Connectivity-Based Garbage Collection

Martin Hirzel

Vordiplom, Darmstadt University of Technology, 1998

M.S., University of Colorado at Boulder, 2000

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
2004

This thesis entitled:
Connectivity-Based Garbage Collection
written by Martin Hirzel
has been approved for the Department of Computer Science

Amer Diwan

William M. Waite

July 2004

The final copy of this thesis has been examined by the signatories,
and we find that both the content and the form meet acceptable
presentation standards of scholarly work in Computer Science.

Abstract

Hirzel, Martin (Ph.D., Computer Science)

"Connectivity-Based Garbage Collection"

Thesis directed by Amer Diwan

Garbage collection is an important feature of modern programming languages: by liberating the programmer from the responsibility of freeing up unused memory by hand, it leads to code with fewer bugs and a cleaner design. However, these software engineering benefits have their costs: garbage collection may incur disruptive pauses, slowdowns, and increased memory requirements. This dissertation presents a novel family of garbage collection algorithms that use information about the connectivity of heap objects to reduce these costs.

This dissertation firsts presents empirical data showing that connectivity information is a good indicator for when objects die. Then, it describes connectivity-based garbage collection, a new family of garbage collectors that exploit connectivity properties to yield short pause times, good throughput, and low memory footprint. It explores the performance of a variety of connectivity-based garbage collectors in a simulator. These collectors rely on an analysis of object connectivity; this dissertation describes the first non-trivial pointer analysis that handles all of Java, including dynamic class loading, reflection, and native code. The dissertation concludes by describing the design and implementation of a connectivity-based garbage collector in a Java virtual machine, using the pointer analysis.

Acknowledgments

First and foremost, I am grateful to Amer Diwan for being an excellent academic advisor and a great friend. He sparked my interest in programming languages, he taught me how to conduct research in experimental computer science, he honed my skills in writing papers and giving talks, and he introduced me to his many friends in the academic community. Getting a Ph.D. under Amer's guidance was a great privilege!

I thank my collaborators on the research that this dissertation is based on. Amer Diwan and Mike Hind were mentors on all contributions, from the preliminary experiments in Chapter 3 all the way to the real-world implementation in Chapters 10 and 11. Johannes Henkel helped me gather the traces for Chapter 3. Hal Gabow helped me come up with the optimal chooser algorithm in Chapter 7. Matthew Hertz's expertise in garbage collection traces improved the methodology for Chapters 3 and 9. I have been fortunate to have had great collaborators on this work, and am honored to have co-authored papers with them.

The students in Amer's research group improved this dissertation by their insightful and patient feedback on paper drafts, practice talks, and group meeting discussions. All of them (Matthias Hauswirth, Johannes Henkel, Han Lee, Jeff Palm, Christoph Reichenbach, and Daniel von Dincklage) made the work environment pleasant, and were also great companions for the occasional hike, movie, party, or game.

Warm thanks go to the members of my thesis committee. In particular, I am thankful for the many great discussions I had with Bill Waite; he is an experienced and passionate teacher. I learned a lot from attending his classes, and from having him as a mentor when I taught classes myself. I also thank Dan Connors, Hal Gabow, and Dirk Grunwald for their interest in my work.

The research in this dissertation would not have been possible without software artifacts that generous researchers contributed to the community. The group at IBM Research who developed Jikes RVM had a huge impact on dynamic runtime systems for object-oriented programming languages. Steve Blackburn and Perry Cheng developed JMTk, a memory management toolkit that provided the basis for numerous garbage collection research projects. This dissertation uses both Jikes RVM and JMTk. In addition, the Spark pointer analysis framework from McGill University, and the garbage collection simulator from the University of Massachusetts at Amherst, inspired my design of the pointer analysis and the garbage collector simulator for this dissertation.

I am thankful to the entities who funded the research for this dissertation. Most of the funding came from my advisor, Amer Diwan. The German Academic Exchange Service and the German National Scholarship Foundation enabled me to come to Boulder in the first place. I warmly thank IBM Research for the support and the honor of granting me a Ph.D. fellowship. And I am grateful to the University of Colorado, to ACM SIGPLAN, and to Kathryn McKinley for financial support to attend conferences and professional meetings.

Many thanks go to the researchers at IBM T.J. Watson Research Center, for fruitful discussions of the ideas in this dissertation. I had a lot of fun, and I learned a lot, during my summer internship there! Thanks go to the members of the DaCapo group, in particular to Sam Guyer, Steve Blackburn, Eliot Moss, and Kathryn McKinley. The DaCapo group was the premier research group for the topic garbage collection during the time when I worked on my dissertation, and it was a awesome to have these bright people as friends.

Many people helped me with this dissertation. If you are one of them, and I forgot to mention you in these acknowledgments, please accept my sincere apology. I am grateful for your help!

I thank my parents, my sister Heidi, and my brother Peter for being a wonderful family, and I am grateful for their support despite my choice to spend the last five years on the other side of the Atlantic.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation introduces connectivity-based garbage collection. Garbage collection is the automatic reclamation of heap objects that the program does not need anymore. Connectivity is the way in which objects are connected by pointers. Connectivity-based garbage collection obtains connectivity information from an analysis of the program code, and uses it to achieve high throughput, good memory efficiency, and smooth responsiveness.

This dissertation is based on the thesis that:

1. Objects are part of distinct connected data structures.
2. Connected objects tend to die at the same time.
3. Garbage collectors can exploit properties 1. and 2. to reclaim objects efficiently.

1.1 Benefits of automatic memory management

Modern programming languages rely heavily on dynamic heap-allocated objects. Objects can form lists, arguably the most prevalent data structure of programs written in functional languages. Likewise, objects can represent instances of classes implementing abstract data types, which are the key feature of object-oriented languages. For example, a hash-table in a language like Java is usually represented by an object for the hash array, and several objects forming lists that store the individual key/value pairs, which are usually also represented by objects.

Most programs continuously allocate more and more objects, requiring more and more heap memory; unless they recycle memory by freeing unused objects, they eventually run out of memory. In the example of the hash table, adding a new entry entails allocating a new object to store the key/value pair, and may also lead to allocating a larger hash array if the hash table grows above a predefined threshold. On the other hand, programs often use a data structure for a while, but then enter a new phase where they do not need it anymore and drop pointers to it. When that happens, the memory of the dead data structure should be reused to avoid eventually running out of memory.

In some languages, the programmer is responsible for freeing up memory of dead objects so it can be reused. For example, in C, *malloc*() dynamically allocates a heap object, and *free*() reclaims its memory again. The allocated object may be accessed via pointers, which may be stored in global variables, passed as parameters, returned as return values, or stored in other objects. Without great care and discipline, it is hard to keep track of all the places in the program that have a pointer to an object and may thus use it.

Manual deallocation, where the programmer explicitly frees up memory of objects, is difficult, because it is hard to tell when an object is not used anymore. This is because object lifetime is inherently a global property: objects are usually shared by many parts of the code. There are two possible kinds of mistakes in manual deallocation: freeing an object too early, or freeing an object too late.

Freeing an object too early leads too memory corruption. Freeing an object too early means that it is freed even though it may be accessed in the future. If the memory of the freed object gets used for another allocation, then there will be other code that uses its memory to store unrelated data. Since both parts of the code are oblivious of each other, and use the same memory to store unrelated pieces of data, memory corruption ensues. In the best case, testing reveals the memory corruption, and the bug is fixed. But it can easily happen that the memory corruption goes unnoticed, or it is detected, but the root cause (too early reclamation) is not tracked down.

Freeing an object too late leads to memory exhaustion. This situation is called a leak. The analogy is that of a ship slowly taking on water; eventually, there is not enough air in its hull anymore to keep it afloat, and the ship sinks. In the case of dynamic allocation, the program is slowly accumulating dead objects; eventually, there is not enough free memory anymore to satisfy allocation requests, and the program crashes. If the leak is detected on small test runs, the bug may be fixed. But the more insidious leaks act slowly, and only lead to a crash after the software has been deployed for a long time. This is analogous to a ship staying afloat while in the safe harbor, but sinking while trying to cross the ocean.

Even in the absence of either kind of bug in manual deallocation, manual deallocation hurts code quality. To keep track of the global property of which objects are used from where, developers have to keep the memory management concern in mind in their entire design. Steering clear of too early or too late reclamation significantly increases the cost of software development: the design takes more work, there is more to implement, and the product requires more thorough testing and more time-intensive debugging.

In addition to increasing development cost, manual deallocation increases maintenance cost. The design of each component of the software considers the global concern of keeping track of when exactly objects have to be freed. Whenever anyone tries to fix a bug or add a feature in one component, they have to understand how that affects manual deallocation, or risk introducing bugs due to too early or too late reclamation. The design is likely to help the maintainers in this task by adding explicit protocols; but this increases the complexity of the software, and makes it harder to understand what else the code does. Often, this inhibits software

reuse: even though a piece of code already exists for performing a particular task, it is more work to find it among the memory management code, and to make sure a reuse does not violate memory management policies, than it is to just write it from scratch.

Fortunately, one can avoid the software engineering costs of manual deallocation with automatic memory management, and most modern languages do so. By now, the benefits of automatic memory management are so widely recognized that it is mandated by the definition of Sun's Java language, it is an integral part of Microsoft's Common Language Runtime and thus available to languages such as C#, and it is also used for wide-spread scripting languages such as Perl and Python. Even for languages with explicit deallocation such as C or C++, many programmers prefer to use garbage collectors provided in the form of libraries. A popular example for this is the Boehm-Demers-Weiser collector [23].

1.2 Benefits of tracing garbage collection

The previous section argued for automatic memory management; this section argues for one specific approach to automatic memory management, namely tracing garbage collection. Figure 1.1 shows a tree of approaches to memory management.

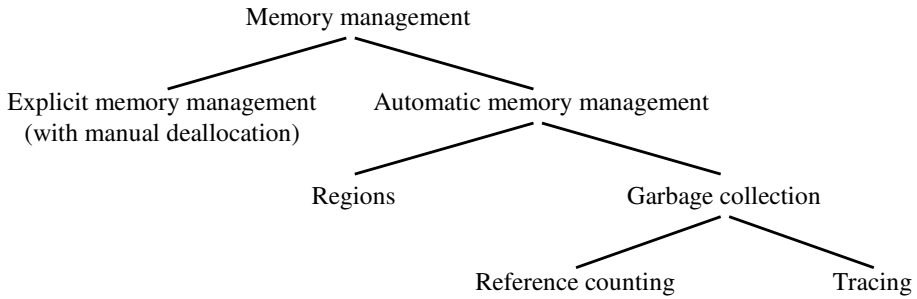


Figure 1.1: Approaches to memory management.

Memory management is either explicit or automatic. As discussed previously, automatic memory management has a variety of software engineering benefits that make it the approach of choice for most modern languages. All of the approaches to automatic memory management shown in Figure 1.1 prevent the problem of too early reclamation leading to data corruption, and all of them reduce the problem of too late reclamation leading to leaks and crashes. They differ fundamentally in how successful they are at preventing leaks. No automatic memory management schemes can fully prevent leaks, since that would require perfect liveness information for variables [80].

One approach to automatic memory management are regions. A region is a memory area that contains objects. No individual objects in a region ever get freed, neither explicitly nor automatically. Instead, the entire region is freed en-masse

when all objects in it are guaranteed to be dead. While some systems require explicit freeing of regions by the programmer [55], the region deallocation time is more commonly determined by a static program analysis [129]. Some implementations of functional programming languages rely on regions for automatic memory management [128].

Garbage collection is more popular than regions due to some problems with regions. The main problem with regions is that since an object is only freed after all objects in its region are guaranteed to be dead, it is often reclaimed very late. This happens whenever object lifetime distributions are not amenable to regions, i.e., do not resemble a stack discipline. One solution is to use profiling to identify this problem, and then manually change the code of the program to make object lifetime distributions fit the region model better. However, that requires extra effort on the part of the developer not needed with garbage collection. Just like with explicit memory management, requiring the developers to hand-tune their applications to work well with regions introduces software engineering costs in design, testing, and maintenance.

The two main approaches to garbage collection are reference counting and tracing (see Figure 1.1). Both collect garbage objects based on their connectivity to roots. A root is a stack variable or global variable storing a pointer to a heap object. An object not connected to roots by pointers is dead, it can be reclaimed as garbage. Figure 1.2 shows an example heap.

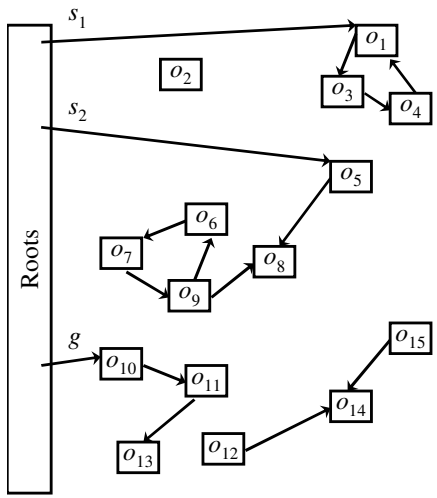


Figure 1.2: Example heap. Arrows (→) represent pointers, small boxes ($\boxed{o_i}$) represent objects. The long box on the left represents the roots, including two stack variables s_1 and s_2 with pointers to objects o_1 and o_5, and one global variable g with a pointer to object o_{10}.

Reference counting garbage collection maintains a counter with each object that counts how many references (i.e., pointers) there are to it [40]. Writing a pointer to

the object increments the counter, and deleting a pointer to the object decrements the counter. When the counter drops to zero, the object is dead and will not be used anymore, so it is freed. In addition, the reference counters of all successor objects are decremented.

The problem with pure reference counting garbage collection is that it can not reclaim cycles. For example, in Figure 1.2, objects o_1, o_3, and o_4 form a cycle. The reference count of o_1 is 2, since both s_1 and o_4 point to it; the reference counts of o_3 and o_4 are 1. If s_1 is deleted, the reference count of o_1 drops to 1. All three objects o_1, o_3, and o_4 are dead, since they can not be accessed anymore from the roots. But since they all still have a reference count of 1, the garbage collector does not reclaim them, and they constitute a leak.

Since pure reference counting garbage collectors can not deal with cycles, there are hybrids that perform tracing to reclaim cyclic garbage [13].

Tracing garbage collectors work in two phases: the first phase is a graph traversal that finds all objects reachable by following pointers from the roots, and the second phase is the reclamation of all objects that the traversal did not reach. For example, in Figure 1.2, the reachability traversal would find objects $o_1, o_3, o_4, o_5, o_8, o_{10}, o_{11}, o_{13}$. The remaining objects are garbage, and the reclamation phase frees their memory for future allocation.

Tracing garbage collection reclaims all objects that are not reachable from roots by following pointers. It has fewer leaks than any of the other approaches to automatic memory management in Figure 1.1. There may still be more subtle leaks, where an object should be freed even though it is still reachable from pointers. This may happen because non-pointers are misinterpreted as pointers due to lack of type accuracy [76, 77]. All garbage collectors considered in this dissertation are type accurate, but the approaches it describes generalize to type-inaccurate collectors [61]. Another reason for leaks in tracing garbage collection is lack of liveness accuracy [77, 80]. Since there are no feasible approaches for obtaining precise liveness accuracy to date, this dissertation only considers liveness inaccurate collectors.

1.3 Garbage collection wish-list

Garbage collectors should provide high application throughput (low cost in time), high memory efficiency (low cost in space), and high responsiveness (few disruptions).

1.3.1 Throughput

Throughput is the speed at which a program gets its work done. Given a fixed amount of work, high throughput means that the program gets it done in a short time. Throughput is important for saving users money and patience.

Garbage collection affects program throughput in many ways. The most obvious way is that performing a tracing garbage collection costs time, and the less time it costs, the higher the program throughput gets. But garbage collection also performs other tasks throughout the program execution. Those include the cost of

allocation, as well as costs for book-keeping. All other things being equal, more efficient allocation and book-keeping lead to higher program throughput.

A more subtle, but equally important, way in which garbage collection affects throughput is by its effects on locality. If the program accesses some objects close together in time, the garbage collector should keep them close together in space, to reduce the likelihood of cache misses, TLB misses, or even page faults. All other things being equal, a garbage collector that creates good program locality leads to high program throughput.

1.3.2 Space efficiency

Space efficiency is the smallness of the memory that a program runs in. Memory is always limited, sometimes severely, for example in small mobile devices, sometimes less, for example when running a small program on a big desktop machine. Space efficiency is important to make it possible to run the required jobs on a given machine without buying more memory or killing other jobs.

Garbage collectors affect space efficiency in many ways. One of them is that garbage collectors introduce a cost in space by storing their own auxiliary data structures, some in object headers, some on the side. Another important way in which garbage collection affects space efficiency is by fragmentation, where free memory is available, but can not be used for allocating objects. An approach to limiting fragmentation is copying garbage collection, but that requires reserving space to copy objects into, which itself reduces space efficiency.

There is a trade-off between throughput and space efficiency. On the one hand, if the program has a small heap (high space efficiency), the garbage collector runs often and takes a lot of time (low throughput). On the other hand, if the program has a large heap (low space efficiency), the garbage collector runs rarely and takes little time (high throughput). Of course, in a large heap, locality may suffer, so providing huge amounts of memory does not necessarily lead to high throughput.

This dissertation deals with the trade-off between throughput and space efficiency by reporting throughput for a few fixed heap sizes. This reflects the situation in practice where the heap size is determined by how much memory is available, and the user is interested in what throughput the program has at that heap size.

1.3.3 Responsiveness

Responsiveness is the absence of disruptive periods where a program responds sluggishly or not at all. It is important for interactive applications such as media replay, word processing, or web-based forms. It is even more important for soft or hard real-time applications, where sluggishness may have catastrophic consequences.

The main impact of garbage collection on responsiveness stems from garbage collection pauses. Pauses should be short, to avoid disruptive periods where the program responds not all; but when there are many short pauses close to each other, this also leads to poor responsiveness, where the program responds, but only sluggishly. Other ways in which garbage collection affects responsiveness include

the time cost of performing allocations or book-keeping; but those disruptions are so small compared to garbage collection pauses that they are usually not considered for evaluating responsiveness.

There is a trade-off between responsiveness and throughput in that some garbage collection algorithms focus mainly on high responsiveness, at the cost of lower throughput, and others focus mainly on high throughput, at the cost of lower responsiveness. This dissertation mainly concentrates on garbage collection that yields high throughput. The responsiveness goal is to perform well for interactive applications, but this dissertation does not consider real-time constraints.

While the primary trade-off between responsiveness and throughput arises from the choice of garbage collection algorithm, some garbage collectors allow a secondary trade-off in the form of tuning parameters (e.g., [17]). Connectivity-based garbage collectors provide this flexibility as well.

1.4 The state of the art

This dissertation is concerned with tracing garbage collectors (Section 1.2) with high throughput, high memory efficiency, and responsiveness that is high enough for interactive applications, but not necessarily for real-time systems. The state of the art for achieving high throughput with reasonable memory efficiency and responsiveness is generational garbage collection. For example, typical numbers for a generational collector are spending 14% of the execution time on memory management tasks when provided with space that is 3 times the high watermark of the program, and experiencing pauses of up to 0.8 seconds [19].

The key idea of generational garbage collection is to segregate heap objects by age into generations, and collecting young generations more often than old generations [94, 130]. Figure 1.3 shows a heap where the young objects occupy a young generation, and the old objects an old generation.

The weak generational hypothesis states that most objects die young [67]. It is true for many programs, and means that the survivor rate in the young generation is lower than in the old generation: the young generation contains proportionally fewer live objects and more dead objects.

Generational garbage collection increases throughput. This is because garbage collections opportunistically focus on the area of memory where it is most likely to find dead objects to reclaim. Opportunistic garbage collections reclaim the same amount of memory with a lower cost in time. Thus, they incur less overall overhead on the execution time.

Generational garbage collection can trade some of the increased throughput for increasing memory efficiency instead. With the increased throughput from opportunism, generational garbage collection can afford to run in a smaller heap and still perform as well as it would in a larger heap without the increased throughput.

Generational garbage collection increases responsiveness. This is because most of the collections collect only a part of the heap. Partial garbage collections take less time than full garbage collections. Thus, they incur shorter pauses.

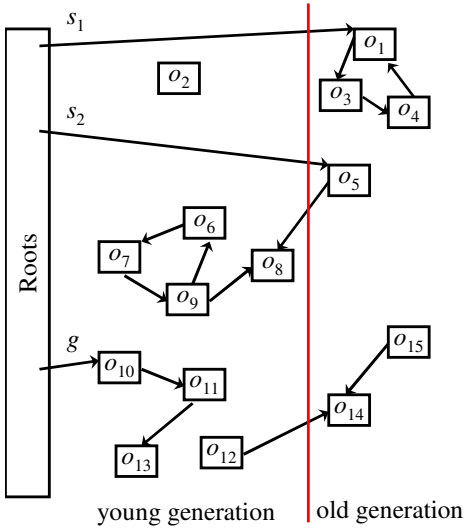


Figure 1.3: The heap from Figure 1.2, with generations.

Generational garbage collection is widely recognized as the state of the art. It is implemented in the Java virtual machines by Sun and by IBM, it is used by Microsoft's implementation of the CLR, and it is implemented in SML/NJ. There are plenty of variations on generational garbage collection, ranging from add-ons such as pretenuring [34], over algorithmic variations such as using mark-sweep or copying or reference counting for the old generation [19], to generalizations of age-based garbage collection [21].

1.5 Why do something else

The state of the art generational collectors yield high throughput with reasonable memory efficiency and responsiveness, but this dissertation proposes a radically different approach.

The wish-list from Section 1.3 is challenging to achieve, and a large part of today's garbage collection literature describes how to improve upon the performance of generational garbage collection along one or more dimensions while remaining within the general context of age-based garbage collection. This dissertation, on the other hand, takes the standpoint that it is worth exploring a fundamentally different approach that is not age-based, but instead, is connectivity-based. The goal is to deliver throughput, space efficiency, and responsiveness that are competitive with generational garbage collection.

Generational garbage collection has some problems that hinder its ability to deliver high throughput. One problem is posed by old-to-young pointers. For example, in Figure 1.3, the old object o_5 points to the young object o_8. In order to collect only the young generation, the collector must be aware of the pointer from o_5 with-

out performing a reachability traversal of the old generation. This is usually done with a write barrier that traps old-to-young pointer writes, and records them in a remembered set. The problem is that this book-keeping adds both time and space overhead. In addition, even if the pointer s_2 to the old object o_5 is deleted, since o_5 is still in the remembered set, the collector will not free up o_8 yet, because it assumes that it might still be reachable. Since less memory is reclaimed, the collector will have to run more often, and throughput suffers.

Another impediment to high throughput in a generational garbage collector is the so-called "pig in the python" problem. The analogy is that of a python that swallows something too big, and is immobilized until it has digested it. In the case of the generational collector, "something too big" is a large data structure that lives long. When it is first allocated, it is still young, and the collector tries in vain to collect it: since it lives long, it survives that collection, and the opportunistic choice of collecting the young generation in the hope for many dead objects went wrong. Since the collector has to pay a high cost trying to collect the young generation, throughput suffers.

Generational garbage collection also has a problem that hinders its ability to deliver high responsiveness. While most collections collect only the young generation and thus incur only a short pause, every once in a while, a generational collector performs a full collection that incurs a long pause. Regular full collections are necessary in generational collectors to ensure that all garbage eventually gets collected.

