total number of promotion occurred in the first five garbage collections have been reduced about 25%. All five garbage collections have less number of promotions than original garbage collections. This is expected results, since biased allocator try to allocate objects in a mature area other than in a nursery where some objects should be promoted later. The first garbage collection has been invoked more lately than before, because a nursery is less populated after applying biased allocation.

6. SUMMARY AND FUTURE WORKS

We proposed a way that different allocators can cooperate with garbage collectors which have a critical role in memory management of virtual machine. For a generational garbage collector, we proposed approaches which make use of existing analysis techniques to relieve the side effect of generational garbage collector. Allocation sites have been chosen and biased with three analyses and each biased allocation site uses new biased allocators instead of original allocator. We implement a proposed approach in real embedded Java device and evaluate the effectiveness. Total pause time of garbage collections has been reduced and promotion overhead of generational garbage collection has been also reduced in overall.

However we can't guarantee correctness of biased allocation with analyses discussed in this paper. Furthermore analyses discussed in this paper are done at static-time and does not make use of any runtime information. We expect that analyses can be more accurate if runtime information is provided. Each allocation site use same allocator after decision had made. We expect allocators can be chosen adaptively or allocator itself can evolve for further improvement. Also we use only single biased allocator to bias objects but more allocators can be used for various garbage collectors. We are

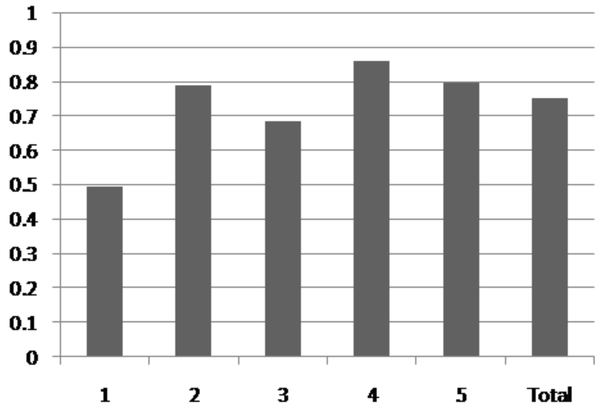

Fig. 6. Ratio of promotions occurred for the first five garbage collections with biased allocator compared to original in _209_db.

also expecting that there are opportunities for biased allocation to improve garbage collectors other than generational.

REFERENCES

APPEL, A. W. 1989. Simple generational garbage collection and fast allocation. *Software: Practice and Experience 19,* 2, 171–183.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive optimization in the Jalapeno JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '00. ACM, New York, NY, USA, 47–65.

AYCOCK, J. 2003. A brief history of just-in-time. *ACM Comput. Surv. 35,* 2, 97–113.

CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. 1999. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '99. ACM, New York, NY, USA, 1–19.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP '95. Springer-Verlag, London, UK, UK, 77–101.

DOLIGEZ, D. AND LEROY, X. 1993. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. ACM, New York, NY, USA, 113–123.

GAY, D. AND STEENSGAARD, B. 1999. Stack allocating objects in Java. Tech. rep., Microsoft Research.

HERTZ, M., FENG, Y., AND BERGER, E. D. 2005. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. ACM, New York, NY, USA, 143–153.

HIRZEL, M., HENKEL, J., DIWAN, A., AND HIND, M. 2002. Understanding the connectivity of heap objects. In *Proceedings of the 3rd International Symposium on Memory Management*. ISMM '02. ACM, New York, NY, USA, 36–49.

HotspotMM 2006. Memory management in the Java HotSpot virtual machine. http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf.

HotspotVM 2013. Java HotSpot virtual machine performance enhancements. http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html.

Interactive TV. Interactive tv web. http://www.interactivetvweb.org.

JONES, R. AND LINS, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 1st Ed. John Wiley and Sons, Inc., New York, NY, USA.

KRALL, A. 1998. Efficient JavaVM just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. PACT '98. IEEE Computer Society, Washington, DC, USA, 205–.

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. ACM, New York, NY, USA, 367–380.

LINDHOLM, T. AND YELLIN, F. 1999. *Java Virtual Machine Specification* 2nd Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

NILSSON, A. AND ROBERTZ, S. 2005. On real-time performance of ahead-of-time compiled Java. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*. IEEE Computer Society, Washington, DC, USA, 372–381.

PhoneME. Phoneme project. https://java.net/projects/phoneme.

PROEBSTING, T. A., TOWNSEND, G., BRIDGES, P., HARTMAN, J. H., NEWSHAM, T., AND WATTERSON, S. A. 1997. Toba: Java for applications a way ahead of time (wat) compiler. In *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*. COOTS'97. USENIX Association, Berkeley, CA, USA, 3–3.

REAMES, P. AND NECULA, G. 2013. Towards hinted collection: Annotations for decreasing garbage collector pause times. In *Proceedings of the 2013 International Symposium on Memory Management*. ISMM '13. ACM, New York, NY, USA, 3–14.

SNELTING, G. AND TIP, F. 2000. Understanding class hierarchies using concept analysis. *ACM Trans. Program. Lang. Syst. 22,* 3, 540–582.

SPECjvm98 1999. SPECjvm98 documentation. http://www.spec.org/osg/jvm98/jvm98/doc/index.html.

VARMA, A. AND BHATTACHARYYA, S. S. 2004. Java-through-C compilation: An enabling technology for Java in embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3*. DATE '04. IEEE Computer Society, Washington, DC, USA, 30161–.

XIAN, F., SRISA-AN, W., JIA, C., AND JIANG, H. 2007. As-gc: An efficient generational garbage collector for Java application servers. In *Proceedings of the 21st European Conference on Object-Oriented Programming*. ECOOP'07. Springer-Verlag, Berlin, Heidelberg, 126–150.

YANG, B.-S., MOON, S.-M., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOGLU, K., AND ALTMAN, E. 1999. Latte: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*. PACT '99. IEEE Computer Society, Washington, DC, USA, 128–.