The fundamental idea used by generational garbage collection that can be extended to non-age-based collectors is to perform partial opportunistic collections. Section 1.4 described how generational garbage collection does that based on age. Partial collections increase responsiveness, whereas opportunistic collections increase throughput. The basis for opportunism in generational collectors is age, and that also dictates the basis for partial collections. In other words, generational collectors emphasize opportunistic collections for high throughput, and as a side-effect, achieve partial collections for high responsiveness.

This dissertation proposes using connectivity to perform partial opportunistic collections. A connectivity-based garbage collector uses connectivity to support partial collections. This avoids the problems of old-to-young pointers and regular full collections that reduce the throughput and responsiveness of age-based partial collections. The partial collections based on connectivity then give flexibility for an opportunistic choice of what to collect. As it turns out, connectivity directly helps the opportunistic choice, but in addition, age-based opportunism is also applicable. In other words, connectivity-based collectors emphasize partial collections for high responsiveness, but the partial collections are more flexible than in age-based collectors, with more flexibility for opportunism that yields high throughput.

Preliminary experiments support the claim that connectivity helps opportunism. Three sets of results that indicate that one can use connectivity information to predict which partial collections are the most likely to reclaim dead objects: connected objects tend to die at the same time; short-lived objects tend to have few ancestors; and objects reachable from global variables tend to live long, whereas objects pointed to only from stack variables, but not from globals or other objects, tend to

die quickly [82].

While a lot of research has focused on age-based garbage collection, this dissertation explores an alternative that has received little attention. Whereas object age has been proven useful for opportunistic collections, using it to perform partial collections introduces difficulties such as old-to-young pointers and regular full collections. Connectivity is a natural principle to base partial collections on, and allows opportunistic collections that exploit connectivity as well as age and other predictors for which objects are dead.

1.6 How connectivity-based garbage collection works

Section 1.5 motivated that connectivity is a sensible principle to base a garbage collector on; this section describes a framework for connectivity-based garbage collectors. The goal is to use connectivity to do partial collections (increasing responsiveness), while at the same time providing a lot of flexibility for opportunism (increasing throughput), some of which is itself also based on connectivity.

The equivalent of an age-based collector's generation is a connectivity-based collector's partition. Figure 1.4 shows an example heap with partitions. Partition edges respect pointers: if there is a pointer between two objects, they are either in the same partition, or there is an edge between their partitions. A partial garbage collection can reclaim any subset of partitions that is closed under the predecessor relationship (for each partition in the set, its predecessors are also in the set).

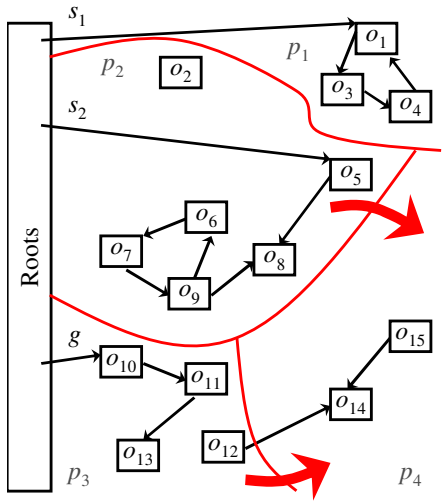


Figure 1.4: The heap from Figure 1.2, with partitions. Each of p_1, p_2, p_3, and p_4 is a partition. Large bent arrows represent partition edges.

The framework for connectivity-based garbage collectors has four components:

1. Partitioning. The partitioning is based on a static program analysis, and decides into which partition each object is is allocated.

2. Estimator. The estimator runs before a garbage collection, and guesses how many dead and live objects each partition contains.

3. Chooser. The chooser picks which set of partitions to collect based on the results of the estimator.

4. Partial garbage collection. The partial garbage collection performs a reachability traversal on all objects in the chosen partitions, and reclaims unreachable objects in that part of the heap.

The goal of the partitioning is to provide fine-grained partitions where partition edges respect object pointers, and objects do not move between partitions. It uses an analysis of the program code to find conservative connectivity information, which says how objects may be connected at run-time. Based on that information, each object is placed into a partition upon allocation. The representation of partitions at run-time must support garbage collections by providing a mapping from objects to partitions, and by guaranteeing the absence of cross-partition pointers unless there is also a partition edge between the partitions in question. For example, in Figure 1.4, the pointer from object o_{12} to o_{14} spans a partition boundary, necessitating the partition edge from p_3 to p_4.

The goal of the estimator is to guess, for each partition, how many objects in the partition are dead and how many are live. It is based on a variety of heuristics for how information available at run-time predicts partition survivor rates. For example, one of the heuristics derived from observations from an experimental study of run-time behavior [82] says that a partition reachable from global variables has a high survivor rate. To use this heuristic, the estimator looks at all global variables, and estimates that partitions containing objects pointed to by globals and their successor partitions contain many survivors. For example, in Figure 1.4, global g points to object o_{10} in partition p_3, so this heuristic would estimate many survivors in p_3 and its successor p_4.

The goal of the chooser is to opportunistically pick a set of partitions to collect for which the estimator predicted many dead and few live objects. Dead objects are objects whose memory can be reclaimed, and thus represent the pay-off of a garbage collection, since that memory becomes available for future allocation. Live objects are objects that the reachability traversal has to visit to find which objects are dead, and thus represent the cost of a garbage collection, since the reachability traversal takes time. The result of the chooser must be a set of partitions that is closed under the predecessor relationship: if the choice contains a partition, it must also contain all its predecessors that have partition edges to it. For example, in Figure 1.4, if the chooser picks p_4, it must also pick its predecessors p_2 and p_3.

The goal of the partial garbage collector is to reclaim all dead objects in the chosen set of partitions. To do this, it traverses all live objects in those partitions, and reclaims the rest. When doing this, it does not need to worry about pointers from

objects in non-chosen partitions, since the partitioning and the chooser guarantee that there are no pointers from unchosen partitions to chosen partitions. Partial collections of connectivity-based collectors provide early reclamation: even before the collection of all chosen partition completes, the memory of dead objects in a partition can be reclaimed as soon as all live objects in it and its predecessors have been traversed. For example, in Figure 1.4, if the set of chosen partitions is $\{p_2, p_3, p_4\}$, then dead objects in p_2 can already be reclaimed before all live objects in p_3 and p_4 have been traversed.

Connectivity-based garbage collection increases throughput. This is because garbage collections opportunistically focus on the area of memory where it is most likely to find dead objects to reclaim. In addition, the problem of old-to-young pointers that reduced generational garbage collector throughput is absent in connectivity-based garbage collectors. On the other hand, if the static analysis for obtaining connectivity information is performed at execution time in a just-in-time compiler, that incurs its own throughput cost.

Connectivity-based garbage collection can trade some of the increased throughput for increasing memory efficiency instead. In addition, the early reclamation of its partial collections help a connectivity-based garbage collector to stay within a smaller memory range rather than walking all over virtual memory, which increases both memory efficiency and throughput.

Connectivity-based garbage collection increase responsiveness. This is because except in pathological cases, all garbage collections are partial, incurring only a short pause. Whereas generational collectors have to perform full collections regularly, connectivity-based collectors only have to perform full collections when there is a partition that contains dead objects and all other partitions are direct or transitive predecessors of it. We never observed this pathological case in practice.

1.7 Contributions

This dissertation makes three contributions to programming language implementation research. It reports empirical studies of program behavior that improve understanding of heap object connectivity; it introduces and evaluates the new family of connectivity-based garbage collectors; and it presents the first non-trivial pointer analysis that works for all of Java.

Understanding the connectivity of heap objects is also useful for improving existing collectors. We originally reported the results of our empirical study in an ISMM paper [82]. It provided the starting point of this dissertation. The results have also been picked up by other people; for example, Guyer and McKinley [63] present a technique for exploiting our finding that connected objects die together in a generational garbage collector.

Connectivity-based garbage collection is partial opportunistic garbage collection based on heap object connectivity. We originally presented this new family of garbage collectors in an OOPSLA paper [78]. It provides the center-piece of this dissertation. Whereas [78] evaluates the design space of connectivity-based garbage

collection using a simulator, this dissertation also includes a description of a concrete connectivity-based garbage collector in a Java virtual machine, proving that the concept works in the presence of all real-world challenges posed by a full-fledged system. Finally, this dissertation also includes an optimal algorithm for the chooser, along with a correctness proof and an evaluation, which we originally published in a technical report [81]. The ideas from our OOPSLA paper have also been picked up by other people; for example, Grothoff [61] describes how a theoretical framework helps extend it for conservative garbage collection.

Connectivity-based garbage collection for Java requires a non-trivial pointer analysis that works for all of Java. We present the first such analysis in an ECOOP paper [79]. It solves an important challenge posed by performing connectivity-based garbage collection in a full-fledged system. The ECOOP conference took place within one month of the defense of the thesis of this dissertation, and the community has not yet reacted with follow-up work; but our pointer analysis has already been cited by concurrent related work by Qian and Hendren on interprocedural program analysis [103].

This dissertation revolves around the thesis stated on page 13, repeated here:

1. Objects are part of distinct connected data structures.

2. Connected objects tend to die at the same time.

3. Garbage collectors can exploit properties 1. and 2. to reclaim objects efficiently.

Connectivity-based garbage collectors operate on many partitions, each containing objects forming a distinct connected data structure, supporting Part 1. of the thesis. This dissertation reports on studies measuring the likelihood that connected objects die at the same time, which found that it is much higher than for unconnected objects, supporting Part 2. of the thesis. This dissertation describes a new family of garbage collectors, and reports on experiments both with a simulator and in a full-fledged system that show that these collectors exploit Parts 1. and 2. of the thesis to reclaim objects efficiently, supporting Part 3. of the thesis.

1.7.1 Overview of the remaining chapters

Appendix A defines terminology used by the remainder of this dissertation. When a chapter refers to a definition in this appendix, it uses the notation $\text{term}^{\text{p.}n}$, where n is the number of the page with the definition for the term.

Chapter 2 describes the infrastructure for experimental work reported in this dissertation. All experiments in this dissertation rely on the Jikes RVM virtual machine for Java either for providing traces, or as a real-world context in which to implement our algorithms.

Chapter 3 reports the results of experiments for understanding the connectivity of heap objects. This dissertation describes garbage collectors based on connectivity, which make opportunistic choices based on the results of this chapter.

Chapter 4 gives an overview of the CBGC family of connectivity-based garbage collectors. It provides the big picture of how these collectors work, and sets the stage for the subsequent chapters that flesh out the details.

Chapters 5 to 8 elaborate on the four components of a CBGC algorithm:

- Chapter 5 gives the details for how the partitioning component of a connectivity-based garbage collector works, and describes a few possible solutions to this sub-problem.

- Chapter 6 gives the details for how the estimator component of a connectivity-based garbage collector works, and also shows how various heuristics can be applied to solve this sub-problem.

- Chapter 7 gives the details for how the chooser component of a connectivity-based garbage collector works. It provides a problem statement and an algorithm that is optimal with respect to that problem statement. It also describes alternative algorithms.

- Chapter 8 gives the details for how the partial garbage collection component of a connectivity-based garbage collector works, and also describes how various well-known garbage collectors from the literature can be generalized as solutions to to this sub-problem.

Chapter 9 reports the results of experiments for exploring the design space of connectivity-based garbage collectors. It is based on a trace-driven simulator, and explores both realistic and oracular points in the design space to give indications for lower and upper bounds on how well CBGC can perform.

Chapter 10 describes the first non-trivial pointer analysis that works for all of Java. While many important client tools and optimizations rely on pointer analysis, in the context of this dissertation, it serves as a foundation for the partitioning component of connectivity-based garbage collection.

Chapter 11 describes a connectivity-based garbage collector in Jikes RVM, demonstrating that the concept works in the presence of all real-world challenges posed by a full-fledged system.

Chapter 12 concludes the dissertation.

Chapter 2

Infrastructure

This dissertation describes several experiments performed in Jikes RVM on a number of Java benchmarks, either directly or in the form of traces. This chapter describes Jikes RVM (Section 2.1), the benchmarks (Section 2.2), and the traces (Section 2.3).

2.1 Jikes RVM

Jikes RVM is an open-source research virtual machine for Java from IBM Research [5, 29]. This section describes details of Jikes RVM that are important for understanding this dissertation.

2.1.1 Jikes RVM compilers

Jikes RVM, like most Java virtual machines, includes JIT compilers[p.176] that compile portable Java bytecode to executable machine code for Power PC or Intel x86 processors. Understanding some details of the Jikes RVM compilers is important because this dissertation describes experiments based on traces created by using compilers to instrument code (Chapter 3), because the compilers allocate many objects that have to be garbage collected (Chapter 9), and because this dissertation describes a novel compiler analysis (Chapter 10).

Unlike most Java virtual machines, Jikes RVM does not include an interpreter, compiling all methods before their first execution instead. Jikes RVM has two JIT compilers: a baseline compiler and an optimizing compiler. Roughly speaking, the baseline compiler runs fast, but the code it produces runs slowly, whereas the optimizing compiler runs slowly, but the code it produces runs fast. In addition to these two compilers, Jikes RVM also includes an adaptive optimization system that baseline-compiles most methods and reserves the expensive optimizing compiler for a few hot methods only.

Either compiler must support the garbage collector in a number of ways. The compiler is responsible for providing type accuracy[p.182], by maintaining type information through all compiler passes, storing it for lookup in so-called stack maps, and supporting stack walks at garbage collection time to identify all pointers in all method stack frames, among other things. All garbage collectors in Jikes RVM are

type accurate. For collectors that require write barriers[p.189], the compiler is also responsible for instrumenting compiled code with the instructions implementing the barrier.

The baseline compiler simply replaces each individual bytecode instruction, one by one, by an equivalent sequence of machine instructions. In doing so, it does not perform any optimizations, and it does not even maintain an abstract model of the stack, relying instead on the consistency of the bytecodes.

The optimizing compiler first translates Java bytecode into a register-based intermediate representation, then makes several analysis and optimization passes over that representation, and finally assembles machine instructions for it. The passes are called "compiler phases", and include all the traditional optimizations, as well as several novel optimizations for object-oriented languages pioneered by the developers of Jikes RVM. The optimizing compiler has different optimization levels, where a higher optimization level includes more expensive passes, in the hope of yielding faster compiled code.

The adaptive optimization system first compiles all methods with the baseline compiler. When a method executes often, it is considered hot, and thus worthy of recompilation with the optimizing compiler. Depending on how hot it is, the adaptive optimization system requests different optimization levels from the optimizing compiler. It may also recompile a method that has already been opt-compiled in the past if it has reason to believe that that may make the compiled code even faster.

2.1.2 Jikes RVM is written in Java

Almost all of Jikes RVM is written in Java. This means that the different components of Jikes RVM rely on each other more strongly than in other virtual machines: for example, the compilers compile the code of the garbage collectors, and the garbage collectors collect dead objects allocated by the compilers.

The self-hosting of Jikes RVM creates both challenges and opportunities. One of the challenges is that the different components are under more stress, since they have to service not only the application, but also the virtual machine itself. The self-hosting puts higher demands on components both in terms of performance (the amount of work is higher) and correctness (the work is more critical and more challenging). One of the opportunities is that all VM components benefit from each other's improvements; for example, the performance of the allocation sequence of the garbage collector benefits from the compiler's ability to inline and optimize it. The main advantage of writing Jikes RVM in Java, however, is that Java is a high-level language that yields high programmer productivity.

Each Jikes RVM process starts from a boot image of pre-compiled Java code and pre-allocated Java data structures. To create this boot image, a host JVM executes a Java program, the boot image writer. The boot image writer executes the code of the Jikes RVM compilers to compile the code of various Jikes RVM components, including the compilers themselves, to machine code. The boot image writer also executes the initialization code for various Jikes RVM classes to pre-allocate Java objects, such as auxiliary data structures for the garbage collectors. Finally, the

boot image writer inspects all those Java objects using reflection, and serializes the raw data in the object format of a running Jikes RVM process. This yields a boot image, a file containing the compiled code and Java objects needed to start executing Jikes RVM. The boot image runner, a tiny C program, loads this boot image into memory, and jumps to the address of the instructions responsible for starting the VM, which were compiled from Java at build time.

Not all of Jikes RVM is written in Java, some of it is written in C or even assembler code. The previous paragraph already mentioned the boot image runner, a tiny C program responsible for starting Jikes RVM. In addition, Jikes RVM can call "magic" methods that look like Java methods, but are recognized by the compilers and translated into lower-level operations, e.g., for raw pointer manipulation. Magic allows for unsafe, low-level, system code that it necessary for implementing a virtual machine, but usually not available in Java. Another part of Jikes RVM that is not written in Java is the interface to the operating system. Jikes RVM includes system call wrappers written in C, called from Java via a special mechanism. A mechanism separate from system calls is JNI[p.176], the usual mechanism for Java code to interact with C code. Applications running on top of Jikes RVM can use JNI to call C code just like in other virtual machines, and the C code can call back into Jikes RVM service methods to manipulate Java-level fields, call Java methods, or allocate Java objects.

The fact that so much of it is written in Java means that in Jikes RVM, every benchmark is large: executing even the simplest "Hello world" program requires executing Jikes RVM itself, a substantial Java program. The FastAdaptiveMarkSweep configuration of Jikes RVM version 2.2.1 consists of 982 classes with a total class file size of 4,559 KB. In addition, both Jikes RVM and the application use standard libraries, which add up to another 2080 classes with a total class file size of 3,543 KB. This is much more Java code than, for example, the code of the benchmark javac (1,909 KB), which is itself one of the largest benchmarks in common use in Java memory management research. The size of the Java code of Jikes RVM and the libraries impacts the results of experiments with Jikes RVM.

2.1.3 Jikes RVM garbage collectors

A Jikes RVM boot image can include one of several garbage collectors. Understanding some details of the Jikes RVM collectors is important because this dissertation relies on them for tracing (Chapter 9) and validation (Chapter 10), and because this dissertation describes a new garbage collector in Jikes RVM, which relies on some support from the existing code base, but also competes against other collectors in that code base (Chapter 11).

The garbage collectors in Jikes RVM evolved considerably at the same time that the research in this dissertation was in progress. Table 2.1 gives a time-line. The time lag between the release of a Jikes RVM version and the paper that experiments with it illustrates the inertia of adopting a new version, as well as the time it takes to understand the new infrastructure, add own code, perform experiments, write a paper, and wait for it to appear in a conference.

Table 2.1: Jikes RVM and CBGC evolution.

April 2001	Jalapeño 1.1 released. Only limited Linux/IA-32 support so far.
October 2001	Jalapeño renamed into Jikes RVM. Jikes RVM 2.0.0 released. It now uses the classpath libraries.
June 2002	The paper underlying Chapter 3 appears in ISMM [82]. It uses Jalapeño 1.1.
December 2002	Jikes RVM 2.2.0 released. It now includes the JMTk.
April 2003	Jikes RVM 2.2.1 released.
June 2003	Jikes RVM 2.2.2 released.
October 2003	The paper underlying Chapter 9 appears in OOPSLA [78]. It uses Jikes RVM 2.2.0.
June 2004	The paper underlying Chapter 10 appears in ECOOP [79]. It uses Jikes RVM 2.2.1.
July 2004	Chapter 11 is written. It uses Jikes RVM 2.2.1.

This section describes the JMTk as of Jikes RVM version 2.2.1. JMTk [20], the Java Memory management Toolkit by Steve Blackburn and Perry Cheng, replaced the old "Watson" collectors used before Jikes RVM 2.2.0.

In Jikes RVM 2.2.1, JMTk includes five garbage collectors: three full collectors[p.189] (copying[p.184], mark-sweep[p.185], and reference counting[p.178]), and two generational collectors[p.190] (both using copying for the nursery[p.190], one with copying and one with mark-sweep for the mature space[p.190]). The collectors share support code, which is general enough for reuse in different collector designs.

JMTk in Jikes RVM 2.2.1 provides exclusively parallel[p.190] stop-the-world[p.191] collectors. Regardless of collector, the collector threads share a work-queue of gray objects. That means that copying GC in JMTk does not use Cheney scan[p.184], and mark-sweep GC does not use a mark stack, contrary to Table A.1. This dissertation is only concerned with single-threaded, stop-the-world GC. In all experiments, the parallel Jikes RVM collectors run on single-processor machines, and thus have only one thread.

All JMTk collectors are hybrids that manage different kinds of objects with different basic garbage collector techniques.

- Boot image objects. These objects are allocated ahead of time by serializing host JVM objects into a file, as described in Section 2.1.2. Boot image objects are never deallocated, but they may point to runtime objects, and they may be mutated at runtime. Thus, the reachability traversal[p.186] of a collector has to traverse them in order to find which runtime objects have become unreachable, even though the reclamation phase[p.186] will not reclaim boot image objects.

Boot image objects are never moved, even in a copying collector.

- Immortal objects. These objects are allocated at runtime, but are never deallocated. Immortal objects in Jikes RVM include TIBs[p.188], stacks[p.175], and certain garbage collector data structures. Immortal objects are never moved, even in a copying collector.
- Meta objects. These include the shared work queues used to maintain gray objects, as well as remembered sets[p.189] for collectors that rely on a write barrier. They are not treated like normal Java objects in allocation, reachability traversal, or reclamation. Instead, they store raw addresses manipulated with magic, and are allocated and deallocated with explicit low-level memory management. Any memory manager in any runtime system has to treat meta objects as a special case, this challenge is not specific to Jikes RVM.
- Large objects. While the JMTk could treat large objects just like small objects, it does not do so for efficiency reasons. Instead, it manages large objects with a treadmill[p.186], and thus, never copies them. This saves space for copy reserve, and it saves work for large objects that do not contain any pointers.

JMTk maintains these different kinds of objects — normal small objects, boot image objects, immortal objects, meta objects, and large objects — in different areas. It segregates the virtual address space into different areas for these different objects ahead of time, so that at runtime, it can quickly determine the kind of an object by its address, and treat it accordingly in the reachability traversal. At runtime, not all pages in an area need to be in use; instead, JMTk uses mmap to acquire them lazily from the operating system. JMTk ensures that even though programs use most of the virtual address space, their usage of the physical address space remains within the heap size specified for that run, to allow the explicit tradeoff between throughput and space efficiency discussed in Section 1.3.2.

Jikes RVM uses different layouts for scalar[p.177] and array objects[p.177], illustrated in Figure 2.1.

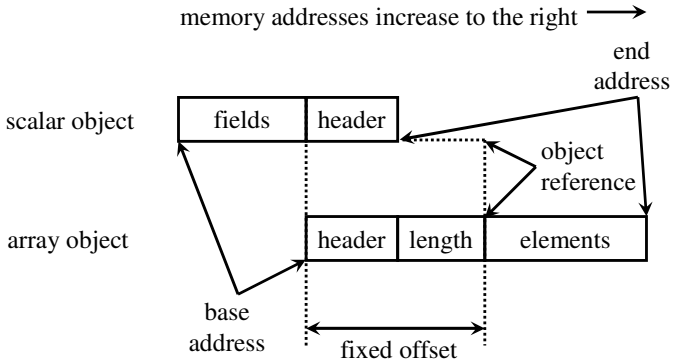


Figure 2.1: Object layout in Jikes RVM.

- A scalar object occupies the memory from its base address to its end address. It consists of fields and a header. The number and size of fields depends on the class of the object, which is stored in the header. When a variable refers to a scalar object, it does so via an object reference, which points to an address at a fixed offset behind the end address of the object.
- An array object occupies the memory from its base address to its end address. It consists of a header, a length field, and elements. The header determines the size of each element, and the length field determines the number or elements. Together, they determine the size of the object. When a variable refers to an array object, it does so via an object reference, which points to the address of the first element.

Regardless of whether an object is a scalar or an array, the header resides at a fixed negative offset from the object reference. Object headers in Jikes RVM can have different sizes depending on the garbage collector [12]. The header stores a pointer to the TIB[p.188], and a mark bit to keep track of whether the ongoing reachability traversal has reached the object.

Jikes RVM with JMTk is a good test bed for garbage collection research. Both projects are open-source, inviting people to try their ideas and contribute them back to the community. Jikes RVM includes implementations of cutting-edge Java virtual machine technology; a new idea for one component can rely on realistic support by other components, and can compete with state-of-the-art versions of the same component. These benefits of Jikes RVM and JMTk for garbage collection research are widely recognized: by June 2004, at least 26 published memory management papers used Jikes RVM.

2.1.4 Standard libraries

Starting from version 2.0.0, Jikes RVM uses the classpath open-source core Java libraries [58]. Executing Java programs require not just a virtual machine, but also a set of libraries. While Jikes RVM provides the necessary runtime support in the form of compilers, garbage collectors, multithreading, class loading, reflection, etc., the libraries are responsible for things like I/O, containers, networking, etc.

The classpath libraries are not complete. They lack some functionality, such as full support for applications with sophisticated graphical user interfaces. This restricts the set of applications that run on top of Jikes RVM: it is limited to programs using only the subset of the Java libraries included in classpath. For research with Jikes RVM, this means that not any program can serve as a benchmark, since some lack library support. This situation is the same for all open-source Java virtual machines. Jikes RVM is one of the most full-featured ones.

The classpath libraries use reflection and JNI[p.176]. For example, parts of the I/O libraries are implemented by native methods. Since most programs use these I/O libraries, most programs indirectly use JNI. Hence, at least limited JNI support is necessary to execute even simple benchmarks on Jikes RVM.

2.1.5 Versions and configurations

The research leading to this dissertation used three different versions of Jikes RVM, to take advantages of ongoing improvements in that infrastructure, see Table 2.1. Unfortunately, the results from the different versions are not directly comparable.

2.1.5.1 Configuration for understanding connectivity

Chapter 3, which is based on the ISMM 2002 paper "Understanding the Connectivity of Heap Objects" [82], uses Jikes RVM to generate traces. The traces serve as input to an analyzer that measures things like how likely it is for two objects to die at the same time if they belong to the same strongly connected component of the object graph.

This work uses Jalapeño version 1.1, running under Linux on a PowerPC machine. It uses the BaseBasenoncopyingGC configuration, meaning that the baseline compiler compiles the boot image, as well as compiling methods just-in-time at runtime; and that the garbage collector is a full mark-sweep GC.

The traces include events for the boot image, as well as the first (and only) execution of the benchmark. The death times of objects are approximated by performing frequent garbage collections and recording the earliest time that an object is found to be unreachable at a garbage collection.

2.1.5.2 Configuration for design space exploration

Chapter 9, which is mostly based on the OOPSLA 2003 paper "Connectivity-Based Garbage Collection" [78], uses Jikes RVM to generate traces. The traces serve as input to a garbage collection simulator that performs allocations and garbage collections, measuring metrics indicative of throughput, space efficiency, and responsiveness.

This work uses Jikes RVM version 2.2.0, running under Linux on an Intel x86 machine. It uses the FastAdaptiveMarkSweep configuration, meaning that the optimizing compiler compiles the boot image, whereas at runtime, the adaptive optimization system controls when methods are compiled by the baseline or optimizing compiler; and that the garbage collector is a full mark-sweep GC.

The traces include events for all objects that are reachable at the beginning of the second run (including boot image objects), as well as the second execution of the benchmark. That means that in generating the traces, Jikes RVM calls the benchmark's *main*() method twice, and omits trace output for the first execution to reduce perturbation of the results by compilation activity. Of course, inside their *main*() method, some benchmarks perform the same work multiple times; this dissertation still refers to all of this as one run. The death times of objects are precise based on the Merlin algorithm [72].

2.1.5.3 Configuration for pointer analysis

Chapter 10, which is based on the ECOOP 2004 paper "Pointer Analysis in the Presence of Dynamic Class Loading" [79], evaluates a pointer analysis that runs online within Jikes RVM. It validates the correctness by checking that all actual pointers are predicted by the pointer analysis results. It measures the performance as wall-clock time.

This work uses Jikes RVM version 2.2.1, running under Linux (with kernel version 2.4) on an Intel x86 machine (a 2.4GHz Pentium 4 with 2GB of memory). Like the design space exploration, this work uses a FastAdaptiveMarkSweep configuration, but it disables inlining during boot image build time. Inlining is, however, enabled when the optimizing compiler recompiles hot methods at runtime.

The measurements are based on executing the benchmark once. Thus, they do not include the time it takes to build the virtual machine, but they include the time for booting it and running the program's *main*() method. To enable stack walks for attributing yieldpoint counts to the application or the pointer analysis, Jikes RVM is built with deterministic yieldpoints. To enable validation of pointer analysis results, all objects have an extra header word.

2.2 Benchmarks

This dissertation uses 29 different Java programs as benchmarks to understand program behavior and to evaluate techniques for improving program performance. Table 2.2 lists these programs.

Not all experiments in this dissertation use all benchmarks. Column "used in" of Table 2.2 refers to the chapters in this dissertation that use the benchmark for experiments. There are many reasons why the benchmark suite changed over time. For example, most of the Olden benchmarks are small, so their behavior is overwhelmed by Jikes RVM objects, since Jikes RVM is itself written in Java (Section 2.1.2). Chapter 3 works around that problem by reporting separate sets of numbers excluding Jikes RVM objects. However, this was not easily possible in the later papers, hence they exclude all Olden benchmarks except for the largest ones.

2.2.1 Benchmark descriptions

The benchmarks in Table 2.2 are grouped by the suite that they originate from. This section describes the benchmarks in the same order.

2.2.1.1 null

The source code of the null benchmark is a one-liner:

class Null { **public static void** *main*(String[] *args*) { } }

The *main*() is empty, it does nothing, and thus, executing it in Jikes RVM demonstrates what it takes to start up the virtual machine and launch an application without perturbation by what the application itself does. Since Jikes RVM is written in Java, starting it takes a lot: it requires compiling a substantial amount of Java

Table 2.2: Benchmark descriptions. Column "used in" refers to the chapters in this dissertation that use the benchmark for experiments.

Benchmark	Used in	Description
null	3 9 10 11	Empty main method, does nothing.
SPECjvm98		
compress	3 9 10 11	Modified Lempel-Ziv method (LZW).
db	3 9 10 11	Performs database functions on memory resident database.
jack	3 9 10 11	Parser generator, earlier version of JavaCC.
javac	3 9 10 11	The Java compiler from the JDK 1.0.2.
jess	3 9 10 11	Java Expert Shell System.
mpegaudio	3 9 10 11	Decompresses audio files.
mtrt	3 9 10 11	Multi-threaded raytracer.
Colorado Bench		
ipsixql	3 9 – 11	Performs queries against persistent XML document.
jigsaw	3 – – –	W3C's web-server, reference implementation of HTML 1.1.
nfc	3 – – –	Chat-server.
xalan	3 9 10 11	XSLT tree transformation language processor.
Miscellaneous		
deltablue	– 9 – 11	Incremental constraint hierarchy solver.
hsql	– – 10 11	Object-relational database management system.
javalex	– – 10 11	Lexical analyzer generator.
jpat	– – – 11	Java protein analysis tool.
pseudojbb	– 9 – 11	Java business benchmark, with fixed number of transactions.
richards	– – 10 11	Simulates task dispatcher in OS kernel.
soot	– – – 11	Bytecode manipulation and analysis framework.
Java-Olden		
bh	3 9 – 11	Solves the N-body problem using hierarchical methods.
bisort	3 – – –	Sorts by creating and merging bitonic sequences.
em3d	3 – – –	Simulates electromagnetic waves propagation in 3D object.
health	3 9 – 11	Simulates Columbian health care system.
mst	3 – – –	Computes minimum spanning tree of a graph.
perimeter	3 – – –	Computes perimeter of quad-tree encoded raster images.
power	3 9 – 11	Solves the power system optimization problem.
treeadd	3 – – –	Adds the values in a tree.
tsp	3 – – –	Computes estimate for traveling salesman problem.
voronoi	3 – – –	Computes Voronoi diagram of a set of points.

code and allocating a substantial number of objects. The null benchmark is an important point of reference to compare real benchmarks against. A benchmark that behaves almost like the null benchmark has little behavior of its own.

2.2.1.2 SPECjvm98

The benchmarks in the SPECjvm98 suite (`http://www.specbench.org/osg/jvm98`) are based on real-world applications from a variety of areas. The SPECjvm98 benchmarks come with a harness that allows repeatable runs with validity checking and performance measurement. They are intended for measuring and reporting the performance of commercial Java virtual machines, in which case one has to obey strict rules to run them. But they are also probably the most popular benchmarks used in current memory management research, where it often makes sense to change the run rules to get more meaningful results.

Unfortunately, the SPECjvm98 benchmarks are neither free nor open-source: to use them, one has to pay a license fee to the SPEC corporation; and, even then jack, javac, and mpegaudio are available only in bytecode format.

2.2.1.3 Colorado Bench

The Colorado Bench suite (`http://systems.cs.colorado.edu/colorado_bench`) is based on four real-world applications. Johannes Henkel put these programs together. The servers nfc and jigsaw require manual set-up of separate client and server processes, which precludes running them in a batch with many other benchmarks. The two non-server programs ipsixql and xalan are more easy to use, and are among the most challenging for a virtual machine to execute, which makes their results particularly relevant.

2.2.1.4 Miscellaneous

The programs deltablue and richards (`http://research.sun.com/people/mario/java_benchmarking`) are small benchmarks that have been translated into many languages. Mario Wolczko provided the Java versions.

The hsql database (`http://hsqldb.sourceforge.net/`) is a significant open-source project used in the real world. Matthias Hauswirth provided the harness and workload.

The javalex program (`http://www.cs.princeton.edu/~appel/modern/java/JLex/`) is a small, but important open-source tool used by students and others writing compilers in Java, available on Andrew Appel's web-page.

The jpat program is part of the Ashes benchmark suite (`http://www.sable.mcgill.ca/software/#ashesSuiteCollection`) of the Sable group at McGill university.

The pseudojbb program is a slight modification of SPECjbb2000 (`http://www.specbench.org/osg/jbb2000/`). It is not free: to use it, one has to pay a license fee to the SPEC corporation. Whereas the original SPECjbb2000 executes transactions

until a fixed amount of wall-clock time has passed, the modified pseudojbb executes a fixed number of transactions. Darko Stefanović came up with these modifications.

The soot framework (`http://www.sable.mcgill.ca/soot/`) is a significant open-source project used in the real world, developed by the Sable group at McGill university.

2.2.1.5 Java-Olden

The Java Olden benchmark suite (`http://www-ali.cs.umass.edu/~cahoon/olden`) consists of "pointer-intensive" kernels originally written in C. Brendon Cahoon provided the Java versions. These programs are kernels, each executes what is deemed the interesting inner loop of a significant real-world application. The Java Olden benchmarks are open-source and free.

Unfortunately, the Java-Olden benchmarks are small both in code size and runtime behavior. Only three of them (bh, health, and power) exercise a non-trivial amount of object allocation and object death; therefore, only these three are used in more than one chapter of this dissertation. An amusing discussion of the limited runtime behavior of health is [138].

2.2.2 Benchmark suite characterization

An important characteristic of a benchmark suite is that it should contain different kinds of real-world programs. This is the case for the benchmarks used by this dissertation. They include server applications (e.g., pseudojbb and jigsaw), compiler tools (e.g., javac and soot), databases (e.g., db and hsql), multi-media applications (e.g., mpegaudio and mtrt), as well as programs from several other domains.

Another important characteristic of a benchmark suite is that the programs are large. However, the notion of size for a program is vague: a large program may be one whose code base is large, one that allocates a lot of objects, or one that runs for a long time. Furthermore, the allocation volume and run time depend not just on the benchmark, but also on the version and configuration of Jikes RVM on which it runs, on the workload of the benchmark, and on what exactly is being measured. For example, if the optimizing compilation counts towards allocation and run time, those are higher.

Table 2.3 shows the workloads and class file sizes of the 29 benchmarks from Table 2.2. The class file sizes give an indication for how much code the benchmarks have. Jikes RVM itself has a class file size of 4,559 KB (version 2.2.1 configuration FastAdaptiveMarkSweep), and both the benchmark and the VM may use some of the 3,543 KB of class files in the standard libraries.

For memory management research, a more relevant metric of benchmark size is the total allocation and the high watermark. They vary between chapters in this dissertation, since the chapters use different Jikes RVM versions and configurations and different ways to measure them. Thus, each chapter with experimental results presents those numbers separately.

For compiler analysis research in a virtual machine, a more relevant metric of

Table 2.3: Benchmark sizes and workloads. Size is measured in KB of bytecodes in class files.

Benchmark	Size	Workload or command line
null	0.2	*(none)*
compress	17.4	`-s 100 -m1 -M1`
db	9.9	`-s 100 -m1 -M1`
jack	127.8	`-s 100 -m1 -M1`
javac	1,909.2	`-s 100 -m1 -M1`
jess	387.2	`-s 100 -m1 -M1`
mpegaudio	117.3	`-s 100 -m1 -M1`
mtrt	56.5	`-s 100 -m1 -M1`
ipsixql	1,986.2	`3 2`
jigsaw	4,312.9	download complete contents
nfc	556.0	10 rooms, 100 users, 100,000 messages
xalan	4,200.0	`3 2` (but Chapter 10 uses `1 1`)
deltablue	29.8	*(none)*
hsql	896.6	`-clients 1 -tpc 50000`
javalex	88.3	`qb1.lex`
jpat	78.1	`-testDigest`
pseudojbb	420.1	1 warehouse, 70,000 transactions
richards	150.2	*(none)*
soot	4,261.1	`-W --app -t Hello --jimple`
bh	17.3	`-b 500 -s 10`
bisort	4.6	`-s 100000`
em3d	7.1	`-n 2000 -d 100`
health	9.8	`-l 5 -t 500 -s 0`
mst	5.8	`-v 50`
perimeter	9.8	`-l 16`
power	11.2	*(none)*
treeadd	3.1	`-l 20`
tsp	5.9	`-c 60000`
voronoi	13.9	`-n 2048`

benchmark size is the number of classes that actually get loaded, and the number of methods that actually get compiled, at runtime. Chapter 10 presents the first non-trivial pointer analysis that works for all of Java, and reports these size metrics for the benchmarks it uses for performance experiments.

2.3 Traces

Many of the experiments in this dissertation are based on traces, which are chronological recordings of all mutator events relevant to the garbage collector. The traces come from two tracers that record similar events, but differ fundamentally in design, and operate in different versions of Jikes RVM. Understanding the traces is important for understanding the experiments conducted with them. But the experiences from their design and implementation should also be relevant to other researchers who write new tracers, or who use the traces from the second tracer, which are available at `http://www.cs.colorado.edu/~hirzel/gcSim/`.

2.3.1 Trace contents

Each traces is a sequence of events, in chronological order of when the events happen during program execution. There are three kinds of events:

- Object allocation events. An allocation event records the id, type, and size (including two header words) of the object, as well as the thread and stack frame allocating it. In addition, allocation events for Chapter 3 include the owner of the object, whereas allocation events for Chapter 9 include the allocation site, described below.
- Pointer assignment events. A pointer assignment event records the l-value (the location to which the pointer is assigned) and the r-value (the id of the target object to which the pointer points). For pointers on the stack, the l-value consists of the thread id, stack frame, and offset in the stack frame. For pointers in globals, the l-value is the id of the global. For pointers in fields of objects, the l-value is the id of the object, together with the offset of the field.
- Object death events. An object death event records the id of the object that died. It does not need to record the time at which it dies explicitly, since one can easily derive that by summing up the sizes of all objects allocated so far (time is measured in bytes allocated[p.180]).

2.3.2 Tracer implementation

The two tracers for Chapter 3 and for Chapter 9 have a few implementation details in common. Both trace object allocation events with code manually inserted in the Jikes RVM routines that perform allocations. And both derive object ids from object addresses, using a non-copying garbage collector to ensure that the addresses of objects do not change over time. Except for these commonalities, the second tracer differs from the first one to take care of some limitations.

2.3.2.1 Tracer for Chapter 3

The tracer for Chapter 3 uses Jikes RVM as discussed in Section 2.1.5.1 (version 1.1 under Linux on a PowerPC processor, with the baseline compiler). The traces contain events for the boot image and the first, and only, execution of the benchmark's *main*() method.

- Object allocation events. This tracer records allocation events as described in Section 2.3.1, but in addition, records the owner of each allocated object. The application owns objects allocated in application methods, and the VM owns objects allocated in Jikes RVM methods. However, the owner of objects allocated from standard libraries can not be derived by just looking at the method doing the allocation. Instead, for allocations in the standard libraries, the tracer walks up the stack until it reaches a method that does not belong to the standard libraries, and then uses that to classify the object as belonging to the application or to the VM.
- Pointer assignment events. This tracer records pointer assignment events by using the baseline compiler to instrument all code.
- Object death events. This tracer records object death events by recording objects reclaimed by the garbage collector. These events are imprecise, since when the garbage collector finds that an object is dead, it has really died at some time after the previous and before the current collection. The tracer triggers garbage collection frequently to limit the time interval between collections and thus the imprecision. In addition, Chapter 3 compares some numbers that are sensitive to trace granularity to numbers based on perfect deathtime traces provided by Matthew Hertz [72].

2.3.2.2 Tracer for Chapter 9

The tracer for Chapter 9 uses Jikes RVM as discussed in Section 2.1.5.2 (version 2.2.0 under Linux on an Intel x86 processor, with the adaptive optimization system). For these experiments, Jikes RVM calls the *main*() method of the application twice. The traces contain events for everything reachable before the second call to *main*(), and for the second execution of the benchmark's *main*() method.

- Object allocation events. This tracer records allocation events as described in Section 2.3.1, but in addition, records the allocation site of the object. The allocation site is the method that allocates the object, along with the index of the allocating Java bytecode instruction.
- Pointer assignment events. This tracer records writes of pointers to fields of heap objects with a write barrier. It defers recording of writes of pointers to stack or global variables by waiting until the next object allocation, and then scanning the roots for changes. It remembers the value of all roots at the previous allocation, and traces pointer assignment events emulating the

writes necessary to yield the current state of all roots. This methodology does not rely on compiler instrumentation; tracing all writes with instrumentations from the optimizing compiler would be difficult to get correct, and would also perturb the benchmark execution significantly.

- Object death events. This tracer first writes a trace that contains only the imprecise death events from garbage collections, but then makes an offline pass over the trace, and uses the Merlin algorithm to compute the precise deathtimes [72].

2.3.3 Trace quality

Because Jikes RVM is written in Java, traces for Jikes RVM record not just application events, but also runtime system events (Section 2.1.2). As it turns out, the optimizing compiler allocates much more memory than the baseline compiler does, leading to significant differences between traces. This limits the generality of the results: most readers of papers are more interested in the behavior of the Java application than in the behavior of one particular configuration of one particular virtual machine.

The traces for Chapter 3 work around this problem by providing ownership information, which the experiments in Chapter 3 use in a meaningful way to record two separate sets of numbers: one including VM object, one excluding them. Unfortunately, for the experiments in Chapter 9, even with ownership information for objects it is not clear how to exclude VM objects in a meaningful way. Therefore, the traces for Chapter 9 go to great length to at least use the most realistic configuration of Jikes RVM. Hence, they use the adaptive optimization system even though it would have been easier to only use the baseline compiler; and, they perform two runs and record only events from the second run, to hide VM activity for starting up an application.

In addition to these generality trade-offs, the tracer for Chapter 9 contains careful validation, giving a high level of confidence that it traces exactly what actually happens in the virtual machine. The validation consists of two parts: one part happens during trace generation in Jikes RVM, and the second part happens during trace consumption in a garbage collection simulator.

The validation during trace generation works as follows. To minimize perturbation of Java-side behavior, the tracer consists of calls to C functions using the same mechanism that Jikes RVM uses for system call wrappers. Those C functions maintain a complete model of the Java-side heap in form of C data structures, based on the events written to the trace. At periodic intervals, the tracer compares the C-side shadow heap with the Java-side actual heap. If there are discrepancies, the C-side shadow heap is incorrect, meaning some of the events used to construct it are incorrect. This design leads to minimal perturbation (the Java-side of the tracer is minimal), while at the same time yielding comprehensive early warnings (discrepancies are flagged while tracing is still in progress).

The validation during trace consumption works as follows. The simulator emulates a garbage collection, including a reclamation phase. Whenever it reclaims an

object, it compares the current time to the death time for that object recorded in the trace. If it is too early, this indicates that either the simulator or the tracer is buggy.

Implementing tracers that record exactly what actually happens in a virtual machine is difficult. There are many corner cases of behavior that needs to be captured, and one can usually not tell from the trace whether anything is missing. Researchers should therefore go out of their way to add validation mechanisms. There is no reason to believe that an unvalidated tracer produces anything close to the real behavior, which limits the credibility of the results of experiments that use it.

Chapter 3

Understanding Connectivity

This chapter presents experimental data on correlations between connectivity and lifetime. Both connectivity and lifetime are fundamental notions for garbage collectors. Connectivity is the manner in which pointers connect objects to each other and to roots, either directly or transitively[p.192]. Tracing garbage collection[p.184] determines connectivity in a snapshot of the heap. Lifetime is the duration of the life of an object[p.181]. Garbage collectors reclaim objects after they become dead, and age-based garbage collection exploits patterns in the lifetime distributions of objects.

The goal of the experiments in this chapter is to find exploitable patterns of connectivity. This dissertation adopts the methodology to first observe, then act based on those observations. This chapter observes exploitable connectivity behavior in programs, and the subsequent chapters present a new family of garbage collection algorithms designed to act on that behavior. The systematic methodology of observation before action is well established in memory management research. For example, generational garbage collection is based on the observation that most objects die young. Wilson et al. reflect on memory management research paradigms and advocate the style of research that this dissertation adopts [135].

This chapter performs its experiments as follows. It uses Jikes RVM (Section 2.1) to run 22 Java benchmarks (Section 2.2) and obtain traces of all object allocations, pointer writes, and object deaths (Section 2.3). It then analyzes these traces to find exploitable patterns in the correlation between connectivity and lifetime. More specifically, it looks at connectivity between stack roots[p.179] and objects, globals[p.179] and objects, and between objects and objects, and observes the lifetime and deathtime of objects given different notions of connectivity.

The results from this chapter motivate the design of connectivity-based garbage collectors. In a nutshell, this chapter finds that connected objects die together, and that connectivity predicts lifetime in a number of ways. This dissertation proposes partial garbage collection based on partitioning objects by connectivity. That makes it possible to collect connected objects together, just like partitioning by age makes it possible to collect objects of the same generation together. Since connectivity predicts lifetime, the collector can pick partitions of collected objects that are likely to yield much reclaimed memory for little expended garbage collection effort.

This opportunism improves throughput[p.180]. In addition, partial garbage collection improves responsiveness[p.180].

This chapter is based on the ISMM 2002 paper "Understanding the Connectivity of Heap Objects", which also formed the centerpiece of the thesis proposal in January of 2002, thus motivating the rest of this dissertation. Besides providing the starting point of this dissertation, the observations from this chapter are also useful for other memory managers: connectivity-based garbage collection as proposed here is not the only way to exploit connectivity. For example, Guyer and McKinley [63] present a technique for exploiting the results of this chapter in a generational garbage collector.

The rest of this chapter is structured as follows. Section 3.1 presents a taxonomy of connectivity and lifetime. Section 3.2 describes the experimental setup for observing connectivity and lifetime of Java applications. Section 3.3 describes the results of those observations. Section 3.4 discusses related work on observing memory-related behavior of Java applications. Section 3.5 concludes.

3.1 Connectivity and lifetime

This section presents a taxonomy of connectivity and lifetime, providing a way to reason about these concepts in the following sections.

3.1.1 Taxonomy of connectivity

From the point of view of a memory manager, a running program creates new heap objects and modifies pointers in existing objects. The snapshot of the heap at any given time can be viewed as a directed graph, where each object is a node and each pointer is a directed edge. In these terms, the running program can be viewed as a mutator that adds and removes edges and creates new nodes. The garbage collector deletes nodes unreachable from the roots along with their outgoing edges.

The taxonomy of connectivity for this chapter is based on the notion of a *global object graph* (GOG), which has a node for each object created in a program execution and a directed edge for each association between two objects via a pointer in a program execution. The GOG is the union of the snapshot object graphs at all points in a program execution. Figure 3.1 illustrates this with an example. The set of nodes of the GOG $\{o_1, o_2, o_3, o_4\}$, is the union of the objects at the first snapshot $\{o_1, o_2, o_3\}$ and the objects at the second snapshot $\{o_1, o_2, o_4\}$. The set of edges of the GOG, $\{(o_1, o_2), (o_2, o_3), (o_2, o_4)\}$ is the union of the edges at the first snapshot $\{(o_1, o_2), (o_2, o_3)\}$ and the edges at the second snapshot $\{(o_2, o_4)\}$.

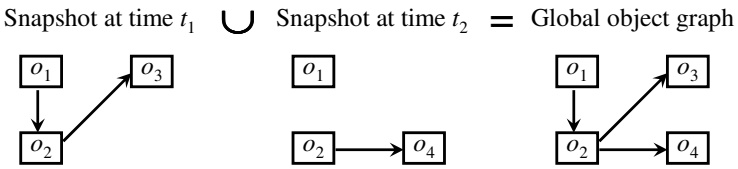


Figure 3.1: The global object graph is the union of the snapshot object graphs.

There are two orthogonal dimensions to connectivity: whether it is immediate or transitive, and what kinds of entities are connected. Along the first dimension, immediate connectivity refers to direct pointers, whereas transitive connectivity refers to paths of pointers via which objects are reachable[p.179]. Along the second dimension, the kinds of entities involved are stack variables, globals[p.179], and heap objects. In Java, the target of a pointer is always a heap object; hence, the second dimension uses only the source of pointers to distinguish three kinds of connectivity: from a stack variable to a heap object, from a global to a heap object, or between two heap objects. For brevity, they are called stack-, globals-, and heap-connectivity.

All six combinations of the connectivity dimensions (immediate or transitive; stack-, globals-, or heap-connectivity) are defined in terms of graph reachability in the global object graph, where immediate stack-connectivity or globals-connectivity is captured as an annotation on the node representing the target object.

In addition to the basic two-dimension taxonomy, this chapter also investigates a few derived notions of connectivity. For transitive stack- and globals-connectivity, it distinguishes whether the target object escapes its stack frame or even its thread. For immediate heap-connectivity, it distinguishes whether the originating object experiences any pointer mutation other than initialization. And for transitive heap-connectivity, it distinguishes the special cases of belonging to the same strongly or weakly connected component.

An object escapes its stack frame if it becomes reachable from a global or from a variable higher in the stack than the stack frame of the method that allocated it. That means that the object is accessible from a caller of the allocating method. Objects that do not escape their stack frames can sometimes be allocated directly on the stack, rather than on the heap. An object escapes its thread if it is reachable from variables on the stacks of different threads. Objects that do not escape their threads can be manipulated without synchronization.

A strongly connected component (SCC[p.176]) in the GOG is a maximal set of objects such that each object in the SCC is reachable from all other objects in the SCC. For example, in Figure 3.2, objects $\{o_6, o_7, o_9\}$ form a strongly-connected component. A weakly connected component (WCC[p.176]) in the GOG is a maximal set of objects such that if one ignores the directions of edges, each object in the WCC is reachable from all other objects in the WCC. For example, in Figure 3.2, objects $\{o_5, o_6, o_7, o_8, o_9\}$ form a weakly connected component.

3.1.2 Taxonomy of lifetime

The lifetime of an object is the duration of its life, measured in bytes[p.180] from its birth time to its death time.[p.181] The taxonomy of lifetime presented here is based on Blackburn *et al.* [18], who classify objects by their lifetime and deathtime into shortlived, longlived, and immortal objects. This dissertation extends the definition to further classify immortal objects into quasi immortal and truly immortal as follows:

- An object that dies at the termination of the program is truly immortal[p.181].

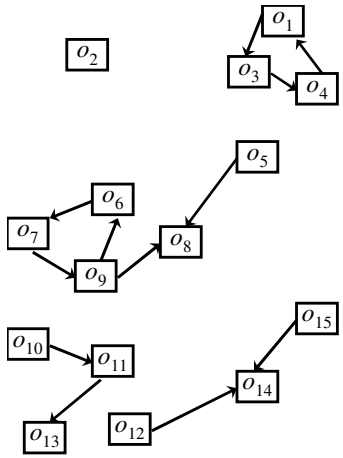


Figure 3.2: Example global object graph. This picture omits the annotations on nodes, which, among other things, carry information on immediate stack-connectivity and globals-connectivity.

- Else, if the time from the object's deathtime to the termination of the program is shorter than the object's lifetime, then the object is quasi immortal[p.181].
- Else, if the object's lifetime is shorter than the threshold $T_a \times high\_watermark$, the object is shortlived (this chapter uses $T_a = 0.2$). The *high_watermark* is the maximum number of bytes in reachable heap objects during the program execution.
- Else, the object is longlived.

The motivation for the definition of shortlived as a fraction of *high_watermark* is that generational garbage collectors often reserve a fixed fraction of the heap for the nursery. The exact value of T_a turns out to have little impact on which objects are classified as shortlived or longlived unless it is very small: most shortlived objects live for a much shorter time than the threshold $0.2 \times high\_watermark$ that this chapter uses as the cut-off.

Ideally, a tracing garbage collector should not expend any effort on quasi immortal or truly immortal objects, but focus mostly on shortlived and occasionally on longlived objects.

Some of the benchmarks, such as those in the SPECjvm98 suite, are invoked by harness code that is also written in Java and executed by the VM. This chapter slightly modifies the lifetime taxonomy by considering objects that survive until the termination of the benchmark proper as truly immortal even if they do not survive until the termination of the harness.

3.1.3 Combining the taxonomies of connectivity and lifetime

The taxonomy for connectivity is based on the global object graph, plus some additional information for the more sophisticated connectivity notions such as whether objects experience mutation.

The taxonomy for lifetime is based on two numbers for each object, birth time and death time, both measured in bytes. They are represented as attributes of global object graph nodes.

This chapter computes statistics on the global object graph by counting nodes to which certain unary predicates apply, or pairs of nodes to which certain binary predicates apply. For example, combining the unary connectivity predicate "object that is transitively reachable from global variables" with the unary liveness predicate "object that is shortlived" answers the question whether objects reachable from globals are more or less likely to die young than other objects.

3.2 Methodology

Chapter 2 described the infrastructure that yields the traces (2.3) of 29 Java programs (2.2) running on top of Jikes RVM (2.1). This section gives some more details on the 22 Java programs used in this chapter, and on how this chapter uses their traces.

3.2.1 Benchmarks

While Section 2.2 only gives some superficial information on the benchmarks used in this chapter (what they do, which workload they use, and how large their code-base is), this section characterizes them in more detail. The ***average*** row in each table gives the arithmetic mean of the results for all benchmarks except benchmark null.

Table 3.1 characterizes how large each benchmark is from the perspective of the garbage collector. The high watermark is the maximum number of bytes simultaneously live at any point during the program execution. It includes the size of the boot image, which explains why even null has a high watermark of about 14 MB. The GC interval is the time in bytes between forced garbage collections (the forced garbage collections provide object death times). The garbage collection intervals are approximately 1/25 of the high watermark; depending on the benchmark, that means one collection per 0.53MB to 1.31MB of allocation. The total allocation is the amount of allocation throughout the entire benchmark run, again including the boot image.

Most measurements in this chapter are presented by two sets of numbers: one for all allocated objects (including VM and library objects), and one for just the objects allocated on behalf of the application. The numbers for all allocated objects are relevant because VM boot-up and compilation are part of the program's execution. The application-only numbers emphasize the differences between benchmarks more, and are probably more transferable to other virtual machines that, unlike Jikes RVM, are not written in Java.

Table 3.1: Benchmark sizes.

Benchmark	High watermark (bytes)	GC interval (bytes)	Total allocation (bytes)	(objects)
null	14,109,770	553,905	14,396,503	106,009
compress	27,688,408	1,193,412	132,931,724	226,002
db	23,503,105	951,881	97,899,266	3,401,539
jack	16,907,838	678,956	331,031,287	8,194,044
javac	25,296,557	1,080,090	285,631,761	8,228,933
jess	17,056,718	684,702	334,187,450	8,662,674
mpegaudio	16,578,151	789,493	35,850,575	380,054
mtrt	22,414,466	895,044	173,683,581	6,889,168
bh	14,580,046	591,819	42,900,870	1,212,329
bisort	14,628,360	588,607	16,085,265	176,878
em3d	19,534,225	805,814	22,101,972	135,894
health	16,563,273	559,042	38,618,097	1,332,116
mst	14,254,193	558,853	15,446,269	124,317
perimeter	27,458,366	1,118,846	31,528,263	595,507
power	14,914,494	608,275	38,101,825	912,770
treeadd	33,644,773	1,375,696	35,751,748	1,159,451
tsp	16,836,300	669,420	21,583,991	310,956
voronoi	14,832,375	590,537	17,712,879	191,434
ipsixql	17,141,410	689,387	99,908,400	2,357,562
jigsaw	26,487,443	1,000,000	257,452,354	4,289,782
nfc	25,643,076	1,000,000	173,637,549	2,154,719
xalan	22,784,083	905,853	123,412,189	1,637,966
average	20,416,555	825,510	110,736,063	2,503,528

Table 3.2: Benchmark statistics. All numbers are in percent of allocated objects. The lifetime numbers in parentheses count only objects where the owner is the application.

Benchmark	Owner (%)			Lifetime (%)			
	Boot	RVM	App.	Shortlived	Longlived	Quasi imm.	Truly imm.
null	76.4	23.6	0.0	2.4 (n/a)	0.0 (n/a)	9.2 (n/a)	88.4 (n/a)
compress	35.8	62.8	1.3	42.9 (37.2)	9.6 (17.1)	0.0 (2.3)	47.4 (43.4)
db	2.4	3.2	94.4	87.6 (90.1)	0.1 (0.1)	0.1 (0.1)	12.2 (9.8)
jack	1.0	12.7	86.3	96.6 (97.6)	1.8 (2.1)	0.1 (0.1)	1.6 (0.2)
javac	1.0	23.3	75.7	81.0 (77.0)	13.5 (17.8)	1.1 (1.5)	4.4 (3.6)
jess	0.9	7.6	91.4	98.0 (99.3)	0.3 (0.3)	0.0 (0.0)	1.7 (0.4)
mpegaudio	21.3	77.7	1.0	68.6 (7.6)	0.0 (0.0)	0.0 (0.0)	31.3 (92.4)
mtrt	1.2	2.9	95.9	93.7 (95.1)	0.0 (0.0)	2.2 (2.3)	4.1 (2.6)
bh	6.7	5.1	88.2	88.4 (95.5)	1.2 (1.3)	0.4 (0.4)	10.1 (2.7)
bisort	45.8	17.2	37.1	11.5 (0.0)	0.0 (0.0)	0.0 (0.0)	88.4 (100.0)
em3d	59.6	28.6	11.8	21.1 (0.0)	0.0 (0.0)	0.0 (0.0)	78.9 (100.0)
health	6.1	4.0	89.9	82.2 (88.1)	0.5 (0.5)	0.5 (0.5)	16.9 (10.8)
mst	65.2	30.7	4.1	13.5 (0.0)	0.0 (0.0)	6.0 (0.0)	80.4 (100.0)
perimeter	13.6	10.3	76.1	8.5 (0.0)	0.0 (0.0)	0.0 (0.0)	91.5 (100.0)
power	8.9	5.3	85.8	85.1 (94.4)	0.0 (0.0)	0.0 (0.0)	14.9 (5.6)
treeadd	7.0	2.6	90.4	1.7 (0.0)	0.0 (0.0)	0.0 (0.0)	98.3 (100.0)
tsp	26.0	10.7	63.2	45.5 (60.7)	0.0 (0.0)	0.0 (0.0)	54.5 (39.3)
voronoi	42.3	26.1	31.6	27.0 (22.2)	0.0 (0.0)	0.1 (0.2)	72.9 (77.7)
ipsixql	3.4	16.6	80.0	84.0 (85.7)	6.7 (8.4)	1.9 (2.4)	7.3 (3.4)
jigsaw	1.9	68.0	30.2	93.4 (92.2)	0.0 (0.0)	0.0 (0.0)	6.5 (7.8)
nfc	3.8	21.7	74.6	93.3 (99.4)	0.0 (0.1)	0.0 (0.0)	6.7 (0.5)
xalan	4.9	92.5	2.5	87.5 (87.2)	0.1 (0.9)	0.1 (1.4)	12.2 (10.5)
average	17.1	25.2	57.7	62.4 (58.5)	1.6 (2.3)	0.6 (0.5)	35.3 (38.6)

Table 3.2 shows the distributions of owners and lifetimes among the objects allocated by each benchmark.

The "Owner" columns in Table 3.2 categorize heap objects into the percentage of total objects that are allocated in the Jikes RVM boot image ("boot"), by the running VM ("RVM"), and by the application ("App."). Section 2.1.2 includes a discussion of the boot image. Objects allocated at runtime are classified into Jikes RVM objects and application objects by their allocation site. In the case where the allocation site is within the standard Java library, the ownership classifier traverses the dynamic chain until it encounters a caller in the Jikes RVM runtime system or the application, and uses that as the allocation site. For the larger benchmarks, most of the objects are allocated by the application itself; on the other end of the spectrum, the benchmark null intentionally does not allocate any application objects.

The "Lifetime" columns in Table 3.2 categorize heap objects by their lifetime following the definitions in Section 3.1.2. The numbers in parentheses count only objects where the owner is the application (this convention holds throughout this chapter). The "n/a" for benchmark null indicates that the application does no allocation.

Most benchmarks have a high percentage of shortlived objects, confirming the weak generational hypothesis [67]. Few objects are longlived or quasi immortal, but many benchmarks have a significant fraction of truly immortal objects. The number of truly immortal objects is particularly high for benchmark null. This is because null allocates only system objects, and many of these survive until the VM terminates. It may therefore be worthwhile to treat system objects specially in a memory manager for a VM implemented in Java. A significant percentage of application-only objects are also truly immortal. This is contrary to the strong generational hypothesis and motivates techniques like pretenuring [18, 66]. For those benchmarks where almost all objects are truly immortal, never attempting to collect garbage may be the best approach to memory management [53].

3.2.2 Global object graph construction

The three kinds of trace events (object allocation, pointer write, and object death) drive the construction of a global object graph (GOG).

An object allocation event creates a node in the GOG and annotates it with birth time, type, size, and allocation context. It also advances the allocation clock by the size.

There are two kinds of pointer assignment events. A pointer assignment event where the pointer is stored into a field of a heap object creates an edge between two objects if the edge does not already exist. A pointer assignment event where the pointer is stored into a global or a stack variable updates information in the pointed-to object that determines whether it escapes.

An object death event updates the deathtime and lifetime of the corresponding object. Since the garbage collector generates the deallocation events, the timings for these events are not precise: an object reported as dead at garbage collection n may have become unreachable any time after collection $n-1$. Performing relatively frequent garbage collections reduces this imprecision. In addition, Section 3.3.3.4 compares the results from the traces with imprecise deathtimes to precise deathtime traces [72] for the set of numbers most sensitive to this issue.

After processing all trace events, the GOG contains the information to capture immediate connectivity and lifetime. A post-processing phase propagates information over the graph to annotate nodes with transitive connectivity information as well.

3.3 Results

This section presents the results. Section 3.3.1 investigates stack-connectivity, Section 3.3.2 investigates globals-connectivity, and Section 3.3.3 investigates heap-connectivity.

While most results in this section are kept general, so they can be used to motivate a variety of garbage collection techniques, Section 3.3.4 is a preliminary

investigation of questions specific to connectivity-based garbage collection.

3.3.1 Correlation between lifetime and stack-connectivity

This sections considers two kinds of stack-connectivity: objects that are reachable only from the stack, and objects that escape their allocating activation records or threads.

3.3.1.1 Objects reachable only from the stack

This section investigates immediate stack-connectivity, it looks at objects that are not reachable from globals or from other objects. Given sufficient compiler support, these objects should be relatively cheap to garbage collect. Figures 3.3(a) and (b) present data for all objects and only for objects allocated on behalf of the application, respectively. In both figures the length of a bar gives the percentage of objects that are reachable only from the stack. Each bar has four segments, for shortlived, longlived, quasi immortal, and truly immortal objects.

The results indicate that the benchmarks have a significant percentage of objects that are pointed to *only* from the stack: in 11 of the 22 benchmarks it is higher than 30%. For most benchmark programs, the majority of these objects are shortlived. If a compiler can identify allocation sites whose objects do not escape into the heap or globals, these objects can be allocated in a special area where they can be garbage collected cheaply.

3.3.1.2 Lifetime of escaping objects

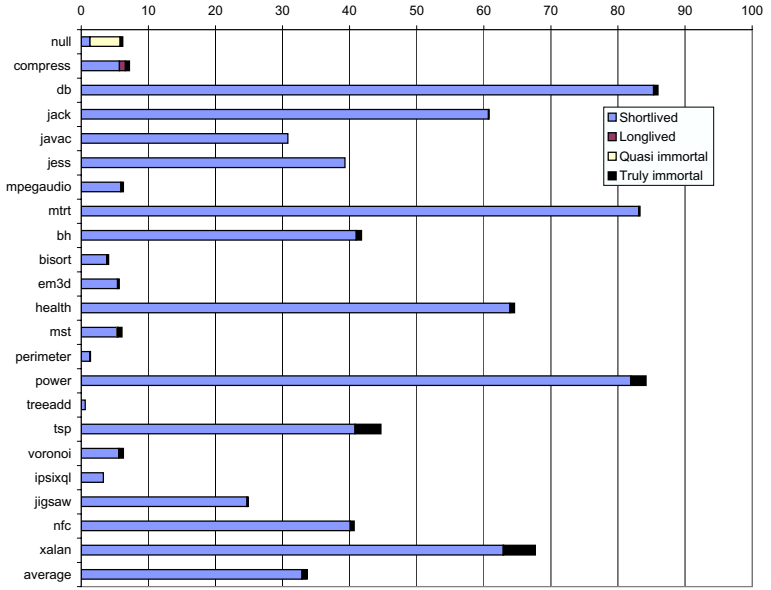
Recently, there has been much work on escape analysis [36, 56, 122]. Prior work has used escape analysis to eliminate synchronization or to allocate objects on the stack [55, 127]. This section investigates whether object escape rates have any correlation to object lifetimes. Stack allocation manages the memory of some of the heap objects based on their stack-connectivity, and can thus be viewed as a special case of connectivity-based garbage collection.

Table 3.3 gives the percentage of objects that escape the stack frame or thread that created them (the numbers in parentheses consider only application objects). The Jikes RVM runtime system creates threads for garbage collection and finalization and thus even single-threaded benchmarks may have thread-escaping application objects.

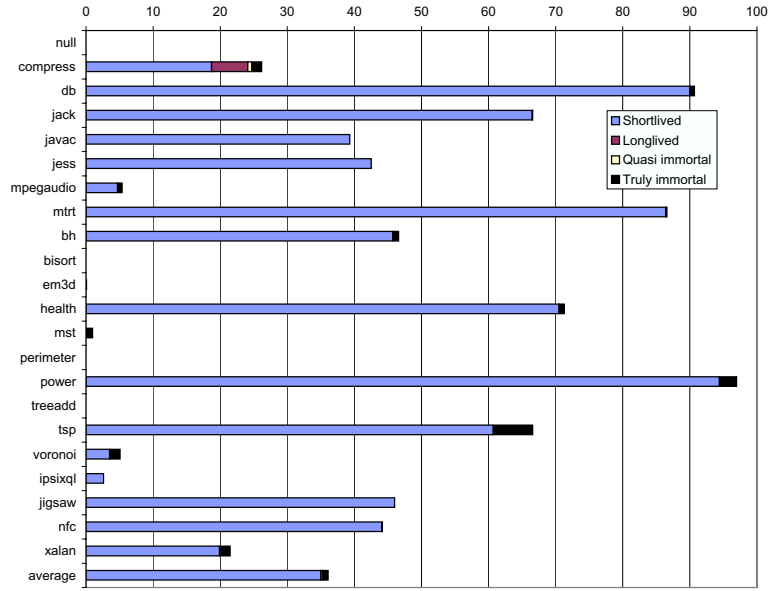
On average, only 26.8% of all objects are non-escaping, the rest escape at least their stack frame, and 19% even escape their thread.

Figure 3.4 shows the lifetime of objects that escape their thread. In Figure 3.4, the length of the bars shows the percentage of objects that escape their thread. Each bar is subdivided into four segments, one for each lifetime bin. Figure 3.4(a) shows data for all objects, whereas Figure 3.4(b) only shows data for the application objects.

Figure 3.4 shows that while escaping objects are often truly immortal, that is not always true. In particular, benchmark nfc has many objects that escape a thread,



(a) In percent of all allocated objects.

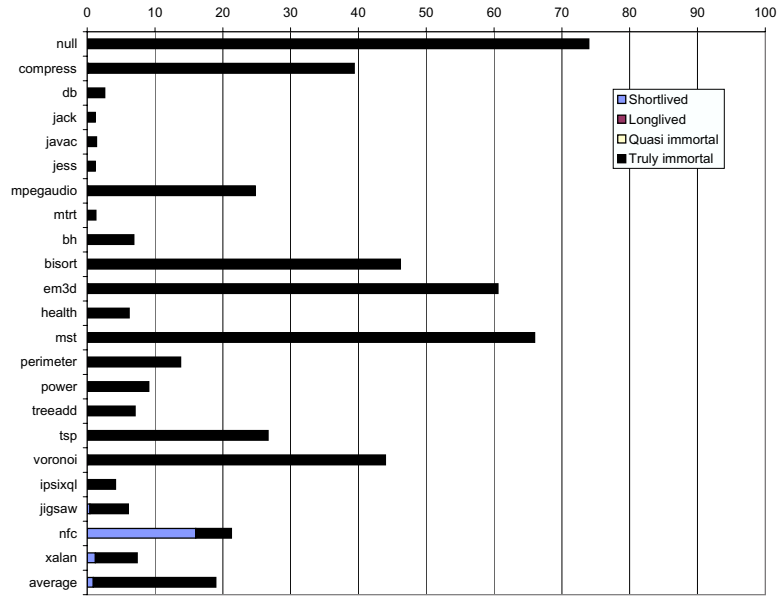


(b) In percent of objects allocated by application.

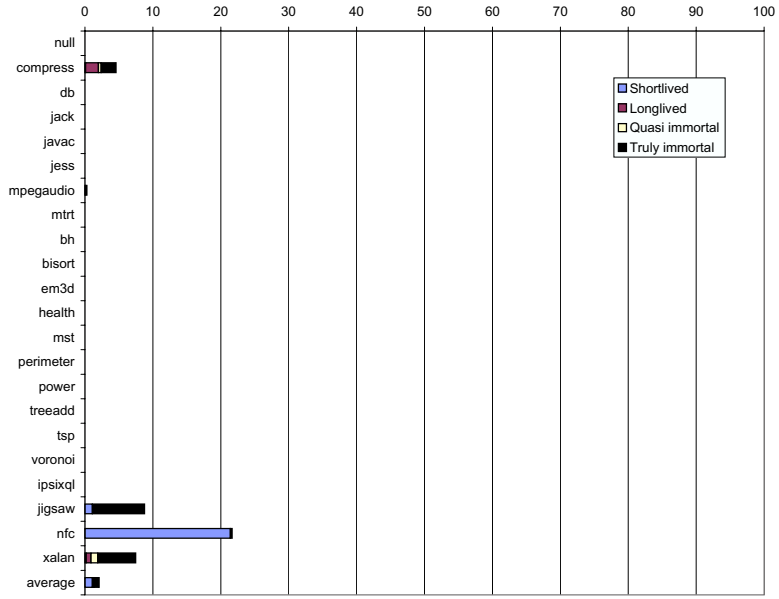
Figure 3.3: Lifetime of objects pointed to only by the stack. For most benchmarks the longlived and quasi immortal segments of the bars are nearly empty.

Table 3.3: Escape rates (in percent of allocated objects). The numbers in parentheses count only objects where the owner is the application.

Benchmark	No escape	Stack frame	Thread
null	10.0 (n/a)	90.0 (n/a)	74.0 (n/a)
compress	10.5 (21.5)	89.5 (78.5)	39.4 (4.5)
db	0.8 (0.1)	99.2 (99.9)	2.6 (0.0)
jack	37.2 (39.2)	62.8 (60.8)	1.2 (0.0)
javac	29.7 (37.3)	70.3 (62.7)	1.4 (0.0)
jess	40.4 (43.3)	59.6 (56.7)	1.2 (0.0)
mpegaudio	9.4 (8.3)	90.6 (91.7)	24.8 (0.3)
mtrt	82.5 (85.6)	17.5 (14.4)	1.4 (0.0)
bh	41.1 (45.2)	58.9 (54.8)	7.0 (0.0)
bisort	6.4 (0.0)	93.6 (100.0)	46.3 (0.0)
em3d	14.8 (50.0)	85.2 (50.0)	60.6 (0.0)
health	13.8 (14.3)	86.2 (85.7)	6.2 (0.0)
mst	9.4 (0.0)	90.6 (100.0)	66.1 (0.0)
perimeter	2.2 (0.0)	97.8 (100.0)	13.9 (0.0)
power	84.7 (97.0)	15.3 (3.0)	9.1 (0.0)
treeadd	1.0 (0.0)	99.0 (100.0)	7.1 (0.0)
tsp	46.2 (66.7)	53.8 (33.3)	26.8 (0.0)
voronoi	7.4 (0.1)	92.6 (99.9)	44.0 (0.0)
ipsixql	4.0 (2.7)	96.0 (97.3)	4.2 (0.0)
jigsaw	26.5 (36.1)	73.5 (63.9)	6.2 (8.9)
nfc	28.2 (26.4)	71.8 (73.6)	21.3 (21.7)
xalan	66.6 (18.8)	33.4 (81.2)	7.4 (7.5)
average	26.8 (28.2)	73.2 (71.8)	19.0 (2.0)



(a) In percent of all allocated objects.



(b) In percent of objects allocated by application.

Figure 3.4: Lifetime of objects escaping their thread. For most benchmarks the longlived and quasi immortal segments of the bars are nearly empty.

but are shortlived. Since nfc is one of the more realistic benchmarks, this leads to the conclusion that a garbage collector cannot ignore thread-escaping objects; indeed many of them may be shortlived. The intuition for this is that server applications often use shortlived thread-escaping objects to communicate between threads. Other experiments showed that the correlation between lifetime and escaping from the stack frame is weaker than the correlation between lifetime and escaping from the thread.

When repeating the above experiments for the three multi-threaded benchmarks, mtrt, jigsaw, and nfc, and ignoring objects that escaped to the garbage collector thread, the data changed slightly, but the main conclusions remained the same. For example, most of the objects that escaped a thread in nfc remained shortlived.

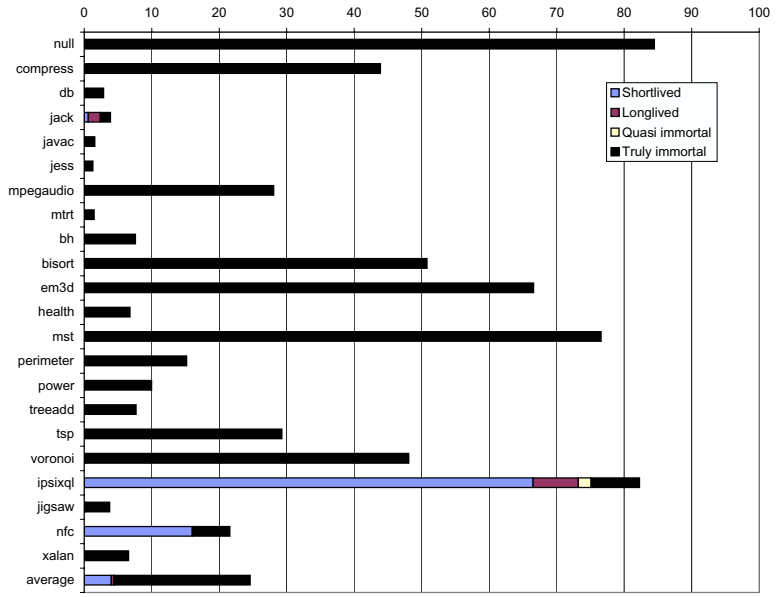
3.3.2 Correlation between lifetime and globals-connectivity

Figure 3.5 shows the lifetime of objects transitively reachable from globals. It includes objects that may also be reachable from the stack or heap. In Figure 3.5, the length of the bars shows the percentage of objects reachable from globals. Each bar is subdivided into four segments, one for each lifetime bin. Figure 3.5(a) shows data for all objects, whereas Figure 3.5(b) only shows data for the application objects.

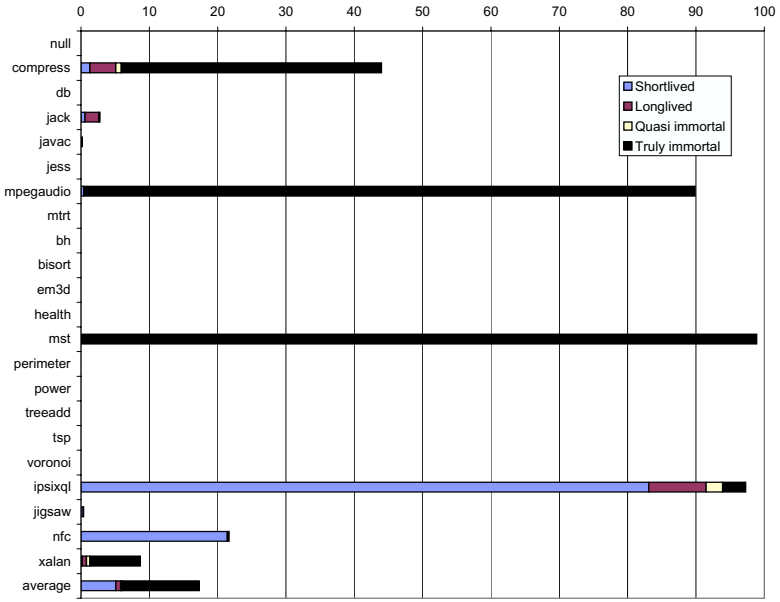
Because global variables exist as long as their classes exist, one can expect objects reachable from globals to be immortal. (With an interprocedural liveness analysis for global variables, a garbage collector may be able to collect objects even if they are still reachable from globals [80]. This chapter disregards this possibility, because interprocedural liveness analysis for globals is probably unrealistic for Java programs.) Figure 3.5 confirms the expectation. Figure 3.5(a) shows that with the exception of benchmarks jack, nfc, and ipsixql, most objects reachable from globals are truly immortal. Of these benchmarks, jack has a relatively small percentage of objects reachable from globals.

The benchmark ipsixql has a large SCC that is reachable from globals and is heavily mutated. Figure 3.6 demonstrates this pictorially. The horizontal axis gives time in bytes allocated. The vertical axis (in log scale) gives the size of an SCC in number of objects. There is one curve for each SCC with at least two objects. A point (x, y) on a curve, C, means that at time x, SCC C has size y objects. Figure 3.6 shows that benchmark ipsixql has one very large SCC and several smaller ones. Although many of the objects in the largest SCC die together at the end of the program, significant parts of the SCC are continuously replaced with newly allocated objects. This large SCC is part of a cache that stores faulted objects. (Because sources for ipsixql are not available, these observations are based on the output of a decompiler, and thus are somewhat speculative). The atypical behavior of this large SCC dominates the behavior of ipsixql, and among other things destroys the correlation between lifetime and reachability from globals. In addition, Section 3.3.4 will show that ipsixql incurs a significant write barrier overhead.

In summary, for all benchmarks (except for jack, nfc, and ipsixql) there is a strong correlation between reachability from globals and lifetime. A generational garbage collector could exploit these observations by eagerly promoting objects reachable



(a) In percent of all allocated objects.



(b) In percent of objects allocated by application.

Figure 3.5: Lifetime of objects reachable from globals. For most benchmarks the longlived and quasi immortal segments of the bars are nearly empty.

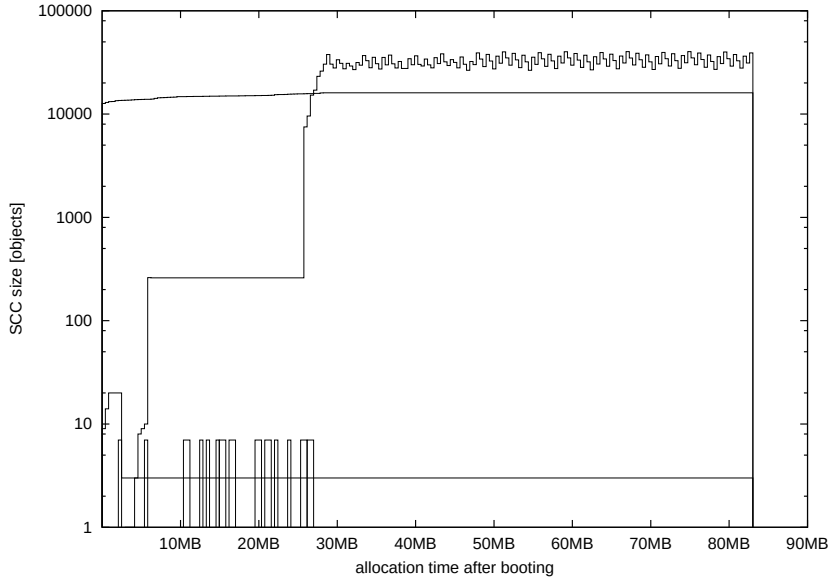


Figure 3.6: SCCs in benchmark ipsixql.

from globals to old generations.

3.3.3 Correlation between lifetime and heap-connectivity

This section considers several kinds of heap-connectivity. It investigates how likely it is for objects that are connected by a pointer to have the same deathtime (Section 3.3.3.1) and whether the popularity of objects is related to their lifetimes (Section 3.3.3.2). Next, it investigate how likely it is for transitively connected objects to have the same deathtime (Section 3.3.3.3), and it conclude by evaluating how sensitive these same-deathtime results are to the methodology of tracing with frequent garbage collections (Section 3.3.3.4).

3.3.3.1 Linked objects

This section considers immediate heap-connectivity, it explores the deathtime of directly-linked objects. First, it looks at how often objects are modified. Consider a program that repeatedly modifies an object o such that a field in o points to one of many different objects at different times. In this case, one can expect o's deathtime to be largely unrelated to the deathtime of objects that o points to. If, on the other hand, the program modifies few objects after initialization, then one can expect a significant correlation between the deathtime of connected objects.

This section views the first non-null assignment to an object field as "initialization" and subsequent assignments to the same field as "mutations". Column

Table 3.4: Mutation rates in % of allocated objects and write barrier overheads in % of total execution time. In the mutation rates, the numbers in parentheses consider only objects where the owner is the application.

Benchmark	Mutated	Write barrier
null	18.6 (n/a)	15.8
compress	10.5 (8.0)	3.9
db	0.7 (0.0)	1.5
jack	4.0 (4.2)	5.7
javac	18.2 (23.4)	19.4
jess	3.5 (3.3)	8.1
mpegaudio	7.5 (3.8)	3.4
mtrt	1.2 (0.9)	4.3
bh	5.7 (4.5)	1.2
bisort	29.8 (50.0)	6.6
em3d	14.8 (0.0)	2.0
health	16.4 (16.6)	1.0
mst	16.2 (1.3)	2.6
perimeter	3.5 (0.0)	1.2
power	2.2 (0.0)	0.1
treeadd	1.7 (0.0)	6.0
tsp	27.5 (33.3)	10.9
voronoi	33.8 (73.2)	5.2
ipsixql	1.7 (0.4)	19.8
jigsaw	4.1 (6.4)	–
nfc	14.6 (17.6)	–
xalan	2.0 (2.5)	32.4
average	10.5 (11.9)	7.6

"Mutated" of Table 3.4 gives the percentage of all allocated heap objects that are mutated during program execution. The numbers in parentheses are the percentage of objects allocated by the application that are mutated during program execution. Table 3.4 shows that programs do not mutate the majority of objects, and thus, the lifetimes of linked objects are likely to be related.

Column "Write barrier" of Table 3.4 gives the overhead of the write barrier. The measurements use a full copying collector that does not need write barriers for correctness, and then run each benchmark twice, once with and once without write barriers. The time difference between the two runs is the write barrier overhead. These measurements use a Jikes RVM version 2.0.2 FastSemispace image on a uniprocessor PPC/AIX machine. Benchmarks jigsaw and nfc are omitted because they are interactive. The write barrier is implemented as a sequential store buffer. Because Jikes RVM is written in the Java programming language, the overheads

include the execution of the application, VM, and optimizing compiler at its default optimization level (1).

The write barrier numbers in Table 3.4 measure the case where the barrier executes the same actions for each write. In practice, write barriers for generational collectors are usually optimized to first perform a fast check whether the write is to an old object, and only executing the (slower) remainder of the actions if that test succeeds. With this optimization, write barriers in Jikes RVM incur a lower overhead: the geometric mean of the write barrier overheads for the SPECjvm98 benchmarks and pseudojbb is only 3.2% [19, Table 2].

Table 3.5 investigates the correlation between direct object connectivity and object deathtimes. Column "$o_1 \rightarrow o_2$" of Table 3.5 gives the probability that two adjacent objects in the GOG have the same deathtime. For many programs the probability is nearly 100%. In contrast, column "Any pair" gives the probability that any two possibly unlinked objects in the program die at the same time. This value is computed by considering all pairs, both linked and unlinked. In most cases, the probability that linked objects die at the same time is much higher than the probability of any two objects dying at the same time.

Column "$o_1 \rightarrow o_2$, o_1 mutated" in Table 3.5 gives the probability that two objects, o_1 and o_2, have the same deathtime given that o_1 points to o_2 and o_1 is mutated. For 14 of the 22 benchmarks (19 of 22 benchmarks when looking at the application-only numbers), these probabilities are lower than the ones in column "$o_1 \rightarrow o_2$".

Tables 3.4 and 3.5 show that for many benchmarks there is both a high probability that objects are not mutated and that objects linked by a pointer have the same deathtime. However, for some programs, such as db, even though it has a low mutation rate (0.7%), it also has a relatively low probability of linked objects dying at the same time (22.7%). In other words, a low percentage of modified objects is no guarantee for a high correlation of deathtimes of connected objects. Apparently, even though db modifies only few objects, the modifications happen in key places and thus have a big impact on the deathtimes of linked objects.

A garbage collector can exploit these results by clustering linked objects together. Since on average linked objects have a 80.4% probability of dying at the same time, the garbage collector will be able to free up many objects at once.

3.3.3.2 Incoming pointers

This section investigates whether there is a correlation between the popularity of an object and its lifetime. A "popular object" is one that is pointed to by many other objects [85].

Counting the number of objects pointed to by at least two other heap objects found that, for most benchmarks, fewer than 40% of the objects had at least two immediate predecessors. The lifetime distribution of objects pointed to by at least two other heap objects varied widely: in some cases, most of these objects were shortlived, while in other cases, most of these objects were truly immortal.

To conclude, there was little correlation between the popularity of an object and

Table 3.5: Pairs of objects with same deathtime (in percent of pairs of objects with given connectivity). The numbers in parentheses count only objects where the owner is the application.

Benchmark	Any pair	$o_1 \rightarrow o_2$	$o_1 \rightarrow o_2$, o_1 mutated
null	79.5 (n/a)	96.5 (n/a)	97.7 (n/a)
compress	22.8 (19.5)	95.5 (63.9)	96.8 (44.2)
db	2.0 (2.1)	22.7 (19.7)	12.5 (0.6)
jack	0.3 (0.3)	54.1 (45.4)	19.1 (5.2)
javac	0.7 (0.9)	66.1 (65.8)	70.4 (68.7)
jess	0.2 (0.3)	63.6 (58.6)	90.1 (89.0)
mpegaudio	10.0 (83.6)	94.8 (74.8)	94.6 (43.8)
mtrt	0.7 (0.7)	77.3 (71.2)	75.9 (57.6)
bh	2.6 (2.3)	89.3 (87.0)	71.0 (51.1)
bisort	78.9 (100.0)	98.9 (100.0)	99.6 (100.0)
em3d	63.7 (100.0)	99.1 (100.0)	97.7 (100.0)
health	4.4 (3.1)	11.4 (9.5)	4.7 (3.6)
mst	66.2 (100.0)	96.1 (100.0)	96.6 (100.0)
perimeter	84.0 (100.0)	99.3 (100.0)	96.0 (n/a)
power	3.8 (2.8)	96.3 (100.0)	97.8 (n/a)
treeadd	96.6 (100.0)	99.4 (100.0)	97.7 (n/a)
tsp	27.9 (15.7)	98.2 (100.0)	99.4 (100.0)
voronoi	44.7 (38.0)	89.1 (82.7)	85.3 (78.9)
ipsixql	1.3 (1.3)	79.1 (79.3)	78.8 (25.2)
jigsaw	0.8 (1.6)	88.9 (83.6)	92.1 (85.9)
nfc	1.0 (0.8)	75.7 (68.9)	69.5 (64.0)
xalan	2.2 (21.9)	94.3 (93.2)	94.2 (82.5)
average	24.5 (33.1)	80.4 (76.4)	78.1 (61.1)

its lifetime.

3.3.3.3 SCCs and lifetime

Table 3.5 suggests that immediate heap-connectivity is usually, but not always, a good indicator of deathtime. This section considers transitive notions of heap-connectivity: strongly and weakly connected components (SCCs and WCCs, Section 3.1).

Column "In nontriv. SCC" in Table 3.6 shows the percentage of objects that belong to SCCs with at least two objects in the global object graph. On average, only a minority of objects are members of nontrivial SCCs.

Column "Same SCC" in Table 3.6 gives the probability that two objects in the same SCC have the same deathtime. Column "Same WCC" in Table 3.6 gives the probability that two objects in the same WCC have the same deathtime. The numbers in parentheses consider only objects allocated by the application. Since an SCC implies stronger connectivity, we expected that the probability would be higher for an SCC than for a WCC.

Table 3.6 shows that for many programs there is a high probability that objects in the same SCC die together. For many benchmarks the probability for two objects in an SCC having the same deathtime is greater than the probability of two linked objects (Table 3.5) having the same deathtime. A garbage collector could exploit these observations by designating any object in an SCC as the *key object* [67], the object whose death likely coincides with the death of other objects connected to it. Thus, when that object dies, there is a good chance that the rest of the SCC is also garbage.

3.3.3.4 Trace granularity

Most of the numbers in this chapter are based on traces with three kinds of events: object allocation events, pointer assignment events, and deallocation events (see Section 2.3.1). To obtain the deallocation events, the tracer performed frequent garbage collections. In the traces, all objects that become unreachable between collection n and collection $n+1$ die at the time when collection $n+1$ happens. Thus, the traces are *granulated*: deathtimes are not precise, but rounded up to a multiple of the GC interval (rightmost column in Table 2.2).

Until recently, the only known way to get *precise deathtime traces* (not granulated traces) was to perform a garbage collection at every allocation (e.g. [117]), which is prohibitively expensive. Recently, Hertz *et al.* proposed the Merlin algorithm [72] that generates precise deathtime traces much faster than the brute force method. Using Hertz's precise deathtime traces to regenerate the results showed that it made a significant difference in the same deathtime numbers (Tables 3.5 and 3.6) but not in the classification of objects by lifetime into shortlived, longlived, quasi immortal, and truly immortal.

Table 3.7 shows how using granulated traces inflated the numbers in Tables 3.5 and 3.6. The numbers in Table 3.7 come from recomputing the numbers in Tables 3.5 and 3.6 using precise deathtime traces, and then, subtracting the numbers

Table 3.6: Column "In nontriv. SCC" shows objects in non-trivial SCCs (in percent of allocated objects). Columns "Same SCC" and "Same WCC" show pairs of objects with the same deathtime (in percent of pairs of objects in the same SCC or WCC, respectively). The numbers in parentheses count only objects where the owner is the application.

Benchmark	In nontriv. SCC		Same SCC		Same WCC	
null	13.0	(n/a)	99.8	(n/a)	95.4	(n/a)
compress	9.8	(13.5)	99.4	(100.0)	25.7	(64.6)
db	0.6	(0.0)	99.5	(100.0)	2.2	(2.1)
jack	0.4	(0.0)	99.0	(100.0)	0.5	(0.3)
javac	15.1	(19.0)	34.1	(34.0)	1.3	(2.6)
jess	0.7	(0.0)	94.9	(19.3)	0.2	(0.3)
mpegaudio	8.7	(8.2)	99.1	(100.0)	10.4	(99.1)
mtrt	0.6	(0.2)	99.5	(100.0)	21.8	(23.1)
bh	1.4	(0.0)	99.6	(100.0)	28.5	(5.4)
bisort	8.0	(0.0)	99.8	(n/a)	87.7	(100.0)
em3d	19.7	(74.9)	99.8	(100.0)	71.5	(100.0)
health	14.4	(14.6)	46.7	(46.3)	6.2	(4.7)
mst	13.9	(50.9)	99.8	(100.0)	76.8	(100.0)
perimeter	78.8	(100.0)	100.0	(100.0)	96.6	(100.0)
power	1.7	(0.0)	99.7	(n/a)	64.1	(100.0)
treeadd	1.2	(0.0)	99.8	(n/a)	99.9	(100.0)
tsp	25.7	(33.3)	100.0	(100.0)	87.3	(100.0)
voronoi	37.0	(91.5)	42.6	(39.2)	56.6	(38.0)
ipsixql	46.9	(56.5)	1.3	(1.3)	1.3	(1.3)
jigsaw	5.0	(3.1)	81.6	(63.8)	1.2	(3.2)
nfc	9.6	(10.8)	1.9	(0.8)	1.5	(0.8)
xalan	2.6	(2.0)	99.0	(98.6)	9.5	(26.0)
average	14.4	(22.8)	80.8	(72.4)	35.8	(46.3)

Table 3.7: Over-estimation of numbers in Tables 3.5 and 3.6 due to granulated traces. The numbers count only objects where the owner is the application.

Benchmark	Any pair	$o_1 \to o_2$	$o_1 \to o_2$, o_1 mutated	Same SCC	Same WCC
compress	7.7	7.2	3.6	0.0	1.9
db	1.3	16.0	−1.0	0.0	1.3
jack	0.3	7.9	3.6	0.0	−0.3
javac	0.4	23.2	27.7	0.7	0.5
jess	−0.3	43.3	14.0	18.3	−0.3
mpegaudio	6.0	4.5	1.8	0.0	0.6
bh	2.3	1.4	6.5	0.0	5.1
bisort	0.0	0.0	0.0	n/a	0.0
em3d	0.2	2.4	100.0	0.0	−0.1
health	2.2	5.4	3.0	0.5	1.9
mst	2.1	2.2	5.8	0.0	2.0
perimeter	0.0	33.3	n/a	0.0	0.0
power	2.7	0.0	n/a	n/a	0.0
treeadd	0.0	0.0	n/a	n/a	0.0
tsp	4.6	0.0	0.0	0.0	0.0
voronoi	8.8	12.3	13.4	8.5	8.8
average	2.3	9.9	13.7	2.1	1.3

based on precise deathtime traces from the numbers based on granulated traces. Table 3.7 reports these differences for application objects only. Since Merlin cannot yet trace multithreaded programs, Table 3.7 does not contain the results for all the benchmarks. Merlin's inability to handle multithreaded programs is also the reason why this chapter does not use precise deathtime traces throughout.

As expected Table 3.7 shows that granulated traces inflate the same deathtime numbers, i.e. most entries are greater than zero. (Since the precise and granulated traces use different runs and different versions of the Jikes RVM, there is some noise in the data leading to a few negative numbers.)

Table 3.8 juxtaposes the last rows of Tables 3.5, 3.6, and 3.7. It shows that even though the likelihood of two linked objects having the same deathtime is lower by 9.9% (on average) with precise traces than with granulated traces, the basic results still hold. In other words, the likelihood of linked objects or objects in the same SCC having the same deathtime is much higher than the likelihood of two random objects having the same deathtime.

3.3.4 CBGC-specific questions

While the previous sections presented results that are general, so they can motivate CBGC as well as other garbage collection techniques, this section is a preliminary

Table 3.8: (a) Average number of pairs of application objects with the same deathtime based on granulated traces (last rows in Tables 3.5 and 3.6); (b) Average over-estimation in these numbers (last row in Table 3.7); (c) Difference.

	Any pair	$o_1 \to o_2$	$o_1 \to o_2$, o_1 mutated	Same SCC	Same WCC
(a)	33.1	76.4	61.1	72.4	46.3
(b)	2.3	9.9	13.7	2.1	1.3
(c)	30.8	66.5	47.4	70.3	45.0

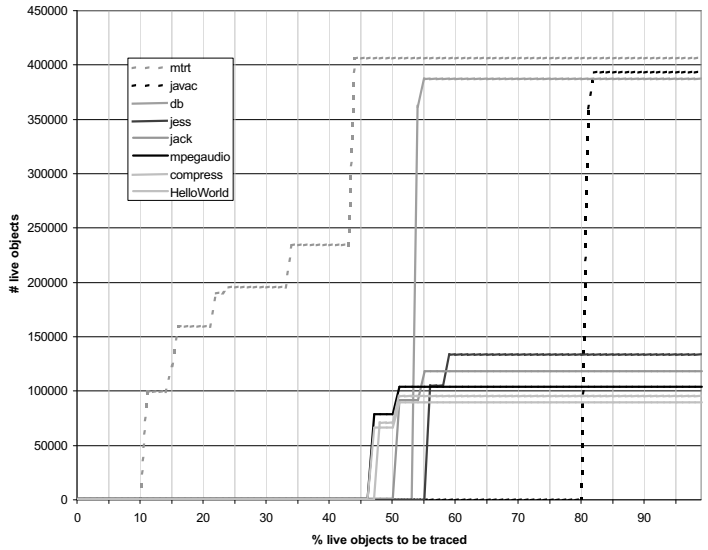
investigation of questions specific to CBGC.

Connectivity-based garbage collection avoids write barrier overhead even for partial collections, whereas partial collections in a generational collector use potentially write barriers. Write barriers are often expensive: in Table 3.4, the average mean of the write barrier overhead is 7.6%, and in [19, Table 2], the geometric mean of the write barrier overhead is 3.2%. Prior work confirms these findings [123]. Fitzgerald and Tarditi [53] report experiments where generational collectors "... did poorly on benchmarks that had low collection costs and high write barrier costs. For those benchmarks, the cost of the write barrier was higher than the reduction in collection cost".

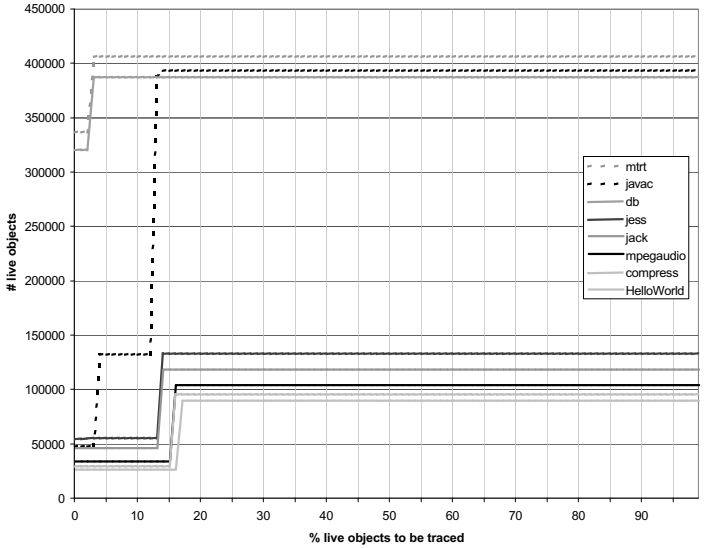
For connectivity-based garbage collection to perform well, two properties must hold. First, for most partitions, the number of objects in their ancestor partitions must be small, otherwise the collector will end up having to collect a good part of the heap at every partial collection. Second, the objects that are close to the roots should be the most profitable to collect since they are the easiest to collect. The following data explores these requirements.

Figure 3.7(a) gives the number of objects in the ancestors of the partition of each object in a benchmark program, where the partitioning is based on a coarse type-based analysis [65]. This graph weighs the partition dag with the live objects in each partition at a particular snapshot in program execution. A point (x, y) in Figure 3.7(a) means that y objects are in partitions whose ancestor sets have sizes of at most x% of all live objects. It turns out that most objects require the garbage collector to look at about 59% of the total objects at that point. These numbers are high because they are based on a partitioning using declared types, and because Java bytecodes do not support generic types and thus container data structures have fields of type Object (and can therefore point to all objects). Stronger static analysis (or a language with generic types) may yield better results.

Figure 3.7(b) presents the same graph as Figure 3.7(a) except that it uses an optimal partitioning: for each object, it reports how many other objects reach it in the snapshot used for Figure 3.7(a). Figure 3.7(b) thus gives an upper bound on the quality of partitioning with a stronger analysis than type-based analysis. Figure 3.7(b) shows that at least in the optimal scenario, all objects can be garbage collected by examining only about 18% of the objects.



(a) Number y of objects that are reachable from at most x% of the live objects, where reachability is based on static type information.



(b) Number y of objects that are reachable from at most x% of the live objects, where reachability is based on the actual pointers in the heap.

Figure 3.7: Reachability in snapshot of heap.

Figures 3.7(a) and (b) present data measured on snapshot object graphs. This is equivalent to sampling the objects that happen to be alive at one particular point in time; such a sample will overemphasize longlived, quasi immortal, and truly immortal objects. Thus, it may be the case that for shortlived objects, a garbage collector may need to look at many fewer objects than 59% (in the case of the Harris partitioning) or 18% (in the case of the optimal partitioning).

Figure 3.8 shows the average number of objects from which each object can be reached in the global object graph. Figure 3.8(a) presents data for all objects and Figure 3.8(b) presents data for application objects only. The length of the bars is the average number of objects with a path to an object on a logarithmic scale. There are four bars for each benchmark, one per lifetime bin. The bars for shortlived objects are usually the shortest (Section 3.3.2 explains the exceptional behavior of benchmark ipsixql). That is encouraging because it means that to garbage-collect shortlived objects, the collector does not have to look at too many other objects. This data also suggests that Figures 3.7(a) and (b) are overly pessimistic, since they are based on snapshots which will be biased towards longer-lived objects.

3.4 Related work

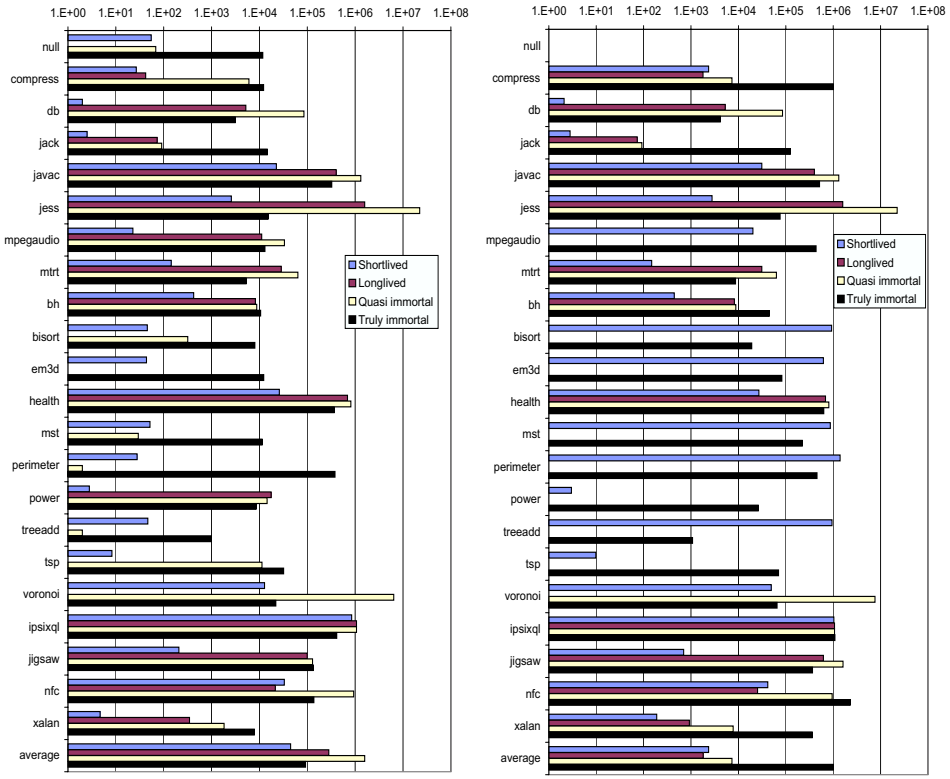
This section summarizes related work on understanding object behavior, generational garbage collection, other relevant memory management schemes, and escape analysis.

3.4.1 Understanding object behavior

Barry Hayes described and tested the weak and strong generational hypotheses [67]. The weak generational hypothesis states that "newly-created objects have a much lower survival rate than older objects" [67]. The strong generational hypothesis states that "even if the objects in question are not newly created, the relatively younger objects have a lower survival rate than the relatively older objects" [67]. He found that even though the weak generational hypothesis is often true, the strong generational hypothesis is usually false. He goes on to describe key object opportunism, where the assumption is that connected objects die together and this can be exploited by collecting a data structure when its root dies. This chapter provides supporting evidence for this claim and explores the correlation of different kinds of connectivity with lifetime.

Stefanović and Moss [119] explore the age distribution of objects. They collect their data by garbage collecting frequently. Unlike the work in this chapter, Stefanović and Moss do not empirically relate age behavior to connectivity.

Dieckmann and Hölzle [46] measure the distribution of object lifetimes, sizes, and types and the reference density (fraction of fields that contain pointers) for the SPECjvm98 benchmarks. They focus on traits inherent in individual objects, whereas this chapter studies connectivity between various objects and how it correlates with lifetime.



(a) All objects

(b) Application objects

Figure 3.8: Average number of objects from which each object can be reached in the global object graph.

Shuf *et al.* [112] study the cache and TLB behavior of the SPECjvm98 benchmarks and pBOB. They use Jikes RVM to trace high-level heap accesses and then use a simulator to correlate cache and TLB misses with object sizes and layouts.

3.4.2 Generational garbage collection

The results in this chapter point at several possibilities for improving generational garbage collection. For example, the results suggest that moving an object near its connected objects is often a good idea since connected objects have a similar deathtime. Thus, connectivity information may give the benefits of pretenuring and locality optimizations without requiring profile information. Guyer and McKinley found a way to apply this idea in practice [63].

3.4.3 Escape analysis

If the lifetime of a data structure ends before the routine that allocated it returns and the size of the data structure is bounded, it can be allocated on the stack instead of the heap. Analyses that try to determine these properties of objects are called escape analyses [36, 56, 106, 122, 132]. Some escape analyses focus on objects that escape a thread (e.g., [115]). The escape behavior numbers in this chapter help judge the potential benefit of escape analyses for garbage collection.

3.5 Conclusions

This chapter explores object connectivity and its relationship with object deathtime and lifetime. It classifies connectivity into six categories: immediate or transitive stack-, globals-, and heap-connectivity.

The results demonstrate that some kinds of connectivity correlate strongly with object deathtime or lifetime:

- Objects that are reachable only immediately from the stack are usually short-lived.
- Objects that are reachable transitively from globals are usually quasi immortal or truly immortal.
- Objects that are connected via pointers (immediately or transitively) usually die at the same time.

Since the infrastructure (Jikes RVM) uses the same heap as the application, this chapter presented results for both all objects (including objects created on behalf of the Jikes RVM) and application objects (objects created on behalf of the application only).

The fact that connected objects tend to die together motivates partitioning objects by connectivity. Objects in a partition are connected, making it likely that they die at the same time. When the garbage collector can also estimate which partition contains objects that just died, it can perform less work to reclaim memory,

improving throughput. The observations on roots-connectivity help estimate which partitions contain the garbage.

Chapter 4

The Cbgc Algorithm Family

This chapter introduces the Cbgc family of garbage collection algorithms. To provide a better understanding of how Cbgc exploits the connectivity of heap objects, this chapter primarily focuses on an abstract Cbgc algorithm, leaving explicit details to later chapters.

4.1 Partitioning

Based upon conservative information about object connectivity, Cbgc divides the set of heap objects, O, into a set of disjoint partitions, P. A partitioning of the objects, $(\mathit{partition}, P, E)$, consists of a partition map $\mathit{partition} : O \rightarrow P$ and a partition dag[p.176] (P, E). The partition map *partition* associates each object $o \in O$ with its partition $\mathit{partition}(o) \in P$. The edges E of the partition dag represent the may-point-to relations.

A partitioning $(\mathit{partition}, P, E)$ for Cbgc must be conservative and stable. In a conservative partitioning, if a pointer may exist between two heap objects, then either the objects must be in the same partition or there must exist an edge between their partitions in the partition dag. Equation (4.1) formalizes this definition of conservatism.

$$\mathit{pointsTo}(o_1, o_2) \Rightarrow \left(\begin{array}{ll} \mathit{partition}(o_1) = \mathit{partition}(o_2) & \vee \\ (\mathit{partition}(o_1), \mathit{partition}(o_2)) \in E & \end{array} \right) \tag{4.1}$$

In a stable partitioning, two objects that belong to the same partition at one point in time must belong to the same partition ever afterward. Thus Cbgc cannot split partitions. More specifically, when (as is common in Java) a class is dynamically loaded, a connectivity analysis of the class may cause Cbgc to add new partitions or merge existing partitions, but never to divide existing partitions.

Figure 4.1 gives an example partitioning. Solid boxes are objects, solid arrows are pointers, dashed ovals are partitions, and dashed arrows are partition edges. Thus $\mathit{partition}(o)$, the partition map, is implicitly defined by graphic inclusion, e.g. $\mathit{partition}(o_1) = \mathit{partition}(o_2) = p_1$. Because there are no cycles of partition edges, the partitions form a dag. This partitioning is conservative because there exists an

edge between the corresponding partitions wherever a pointer crosses the partition boundaries. Because of possible weaknesses in the program analysis, the reverse does not have to hold. For example there is an edge (p_2, p_3) even though no object in p_2 points to an object in p_3.

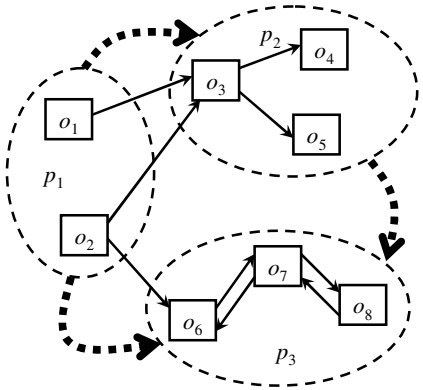


Figure 4.1: Example partitioning. Solid boxes are objects, solid arrows are pointers, dashed ovals are partitions, and dashed arrows are partition edges.

While this section gave an abstract overview of what a partitioning is, Chapter 5 is devoted to giving details of the many concrete ways for coming up with a partitioning.

4.2 Partial garbage collection

A partial garbage collection[p.188] is a GC of only a portion of the heap, such as when a generational collector collects only one or more young generations[p.190]. Because they do not examine the entire heap, partial GCs usually take less time than full GCs, improving responsiveness. A partial GC ideally focuses on the parts of the heap where it can reclaim a lot of garbage at low cost, thereby improving program throughput. CBGC normally performs only partial GCs and performs full GCs only in pathological cases.

While this section gives an abstract overview of what partial garbage collection in CBGC is, Chapter 8 is devoted to giving details of the many concrete ways for implementing it.

The algorithm in Figure 4.2 shows how CBGC performs partial collections based on the tricolor abstraction[p.183]. It is a generalization of the abstract full garbage collection algorithm from Figure A.6. Note that the algorithm does not rely on write barriers[p.189] to perform partial collections.


```
 1: estimate how many objects are dead and live in each partition
 2: choose a set C of partitions
                    Partial garbage collection
 3: for each root v
 4:     if v ≠ null and partition(target(v)) ∈ C
 5:         color(target(v)) ← gray
 6: for each chosen partition p ∈ C in topological order
 7:     while there are gray objects in p
 8:         o ← pickGray(p)
 9:         color(o) ← black
10:         for each field f of o
11:             if o.f ≠ null and partition(target(o.f)) ∈ C
12:                 if color(target(o.f)) = white
13:                     color(target(o.f)) ← gray
14:     for each heap object o with partition(o) = p
15:         if color(o) = white, reclaim its memory
16:         else color(o) ← white
```

Figure 4.2: Abstract connectivity-based stop-the-world garbage collector.

4.2.1 The algorithm in detail

This section explains the CBGC algorithm from Figure 4.2, which is central to this dissertation.

Section 4.3 discusses Lines 1 and 2 of the algorithm from Figure 4.2; for now, the only thing one needs to know about them to understand the subsequent lines is that the chosen set is closed under the predecessor relation. In other words, for each partition q in the chosen set $C \subseteq P$ of partitions, all its predecessors are chosen as well ($q \in C \wedge (p, q) \in E \Rightarrow p \in C$).

At the beginning of Line 3, all objects in C are white. Nothing has been visited yet. The reachability traversal will only be concerned with objects in C, hence, it does not care about the color of objects in the rest of the heap.

Lines 3 to 5 scan the program roots[p.179]. They are the equivalent of Lines 1 and 2 in Figure A.6. For each pointer in a stack or global variable that points to a white object o with $partition(o) \in C$, they color that object gray. After that, all objects that are reachable from a root are reachable from a gray object.

The outer loop of Lines 6 to 16 visits each partition $p \in C$ in topological order. Since Line 2 ensures that C is closed under the predecessor relation, the topological order guarantees that at the loop iteration for a partition, all of its predecessor partitions have already been visited.

Lines 7 to 13 of the algorithm from Figure 4.2 finish the reachability traversal[p.186] for the current partition p. They are the equivalent of Lines 3 to 9 in Figure A.6. At the end of Lines 7 to 13, all reachable objects in p are black, and all unreachable objects in p are white.

Lines 14 to 16 of the algorithm from Figure 4.2 are the reclamation phase[p.186]

for the current partition. Since they happen before the next outer loop iteration processes the next partition, CBGC performs early reclamation[p.186]. That means that in the case of copying CBGC, the memory reclaimed for one partition is already available when the reachability traversal of the next partition needs a place to copy objects into.

4.2.2 Correctness proof

Lemma 1 *Line 15 of the algorithm from Figure 4.2 reclaims exactly the unreachable objects in partition p.*

Proof.

i. *After Lines 7 to 13, if an object o_n in partition p is reachable from the roots via objects in p or any predecessor partitions of p, then it is black, otherwise white.* To see this, suppose root r reaches object o_n via the pointer chain $r \to o_1 \to \ldots \to o_n$. Per induction hypothesis, the prefix $r \to o_1 \to \ldots \to o_i$ residing in predecessor partitions of p is black. Because of the tricolor abstraction, o_{i+1} is gray, and Line 9 makes $o_{i+1} \ldots o_n$ black, since they reside in p.

ii. *If an object o_n in partition $p = partition(o_n)$ is reachable, then it is reachable via objects in p or in predecessor partitions of p.* Suppose o_n is reachable via the pointer chain $r \to o_1 \to \ldots \to o_n$. Let $r \to p_1 \to \ldots \to p_n$ be the corresponding partitions, where $p_n = p$. Then conservatism (Equation (4.1)) guarantees that $p_1 \ldots p_{n-1}$ are predecessors of p or the same as p.

Parts i. and ii. together show that after Lines 7 to 13, exactly the unreachable objects in partition p are white. Line 15 reclaims them. □

A corollary of Lemma 1 is that a partial GC reclaims exactly the unreachable objects in the chosen set of partitions $C \subseteq P$.

4.3 Opportunism

Chapter 3 showed that connected objects die together and that lifetime and connectivity are related. CBGC exploits these properties by making some connectivity information explicit, allowing an opportunistic choice about where to collect. This is similar to how generational GC exploits the hypothesis that young objects die quickly by making some age information explicit.

When CBGC performs a partial GC, it first chooses a set of partitions to collect, in Lines 1 and 2 of the algorithm from Figure 4.2. CBGC uses two functions in making this choice: the estimator estimates how many objects are dead and live in each partition (Line 1), and the chooser chooses a set of partitions where, based on the estimates, it expects to collect a sufficient amount of garbage at low cost (Line 2).

The task of the estimator is to annotate each partition $p \in P$ with two integers $dead(p)$ and $live(p)$ as shown in Figure 4.3.

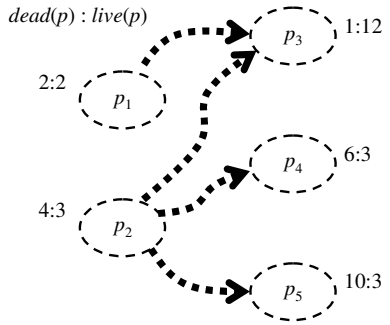


Figure 4.3: Example partition dag annotated by the estimator.

After this annotation, the chooser opportunistically chooses a set C of partitions that is closed under the predecessor relation. In Figure 4.3, it might choose $C = \{p_2, p_5\}$, because they have the best ratio of total dead to live objects $\sum dead / \sum live = (4+10)/(3+3) = 7/3$. The ratio 10/3 for the individual partition p_5 would be better, but the set $\{p_5\}$ is not closed and thus cannot be independently collected.

While this section gave an abstract overview of what the chooser and the estimator have to do, Chapter 7 describes how concrete choosers work, and Chapter 6 describes how concrete estimators work.

4.4 Related work

Jones and Lins [86] and Wilson [134] provide good introductions to garbage collection and also describe many techniques and algorithms that inspired CBGC.

4.4.1 Eliminating write barriers

CBGC does partial GC without any write barriers. Shuf *et al.* describe a simple type-based partitioning that allows eliminating some write barriers [110]. Zee and Rinard have presented an analysis for eliminating some of the write barriers for generational GC [137]. We are not aware of any other work on eliminating write barriers for partial GC.

4.4.2 Partitioning

CBGC partitions heap objects by connectivity to improve locality (allocate connected objects together, since the mutator is likely to access them together), to enable opportunism (do GC where it gets high payoff with little effort), and to improve responsiveness (reduce pause times by avoiding full GC). Over the years, other partitioning techniques have been proposed to achieve these goals.

4.4.2.1 Age-based partitioning

Age-based garbage collectors (see Section A.10) partition objects by age. They assume that the survivor rate is related to object age, and exploit this by making opportunistic choices about where to do partial GC.

The weak generational hypothesis states that most objects die young, and thus generational collectors collect young objects most frequently [130]. While Appel flexibly adapts the boundary between young and old objects to achieve optimal memory usage [7], Barrett and Zorn adapt the boundary to achieve a variety of objectives [17].

Some researchers have found that the weak generational hypothesis does not allow the best opportunistic choices about where to collect. Pretenuring allocates objects that are expected to be long-lived directly into the old generation to avoid copying long-lived objects out of the nursery [34]. Older-first collection assumes that the very youngest objects are unlikely to be dead since they have not yet had time to die [117]. The Beltway collector generalizes existing copying age-based collectors by using a configurable partitioning [21].

Age-based collectors are well-studied and often successful, but there is reason to believe one should not rely solely on age, as it is not always a reliable predictor for when objects die. While this dissertation proposes using heap object connectivity as the main principle for guiding garbage collection, it may also be beneficial to use CBGC just for the old generation of a generational collector.

4.4.2.2 Stack-based partitioning

Stack-based techniques partition objects by the stack frames that allocated them. They assume that the lifetime of objects is related to the time at which the stack frame gets popped, and exploit this by opportunistically deallocating objects together with stack frames if possible.

Some stack-based techniques rely on static information. Stack allocation is based on escape analysis and allocates the objects directly on the stack [100]. In region allocation (see Figure 1.1), regions are allocated and deallocated at statically predefined program points following a stack discipline, and individual objects are allocated into their region, but deallocated only when the entire region is deallocated [55, 129].

Other stack-based techniques rely on dynamic checks. Contaminated GC tracks the lowest stack frame from which an object is reachable, and only deallocates the object when that stack frame gets popped [28]. Qian and Hendren associate a region with each stack frame and track whether objects escape from it; if so, the region is merged into the global region and handled by conventional GC, otherwise it is reclaimed en-masse when the stack frame is popped [102].

Stack-based techniques may work well for functional languages, but they often require hand-tuning by the programmer. Section 1.2 argues that the job of automatic memory management should be to relieve the programmer from having to pay much attention to memory. Also, large object-oriented programs are unlikely to obey a strict stack discipline.

4.4.2.3 Other partitioning techniques

Age-based and stack-based partitioning are the most popular techniques for improving locality and allowing partial, opportunistic GC. But this dissertation is not the only example of seeking other partitionings to achieve this goal.

Some techniques use static analyses. Dolby and Chien analyze ownership relations between objects. If they find a relation between two objects such that the owned object has a fixed size and dies before the owner, they inline it into the owner [49]. Steensgaard describes thread-specific heaps for objects that are not shared among multiple threads [115]. Other techniques rely on profile information. Seidl and Zorn partition heap objects by the number of dynamic references to them and by their expected lifetime [108], and Shuf *et al.* partition heap objects by whether their type has many instances or not [110, 111].

As early as 1991, Barry Hayes envisioned key object opportunism, which observes when a key object dies and opportunistically reclaims the objects connected to it [67]. Key object opportunism relies on the hypothesis that connected objects die together, and Chapter 3 provides evidence that supports this. To implement key object opportunism, a GC needs to be aware of connectivity, hence CBGC is a step closer to making Hayes's vision a reality.

4.5 Discussion

CBGC has some inherent advantages over other collectors that perform partial GC. In CBGC's partial GC, the uncollected remaining partitions, R, do not affect the reachability of objects in the portion of the heap chosen for GC, C. Therefore, CBGC need not track pointers from R to C, eliminating the need for a write barrier. CBGC also does not suffer from nepotism[p.189] (nepotism is when a dead object in R falsely keeps an object in C live). Furthermore, CBGC allows early reclamation[p.186] in that some objects in C can be reclaimed even before all of C is collected (Harris used a similar approach for early reclamation during full GC [65]). Early reclamation means that when collecting partition p, the memory reclaimed earlier during the collection of p's predecessors can already be reused. Finally, except for the degenerate case where one of the partitions is reachable from all other partitions, CBGC can collect all heap objects without ever performing a full GC. This substantially improves upon most other garbage collectors that require occasional full GCs for completeness[p.189] (the MOS collector is an exception [85]).

In addition to the above, CBGC may make better opportunistic choices and deliver better locality than existing collectors.

Chapter 5

Partitioning

CBGC relies on a partitioning of the set of heap objects, O, based on a conservative approximation of their connectivity. A partitioning $(\mathit{partition}, P, E_P)$ consists of a partition map $\mathit{partition} : O \to P$ and a partition dag (P, E_P). The partition map associates each object $o \in O$ with its partition $\mathit{partition}(o) \in P$. The edges of the partition dag represent may-point-to relations.

There are many ways to come up with a partitioning $(\mathit{partition}, P, E_P)$ for CBGC. The simplest possible partitioning is to have only one partition, in which case CBGC degenerates to full GC[p.189]. This chapter explores a variety of partitionings that allow CBGC to do true partial GC[p.188].

This chapter considers only partitionings that are based on strongly connected components of some form of may-point-to graph. Using finer-grained partitions than SCCs would require some changes to the framework, e.g. introducing write barriers[p.189]. Using coarser-grained partitions would not require any changes to the framework, but would reduce the flexibility for partial garbage collections.

Section 5.1 defines a way for comparing partitionings. Section 5.2 presents invariants that a partitioning for CBGC must satisfy. Section 5.3 explores how several compiler analyses can yield partitionings that satisfy those invariants. Section 5.4 introduces two oracular partitionings to evaluate realistic partitionings against.

5.1 Granularity

This section defines the concepts of granularity for partitionings: given two partitionings, one may be more fine-grained, and the other more coarse-grained. Intuitively, a partitioning that is more fine-grained has smaller partitions and fewer edges. Objects that belong to the same partition in a more coarse-grained partitioning may belong to different partitions in a more fine-grained partitioning. Edges in the more coarse-grained partitioning may be missing from the more fine-grained partitioning.

In general, a more fine-grained partitioning is harder to find, but leads to better performance of the connectivity-based garbage collector. It is harder to find because it requires a more precise compiler analysis, which is both more difficult to develop, and more expensive to run. It improves the performance of a connectivity-based

garbage collector out of several reasons. In a more fine-grained partitioning, the lifetime distributions within individual partitions are more skewed; if the estimator can predict the skew, that leads to better throughput. Also, with smaller partitions, a partial collection takes less time, and that leads to better responsiveness. Finally, having fewer edges between partitions means that it is easier to collect partitions independently from each other, improving both throughput and responsiveness.

Formally, a partitioning $(\mathit{partition}_{\text{finer}}, P_{\text{finer}}, E_{\text{finer}})$ is more fine-grained than a partitioning $(\mathit{partition}_{\text{coarser}}, P_{\text{coarser}}, E_{\text{coarser}})$ if there is a function $\mathit{coarsening} : P_{\text{finer}} \to P_{\text{coarser}}$ such that the following holds.

- The function *coarsening* is defined for all partitions in the more fine-grained partitioning.
$$\mathit{domain}(\mathit{coarsening}) = P_{\text{finer}} \tag{5.1}$$
- The more coarse-grained partitioning has at least the edges corresponding to edges in the more fine-grained partitioning, perhaps more.
$$(p, q) \in E_{\text{finer}} \Rightarrow \left(\begin{array}{l} \mathit{coarsening}(p) = \mathit{coarsening}(q) \quad \vee \\ (\mathit{coarsening}(p), \mathit{coarsening}(q)) \in E_{\text{coarser}} \end{array} \right) \tag{5.2}$$
- The partition map of the more coarse-grained partitioning is the composition of the function *coarsening* with the partition map of the more fine-grained partitioning.
$$\mathit{partition}_{\text{coarser}} = \mathit{coarsening} \circ \mathit{partition}_{\text{finer}} \tag{5.3}$$

Intuitively, Equation 5.3 means that the more fine-grained partitioning must have at least as many partitions as the more coarse-grained partitioning. It has two simple corollaries:

- The set of objects in a partition $p_c \in P_{\text{coarser}}$ of the more coarse-grained partitioning is the union of the sets of objects of all partitions $p_f \in P_{\text{finer}}$ of the more fine-grained partitioning that map to p_c.
$$\{o \mid p_c = \mathit{partition}_{\text{coarser}}(o)\} = \bigcup_{p_c = \mathit{coarsening}(p_f)} \{o \mid p_f = \mathit{partition}_{\text{finer}}(o)\} \tag{5.4}$$
- If two objects belong to the same partition $p_f \in P_{\text{finer}}$ of the more fine-grained partitioning, then they also belong to the same partition $p_c \in P_{\text{coarser}}$ of the more coarse-grained partitioning.
$$\mathit{partition}_{\text{finer}}(o_1) = \mathit{partition}_{\text{finer}}(o_2) \Rightarrow \mathit{partition}_{\text{coarser}}(o_1) = \mathit{partition}_{\text{coarser}}(o_2) \tag{5.5}$$

The concept of partitioning granularity yields a partial order on partitions, but not a total order. Given two partitions, it may be that one of them is more fine-grained than the other; but it may also be that neither it more fine-grained, because

their granularity is incomparable. But even though it yields only a partial order, granularity is useful for defining how a partitioning may evolve over time in the presence of dynamic class loading (Section 5.2.3), and to compare partitionings resulting from different compiler analyses to oracular partitionings (Section 5.4).

5.2 Invariants

CBGC relies on a compiler analysis to come up with a partitioning. This section describes what invariants such a partitioning must satisfy. This section honors the important issues of dynamic class loading, the native code interface, and reflection. The intent is to have a formulation of the invariants that is precise enough to validate proposed analyses against them. This section also motivates each individual invariant; it it conceivable that an analysis does not quite fulfill one of the invariants, and in that case it is important to make realistic tradeoffs for weakening it.

A partitioning $(partition, P, E_P)$ for CBGC must satisfy three invariants: acyclicity, conservatism, and stability. Sections 5.2.1, 5.2.2, and 5.2.3 formalize and motivate these invariants.

5.2.1 Acyclicity

The directed graph of partitions, (P, E_P), must be a acyclic, hence the name partition dag[p.176].

$$\neg\Big((p \rightarrow^+ q) \wedge (q \rightarrow^+ p)\Big) \tag{5.6}$$

Motivation

When CBGC collects a partition $p \in P$, it collects all of its predecessors along with it. If there were cycles of partitions, they would all be predecessors of each other, and thus be collected together. That means that one can collapse the cycle to just one partition without reducing the number of choices for which partitions to collect. In addition, acyclicity establishes a well-defined topological order, which the CBGC algorithm (Figure 4.2) uses in Line 6; the topological order guarantees the correctness of the early reclamation in Line 15. Finally, collapsing cycles reduces the number of partitions, which in turn speeds up algorithms that operate on the partition dag, and may also reduce fragmentation in the memory representing partitions. Therefore, this dissertation demands acyclicity for CBGC partitionings.

5.2.2 Conservatism

If there may be a pointer between two heap objects, either they must be in the same partition or there must exist an edge between their partitions in the partition dag.

$$pointsTo(o_1, o_2) \Rightarrow \left(\begin{array}{ll} partition(o_1) = partition(o_2) & \vee \\ (partition(o_1), partition(o_2)) \in E_P & \end{array} \right) \tag{5.7}$$

Motivation

To do a partial GC, CBGC chooses a set $C \subseteq P$ of partitions to collect. It chooses C such that it is closed under the predecessor relation, i.e. for each chosen partition, all its predecessors are chosen as well ($q \in C \wedge (p, q) \in E \Rightarrow p \in C$). Let $R = P \setminus C$ be the remaining partitions that have not been chosen. The correctness of CBGC's partial GC relies on the fact that no objects in R have pointers to any objects in C (see the proof of Lemma 1). This is guaranteed by conservatism, so this dissertation demands conservatism for CBGC partitionings.

5.2.3 Stability

Stability constrains how partitionings may change over time. A partitioning at an earlier point in time must be more fine-grained than the partitioning at a later point in time, where granularity is defined as in Section 5.1. In the context of the stability invariant, the three granularity equations mean the following:

- Equation (5.1): Partitions are never removed. If a partition exists at one point in time, it either stays or is merged with other partitions.
- Equation (5.2): Edges are never removed. If there is an edge between two partitions at one point in time, there is an edge between these partitions as long as they are not merged.
- Equation (5.3) and its corollaries (5.4,5.5): Partitions are never split. If two objects belong to the same partition at one point in time, they belong to the same partition forever after.

Stability is trivially satisfied if the partition map and the partition dag never change. Stability also admits adding partitions, adding edges, and merging partitions, just not removing partitions, removing edges, or splitting partitions.

Motivation

Java has a few features that make it hard to analyze, such as dynamic class loading, the native code interface, and reflection. The goal of this dissertation is to accept these challenges and design CBGC so that it deals with them correctly. Therefore, instead of just restricting partitionings to never change, this dissertation allows adding partitions, adding edges, and merging partitions.

The goal is to allow flexible and efficient algorithms for CBGC even in the face of Java's challenging features. Removing partitions, removing edges, and splitting partitions may be expensive at runtime, so these actions are forbidden. Fortunately, CBGC can deal with Java's challenging features despite of these restrictions.

- Why is adding partitions allowed?

  Assume the JIT compiler compiles a new allocation site or a new class, and that allocation site or class allocates new objects. In general, these new objects

have different connectivity properties than existing objects. If that is the case, the system should allocate them into a new partition to increase the number of choices for which partitions to collect at future partial GCs.

It is easy to create the bookkeeping data structures for a new partition, so that new objects can be allocated into that partition and the partition can be a candidate when choosing partitions for partial GC.

- Why is adding edges allowed?

  Assume the JIT compiler compiles a new pointer assignment, or native code or reflection performs a pointer assignment. The assignment may result in a pointer $o_1 \to o_2$ that would violate conservatism. If that is the case, the system should introduce a new edge $(\mathit{partition}(o_1), \mathit{partition}(o_2))$ to repair conservatism.

  It is easy to create the representation for a new edge, so that doing partial GC on a set of partitions that is closed under the predecessor relation will be safe.

- Why is merging partitions allowed?

  Assume the JIT compiler compiles a new pointer assignment, or native code or reflection performs a pointer assignment. If the assignment would result in a pointer $o_2 \to o_1$ in the opposite direction of an existing path of edges from $\mathit{partition}(o_1)$ to $\mathit{partition}(o_2)$, introducing an edge $(\mathit{partition}(o_2), \mathit{partition}(o_1))$ would violate acyclicity. If that is the case, the system should merge all involved partitions to repair acyclicity.

  Merging two partitions involves changing the partition map for the objects in at least one of them. Changing the partition map for objects that have not yet been allocated is easy: the allocation routine evaluates the map with some data structure, so the system just changes the entry into that data structure.

  Changing the partition map for all the objects that have already been allocated into a partition p may appear difficult at first glance. However, the system can use a two-level data structure, where the object is first mapped to a group of objects belonging to the same partition, and that group is in turn mapped to the partition. So the system again just changes the entry into the second level of the data structure.

  For example, objects may be segregated by partition into blocks. Each block represents a group of objects, and a side array maps each block to a partition. When the system merges two partitions, it updates the entries in the side array to point to the new representative.

- Why is removing partitions forbidden?

  Removing a partition does not only require proving that no new objects will be allocated into that partition, but also that the partition contains no existing live objects. Showing that the partition contains no live objects may require a garbage collection, which is expensive.

- Why is removing edges forbidden?

  Removing an edge does not only require proving that no new pointers will be installed between the partitions in question, but also that there are no existing pointers between them. Showing that there are no existing pointers between them may require a garbage collection, which is expensive.

- Why is splitting partitions forbidden?

  Splitting a partition does not only require proving that no new pointers will be installed that make the contained objects cyclic, but also that they are not connected in a cycle by existing pointers. Showing that they do not form a cycle may require a garbage collection, which is expensive. In addition, the collector would have to update the partition map for the objects in the split partitions in a way that is more difficult than for merging partitions.

5.3 Using compiler analysis

This section describes how to use a compiler analysis of the code of a program to come up with a partitioning of the objects allocated by the program that satisfies the invariants from Section 5.2. The literature proposes many analyses whose results may be postprocessed to yield CBGC partitionings; most of these analyses have been invented for different purposes. This section reviews some of them, and demonstrates how to post-process their results.

Let O be the set of objects. This section proposes composing the partition map $\mathit{partition} : O \to P$ of a partitioning $(\mathit{partition}, P, E_P)$ from two mapping functions $m_1 \circ m_2$, where $m_1 : O \to N$ and $m_2 : N \to P$. The set N of nodes is chosen in such a way that evaluating m_1 at allocation time is trivial, and may even be hardcoded into the compiled allocation sequence.

This section proposes using a compiler analysis to find a directed graph (N, E_N), where each node $n \in N$ represents a set of objects, and an edge (n_1, n_2) represents may-point-to relations. Then, the partition dag (P, E_P) is just the dag of strongly connected components of (N, E_N), found by a simple depth-first search on that graph. The second part $m_2 : N \to P$ of the partition map is the mapping from a node to its SCC, which is a by-product of the depth-first search. A connectivity-based garbage collector stores the map m_2 in an efficient data structure, and consults it upon object allocation to determine which partition the new object belongs to.

5.3.1 Compiler analysis classification

Compiler analyses for Java fall into two categories: offline or online. Whereas online analyses are compiler analyses that can deal with dynamic class loading, reflection, and native code, offline analyses ignore those features, which means they do not handle the full Java language. Online analyses happen partly during JIT compilation[p.176], finding new results as new code is compiled, and may thus lead to changes in the partitioning. While the invariants from Section 5.2 are general enough to admit online analyses, most compiler analyses from the literature work

Table 5.1: Classification of compiler analyses. Column "Set N" indicates whether the nodes of the may-point-to graph (N, E_N) of postprocessed results are types or allocation sites. Column "Online" indicates a paper that demonstrated that the analysis can deal with dynamic class loading. Column "Section" refers to the section that describes how to postprocess the analysis results for CBGC.

Analysis		Set N	Online	Section
[65]	Harris's analysis	types	[65]	5.3.2
[126]	XTA	types	[103]	5.3.3
[48]	Diwan et al.'s analyses	types	–	5.3.4
[6]	Andersen's analysis	allocation sites	[79]	5.3.5
[114]	Steensgaard's analysis	allocation sites	–	5.3.6
[57]	Connection analysis	allocation sites	–	5.3.7
[89]	Data structure analysis	allocation sites	–	5.3.8

only offline. This section does not restrict itself to online analyses. Chapter 10 is dedicated to describing how to turn one particular offline analysis, Andersen's analysis, into an online analysis; many of the solutions from Chapter 10 are transferable to other analyses.

There are many compiler analyses in the literature that can come up with results that can be post-processed into a may-point-to graph (N, E_N). This chapter considers two categories: analyses where the set N is the set of types, and analyses where the set N is the set of allocation sites. The allocation site is the place in the source code that allocates the object; in the presence of inlining, an analysis can actually derive a more fine-grained set N of inlined allocation contexts. Since all objects allocated at an allocation site have the same type, but a type may be used by multiple allocation sites, the set of types can be viewed as a coarsening of the set of allocation sites.

Table 5.1 classifies various analyses that can yield partitionings for CBGC by whether they have been demonstrated to work online or not, and by whether they yield a may-points-to graph (N, E_N) where nodes are types or allocation sites.

CBGC can use a wide spread of analyses; Table 5.1 shows only a few examples. For an analysis to be useful for CBGC it has to find complete information of what all fields of all objects in the program may point to. This means the analysis has to analyze the whole program. Since the clients of escape analyses often require information for some objects only, most escape analyses are insufficient for CBGC.

The following sections describe each of the analyses from Table 5.1, and discuss how to post-process their results for CBGC. Later parts of the dissertation will refer back to Harris's analysis (Section 5.3.2) and Andersen's analysis (Section 5.3.5); readers who are less interested in compiler analyses may skip the other sections, and continue reading in Section 5.4.

5.3.2 Harris's analysis

Harris's analysis is based on inspecting the declared types of fields and the type hierarchy [65].

Harris's analysis constructs a directed graph (N, E_N) where nodes N are types, and edges E_N are may-point-to relationships. A type $t_1 \in N$ may point to a type $t_2 \in N$ if t_1 or one of t_1's supertypes has a field that may store objects of type t_2 or one of t_2's supertypes. Harris used this analysis for early reclamation in an incremental treadmill collector. This analysis does not need to inspect program statements, just type declarations. Thus, it trivially works with reflection and JNI.

Format of results

The results are a directed graph (N, E_N), where the nodes N are types, and the edges E_N are may-point-to relationships.

Postprocessing

To obtain a partitioning, collapse SCCs of the directed graph (N, E_N).

5.3.3 XTA

XTA, or extensible type analysis, is a subset constraint based analysis for type resolution [126].

Format of results

XTA finds a points-to set of types for each method and each field. Local variables of a method M can point to objects of any type in the set $pointsTo(M) = S_M$. Field x of all objects that declare that field can point to objects of any type in the set $pointsTo(x) = S_x$.

Postprocessing

One can turn the results of XTA into a type may-point-to graph (N, E_N) by looking at the points-to sets of fields. For each field x, for each subtype $t_1 \in N$ of the type that declares field x, for each element $t_2 \in S_x$, add an edge (t_1, t_2) to E_N.

To obtain a partitioning, collapse SCCs of the directed graph (N, E_N).

5.3.4 Diwan et al.'s analyses

Diwan, McKinley, and Moss propose a number of different type-based analyses for object-oriented languages [48].

Format of results

Let N be the set of types, and A be the set of access paths. An access path $a \in A$ is a pointer expression at a program point. The paper describes various analyses that, among other things, find the following mappings:

- $\mathit{Subtypes} : N \to 2^N$ maps each type $n \in N$ to the set $\mathit{Subtypes}(n) \subseteq N$ of its subtypes.
- $\mathit{Type} : A \to N$ maps each access path $a \in A$ to its declared type $\mathit{Type}(a) \in N$.
- $\mathit{TypeRefsTable} : N \to 2^N$ maps each type $n \in N$ to the set $\mathit{TypeRefsTable}(n) \subseteq N$ of types that may be referenced by an access path a of declared type $\mathit{Type}(a) = n$ (TM-TBAA).
- $\mathit{ResolvedTypes} : A \to 2^N$ maps each access path $a \in A$ to a set of types $\mathit{ResolvedTypes}(a) \subseteq N$ it may point to. The paper describes different analyses that find this kind of mapping for virtual method resolution (TPA, ITPA, TPA-TBAA, and ITPA-TBAA).

Postprocessing

There are different ways to use the above mappings to find the edges of may-point-to graphs (N, E_N) where the nodes N are types.

1. Initialize $E_N \leftarrow \emptyset$. For each field assignment $a_1.f \leftarrow a_2$ or array slot assignment $a_1[i] \leftarrow a_2$, for each $n_1 \in \mathit{Subtypes}(\mathit{Type}(a_1))$ and each $n_2 \in \mathit{Subtypes}(\mathit{Type}(a_2))$, add edge (n_1, n_2) to E_N.
2. Initialize $E_N \leftarrow \emptyset$. Let $\mathit{FieldTypes} : N \to 2^N$ map a type $n \in N$ to the set $\mathit{FieldTypes}(n) \subseteq N$ of declared types of its fields (for a class type) or to its components (for an array type). Then, for each type $n_1 \in N$, for each type $n_2 \in \mathit{TypeRefsTable}(\mathit{FieldTypes}(n_1))$, add edge (n_1, n_2) to E_N.
3. Initialize $E_N \leftarrow \emptyset$. For each field assignment $a_1.f \leftarrow a_2$ or array slot assignment $a_1[i] \leftarrow a_2$, for each $n_1 \in \mathit{ResolvedTypes}(a_1)$ and each $n_2 \in \mathit{ResolvedTypes}(a_2)$, add edge (n_1, n_2) to E_N.

To obtain a partitioning, collapse SCCs of the directed graph (N, E_N).

5.3.5 Andersen's analysis

Andersen's analysis is a flow-insensitive, context-insensitive, subset constraint based pointer analysis that works on the granularity of allocation sites [6].

Format of results

Some papers on pointer analysis for Java [79, 91, 92, 105, 133] describe how to perform Andersen's analysis on Java programs. The following description of the analysis results follows [79], which bases its analysis results terminology on [91].

Andersen's analysis computes points-to sets containing allocation sites. Points-to sets are attached to analysis entities called v-nodes and $h.f$-nodes. A v-node represents a variable, and an $h.f$-node represents a field. Let h be an allocation site, and let f be a field. Then $\mathit{pointsTo}(h.f)$ is the set of allocation sites of objects pointed to by the field f of objects allocated at allocation site h. For example, if h allocates an object o_1, and if $\mathit{pointsTo}(h.f)$ contains an allocation site that allocates an object o_2, then $o_1.f$ may point to o_2.

Postprocessing

Each h-node corresponds to a node in the set N of nodes. For each $h_1 \in N$, for each field f, for each element $h_2 \in \mathit{pointsTo}(h_1.f)$, add edge (h_1, h_2) to the set of edges E_N.

To obtain a partitioning, collapse SCCs of the directed graph (N, E_N).

5.3.6 Steensgaard's analysis

Steensgaard's analysis is a flow-insensitive, context-insensitive, unification based pointer analysis that works on the granularity of allocation sites [114].

Format of results

Some papers on pointer analysis for Java [91, 92] describe how to use Steensgaard's analysis for Java, and obtain results in the same format as for Andersen's analysis (Section 5.3.5).

Postprocessing

See Andersen's analysis (Section 5.3.5).

5.3.7 Connection analysis

Connection analysis is a flow-sensitive and context-sensitive analysis for finding whether pointer targets may be in the same WCC[p.176] of the object graph [57]. It works on the granularity of allocation sites.

Format of results

For each function f, let V_f be the set of variables that f operates on. For each program point in f, connection analysis computes a Boolean matrix $C : V_f \times V_f \to \{0, 1\}$. For a pair of variables $v_1, v_2 \in V_f$, the result $C[v_1, v_2] = 1$ indicates that the objects that v_1 and v_2 point to may be weakly connected. In other words,

$C[v_1, v_2] = 1$ iff *target*(v_1) and *target*(v_2) may be reachable from each other (ignoring the direction of heap pointers).

Postprocessing

Call a variable v an *avar* if it appears as the left-hand side of an allocation $v =$ **new**.... Call a variable v a *pvar* if it is involved in parameter or return value passing, i.e. it is used as a formal or actual parameter, receives a return value, or appears in a **return** statement. Without loss of generality, assume that all allocations assign to simple avars, and all parameter or return value passing happens through simple pvars.

Initialize a disjoint-set data structure with one singleton set for each avar or pvar. For all matrices C, for all pairs of variables v_1, v_2, if $C[v_1, v_2] = 1$ then merge the sets that v_1 and v_2 belong to.

Use the resulting disjoint-set data structure to get an undirected may-point-to graph (N, E_N) where the nodes N are allocation sites. To do that, for each avar v_1, for each avar v_2 in the same set, introduce an edge between the corresponding allocation sites (n_1, n_2). Note that since the graph is undirected, SCCs are the same as WCCs, and that the disjoint-set data structure already yields WCCs. Use those SCCs as partitions.

5.3.8 Data structure analysis

Data structure analysis is a flow-insensitive by context-insensitive analysis that discovers parts of the shape of data structures, and is used for segregated explicit memory management [89].

Format of results

The analysis is described based on SSA form. The following description is simplified for Java.

The analysis finds one data structure graph per function. There are scalar nodes (for stack and global pointers) and heap nodes (for heap-allocated objects). There are two kinds of heap nodes: allocation nodes (for objects allocated in this procedure or its callees) and shadow nodes (for objects allocated elsewhere). Each directed edge is a triple (*srcNode*, *field*, *tgtNode*). For uniformity, scalar nodes have one "field" for their contents, and array nodes have one "field" for their elements (i.e. array elements are not tracked separately).

Postprocessing

The goal is to construct a directed may-point-to graph (N, E_N) where the nodes N are allocation sites.

First obtain a mapping from shadow nodes to sets of allocation nodes by looking at all possible unifications at call-sites. Assuming a mapping from allocation heap

nodes to allocation sites, this yields a mapping $allocSites : H \to 2^N$ from heap nodes $h \in H$ to sets $allocSites(h) \subseteq N$ of allocation sites.

Initialize $E_N \leftarrow \emptyset$. For each data structure graph edge (*srcNode*, *field*, *tgtNode*) connecting heap nodes, each $n_1 \in allocSites($*srcNode*$)$ and $n_2 \in allocSites($*tgtNode*$)$, add edge (n_1, n_2) to E_N.

To obtain a partitioning, collapse SCCs of the directed graph (N, E_N).

5.4 Oracular partitionings

Section 5.3 demonstrates that CBGC can use a wide variety of compiler analyses to obtain a partitioning. Therefore, it is important to determine which of these analyses can lead to good CBGC performance. Chapter 9 performs limit studies for two broad categories of analyses: those that are based on types, and those that are based on allocation sites, see Table 5.1. The limit studies use partitionings obtained with oracular knowledge rather than with a compiler analysis.

By the definition of granularity in Section 5.1, the type-dynamic partitioning (Section 5.4.1) is the most fine-grained partitioning possible that is at least as coarse-grained as types; and the allocsite-dynamic partitioning (Section 5.4.2) is the most fine-grained partitioning possible that is at least as coarse-grained as allocation sites. Referring to Table 5.1, that means that the type-dynamic partitioning is more fine-grained than partitionings based on Harris's analysis, XTA, or Diwan et al.'s analyses. The allocsite-dynamic partitioning is more fine-grained than any of the analyses in Table 5.1. A partitioning that is more fine-grained usually leads to better CBGC performance than a partitioning that is less fine-grained.

Both the type-dynamic partitioning and the allocsite-dynamic partitioning require oracular knowledge. They can not be used for a realistic CBGC algorithm that has no knowledge of the future of the computation. This dissertation describes experiments where the future of the computation is known, since the garbage collector is driven by a trace that can be analyzed beforehand. Those experiments are limit-experiments investigating the hypothetical case of operating with the most fine-grained partitionings of a given kind; any real partitioning of that kind will be more coarse-grained than that.

5.4.1 Type-dynamic partitioning

The initial partitions in the type-dynamic partitioning are program object types (classes in Java terminology). If the trace of a benchmark run contains an assignment that creates a pointer from an object of one type to an object of another type, the type-dynamic partitioning adds a corresponding edge between the types. Finally, the type-dynamic partitioning collapses SCCs in the type may-point-to graph to form partitions.

5.4.2 Allocsite-dynamic partitioning

The initial partitions in the allocsite-dynamic partitioning are the allocation sites. If the trace of a benchmark run contains an assignment that creates a pointer from an object created at one allocation site to an object created at another allocation site, the allocsite-dynamic partitioning adds a corresponding edge between the allocation sites. Finally, the allocsite-dynamic partitioning collapses SCCs of the allocation site may-point-to graph to form partitions.

Chapter 6

Estimator

The estimator's job is to annotate each partition p with two numbers $\mathit{dead}(p)$ and $\mathit{live}(p)$, where $\mathit{dead}(p)$ is a guess for how much memory of dead objects a collection of p would reclaim, and $\mathit{live}(p)$ is a guess for how much work a reachability traversal of p would expend tracing live objects. The estimator is invoked in Line 1 of the partial garbage collection for CBGC in Figure 4.2. Along with the chooser, the estimator is responsible for opportunism in CBGC, which impacts throughput.

An estimator is based on heuristics. The only invariant it has to satisfy is that for each partition p, $\mathit{dead}(p) + \mathit{live}(p)$ should add up to the total amount of memory that p currently occupies. The estimator problem for CBGC is akin to machine learning problems, in that an estimator uses features (information available at runtime) to guess results that are as accurate as possible. Chapter 3 presents empirical observations of correlations between connectivity and lifetime, which an estimator can exploit by observing connectivity to predict how many objects are dead and life. But CBGC is flexible enough to allow for other estimators that use other heuristics, for example, based on object age.

In practice, an estimator usually does not just guess the total number of bytes in dead and life objects in a partition p. Rather, it estimates a survivor rate $s(p)$ with $0 \leq s \leq 1$. The survivor rate $s(p)$ times the total bytes in objects in the partition yields $\mathit{live}(p)$. Since a partition usually occupies a multiple of the block size, $\mathit{dead}(p)$ is the total number of bytes occupied by blocks of the partition minus $\mathit{live}(p)$. In a copying collector, this is likely to yield an accurate estimate of how much work the collection takes ($\mathit{live}(p)$), and how much memory it reclaims ($\mathit{dead}(p)$). In a non-copying collector, fragmentation may prevent reuse of the freed memory $\mathit{dead}(p)$ for allocation to other partitions than p.

Section 6.1 discusses a number of realistic estimators. Since there are plenty of ways to implement an estimator for CBGC, Section 6.2 also presents an oracular estimator that serves as a point of reference to evaluate less precise estimators against.

6.1 Realistic estimators

A realistic estimator is an estimator that uses only information available at low overhead at runtime. More precisely, a realistic estimator should recoup the cost of obtaining the information it requires by the benefit of improved throughput from opportunistic garbage collection.

Sections 6.1.1 to 6.1.3 describe three realistic estimators that this dissertation experiments with. Section 6.1.4 discusses alternative estimators that one could realistically implement as future work.

6.1.1 Roots estimator

The roots estimator is motivated by the results from Chapter 3 that show that objects reachable from global variables tend to be immortal, while objects reachable only from the stack tend to be shortlived. The roots estimator first scans the roots to find out which partitions contain objects directly pointed to by stack variables and which by global variables. It then propagates this information over the edges of the partition dag to find out which partitions may contain objects that are reachable from stack variables, and which may contain objects that are reachable from global variables. The information is conservative: if a partition p contains objects reachable from stack/global variables, this is noted, but the reverse is not necessarily true. The roots estimator then assumes the survivor rate function

$$s_{roots} = \textbf{if}(\mathit{globalsReach}(p))\ \textbf{then}\ 90\%\ \textbf{else if}(\mathit{stackReach}(p))\ \textbf{then}\ 20\%\ \textbf{else}\ 0\%$$

6.1.2 Decay estimator

The decay estimator assumes that the survivor rate of a partition p is an inverse exponential function of the average age, a, of its objects as shown in Figure 6.1. This expresses the intuition that the longer you wait, the more objects die. The literature refers to the model behind the decay estimator as the radioactive decay model. Stefanović found that none of the well-known analytical models for object lifetime distributions is completely satisfactory [118], but it is still interesting how well a simple model works for CBGC.

For each partition p, the decay estimator maintains the observed decay factor, d, at the previous GC of p, the average age, a, of objects in p, and the total number, n, of objects in p. At each allocation, the decay estimator updates the average age: $a \leftarrow (a + \mathit{timeSinceLastAlloc}) \cdot (n/(n+1))$. After each garbage collection of partition p, the decay estimator updates the decay factor $d \leftarrow -\ln(S)/a$ based on the exact observed survivor rate S in p. Before the first garbage collection, the decay factor d defaults to the same constant for all partitions, e.g. 10^{-8}.

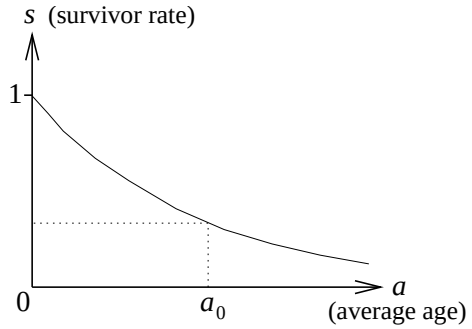


Figure 6.1: Decaying survivor rate $s_{decay}(a) = e^{-da}$.

6.1.3 Combined estimator

The combined estimator is a hybrid estimator combining the roots estimator from Section 6.1.1 and the decay estimator from Section 6.1.2. If a partition has not been collected before, the combined estimator uses the roots estimator, otherwise it uses the decay estimator. This means that it does not have to use an arbitrary default decay factor, since it has an opportunity to learn a better one before using the decay estimator.

6.1.4 Alternatives

The field of estimator design for connectivity-based garbage collection is wide open, and limited only by the creativity of the designers. This section discusses some ideas for future work on CBGC estimator design, but there are plenty of other ways to it.

6.1.4.1 Access density estimator

Access density of a partition is the frequency of accesses to objects in that partition. It can be computed by counting the loads and stores of memory addresses in the partition, and dividing it by the time during which they happened. Access density can be profiled at low overhead with sampling in software [9, 75] or with hardware performance monitors, and can be attributed to partitions by the blocks they occupy.

The access density estimator periodically records the observed access density for each partition. When the access density of a partition suddenly drops, that indicates that the objects in that partition may be dead, because the mutator does not use them anymore. At that point, the access density estimator would estimate a low survivor rate for that partition.

6.1.4.2 Correlation matrix estimator

The correlation matrix estimator would learn, for each pair of partitions, how well-correlated their survivor rates are. It would maintain a correlation matrix with one entry for each pair (p, q) of partitions. This matrix would be initialized based on the

distance of the partitions (p, q) in the partition dag; for example, if (p, q) is an edge, then p and q are likely to have related survivor rates. During garbage collections, the correlation matrix estimator would learn more precise matrix entries based on actual observed survivor rates.

Suppose that the survivor rates of two partitions p and q are highly correlated. When the estimator estimates the survivor rate of p, it guesses that it is similar to the most recent survivor rate of q.

6.1.4.3 Key object estimator

A key object is an object whose death is likely to predict the death of many other objects [67]. One form of key objects is an object that "dominates" others in the object graph. An object o_1 dominates an object o_2 if all paths by which o_2 is reachable go through o_1. The notion of dominance among objects has been shown to be useful for leak detection [97].

An estimator based on key objects could use observations from two earlier garbage collections. The first of these garbage collections would identify a key object based on its observed connectivity in the snapshot of the object graph at that collection. The second of these garbage collections would identify one path of pointers by which the key object is reachable from the roots. The key object predicts the survivor rate of the objects in a partition p. The key object estimator would estimate the survivor rate of p by trying to traverse the pointer chain by which its key object was reachable in the past. If that chain is broken, it would guess that the key object became unreachable, and that p has a low survivor rate.

6.2 Oracle estimator

The most accurate estimator would predict exactly how many objects are dead and live in each partition. In general, a more accurate estimator leads to better CBGC performance. To evaluate the realistic estimators described in Section 6.1, this dissertation also experiments with an estimator that is based on oracular information that would usually not be available to a garbage collector.

The oracle estimator uses the precise deathtimes obtained for a trace using the Merlin algorithm [72], see Section 2.3. When a garbage collection simulator simulates CBGC based on the trace, it uses the Merlin-based death events in the trace to keep a running tally of how many objects are dead and live in each partition. At garbage collection time, the oracle estimator consults that tally to quickly and accurately "estimate" survivor rates.

The oracle estimator assumes information unavailable without a reachability traversal of the whole heap. Thus, unlike the realistic estimators from Section 6.1, it is not useful in practice. However, it allows a limit study of how CBGC behaves with a very accurate estimator. One caveat is that even the oracle estimator may, due to fragmentation, still mis-estimate the number of blocks that a collection of a partition actually reclaims.

Chapter 7

Chooser

The chooser's job is to chose a subset C of the set P of partitions for a partial garbage collection. The chosen set C must be closed under the predecessor relation. The chooser is invoked in Line 2 of the partial garbage collection algorithm for CBGC in Figure 4.2. The estimator finds two numbers $\mathit{dead}(p)$ and $\mathit{live}(p)$ for each partition, and those serve as input to the chooser.

The choice made by the chooser should maximize the expected benefit in reclaimed memory, while minimizing the cost in expended work. This chapter formalizes this problem and presents two solutions: the flow-based chooser, an algorithm that uses network flow to find an optimal solution, and the greedy chooser, a simpler algorithm that may not find an optimal solution.

Section 7.1 states the chooser problem, section 7.2 presents the greedy chooser, and Section 7.3 presents the flow-based chooser. Section 7.4 discusses alternatives to the greedy or flow-based chooser, and even to the problem statement.

7.1 Problem statement

A connectivity-based garbage collector (CBGC) divides the set O of heap objects into a set P of partitions based on a conservative estimate of their connectivity. A partitioning $(\mathit{partition}, P, E_P)$ of the objects consists of a partition map $\mathit{partition} : O \to P$ and a partition dag (P, E_P). The partition map $\mathit{partition}$ associates each object $o \in O$ with its partition $\mathit{partition}(o) \in P$. The edges E_P of the partition dag represent may-point-to relations. In other words, if a pointer may exist between two heap objects, then either the objects must be in the same partition, or there must exist an edge between their partitions in the partition dag. Figure 7.1 gives an example partitioning.

When the CBGC needs to free up some memory, it scavenges a subset $C \subseteq P$ of the partitions. The goal is to choose C such that (i) the objects in C can be collected independently from the rest of the heap, and (ii) the benefit/cost ratio for collecting C is as high as possible.

For the independence property (i), the chooser uses the connectivity information that the partition dag gives it: it picks a set $C \subseteq P$ of partitions that is closed under

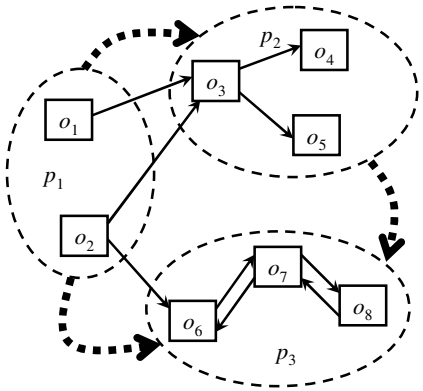


Figure 7.1: Example partitioning. Solid boxes are objects, solid arrows are pointers, dashed ovals are partitions, and dashed arrows are partition edges.

the predecessor relation ($q \in C \wedge (p, q) \in E_P \Rightarrow p \in C$). When an object in C is not reachable from the roots via any objects in C, it is not reachable from the roots via any objects in O, and is therefore garbage; its memory can be reclaimed.

For the benefit/cost ratio property (ii), CBGC starts out by estimating the local cost and benefit of collecting an individual partition. This estimate is represented by a pair of functions $\mathit{dead}, \mathit{live} : P \to \mathbb{N}$ mapping partitions to nonnegative integers. The number of dead objects $\mathit{dead}(p)$ in a partition p is the benefit of collecting it, since their memory can be reclaimed for reuse. The number of live objects $\mathit{live}(p)$ in a partition p is the cost of collecting it, since they need to be traversed before unreachable objects can be reclaimed as garbage. These estimates can come from any of the estimators described in Chapter 6.

Figure 7.2 shows an example of a partition dag with cost/benefit estimates. For instance, partition p_1 may contain objects pointing to objects in p_3, and the estimates say that p_2 contains four dead objects and three live objects $(4 : 3)$.

Equation (7.1) defines the *quality* of a set $C \subseteq P$ of partitions.

$$\mathit{quality}(C) = \frac{\sum_{p \in C} \mathit{dead}(p)}{\sum_{p \in C} \mathit{live}(p)} \tag{7.1}$$

The chooser needs to solve the following problem:

> Given a partition dag (P, E_P) and a pair of functions $\mathit{dead}, \mathit{live} : P \to \mathbb{N}$, find a closed subset $C \subseteq P$ of partitions that maximizes $\mathit{quality}(C)$.

The solution to this problem may not be uniquely defined. The naive algorithm of computing the quality of all closed sets of partitions has complexity $O(2^P)$.

Table 7.1 shows all closed subsets of the partition dag in Figure 7.2 and their quality. The best quality set is $\{p_2, p_5\}$.

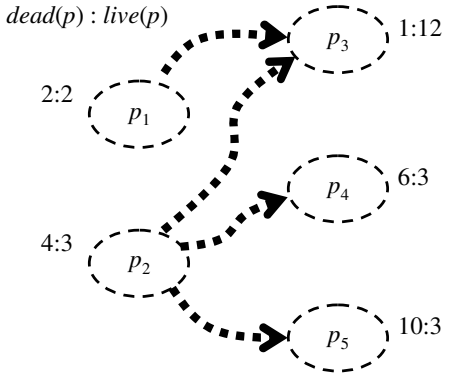


Figure 7.2: Example partition dag annotated with the estimated *dead* and *live* functions.

Table 7.1: Qualities of closed subsets of the partition dag in Figure 7.2.

Closed set $C \subseteq P$	Quality
p_1 p_2 p_3	$7/17 = 0.41$
p_1 p_2 p_3 p_4	$13/20 = 0.65$
p_1 p_2 p_3 p_5	$17/20 = 0.85$
p_1 p_2 p_3 p_4 p_5	$23/23 = 1.00$
(empty set)	$0/0 =$ undefined
p_1	$2/2 = 1.00$
p_2	$4/3 = 1.33$
p_2 p_4	$10/6 = 1.67$
p_2 p_4 p_5	$16/9 = 1.78$
p_2 p_5	$14/6 = 2.33$

7.2 Greedy Chooser

Figure 7.3 shows the greedy chooser algorithm. The goal is to compute the set C of chosen partitions. For a partition $q \in P$, let $A(q) \leftarrow \{p \in P \setminus C \mid p \to^* q\}$ be the set that contains all ancestors of q that have not yet been chosen. Since the ancestor relation is reflexive, q is an ancestor of itself.

Line 1 initializes the set C, and Lines 2 to 3 initialize the unchosen ancestor sets $A(q)$. Since nothing is chosen yet, Line 3 sets $A(q)$ to the set of all ancestors of q, i.e., all partitions p that reach q via a path of edges.

Lines 4 to 16 are the main loop of the greedy chooser algorithm. In each iteration, Line 6 chooses the partition p with the unchosen ancestor set of the highest quality as a candidate for addition. It is added if either a garbage collection of the current choice C would not reclaim enough memory to satisfy the allocation request that triggered GC (Line 8), or the addition would improve the quality of the overall choice (Line 10). If p and its ancestors are chosen, Line 12 updates the chosen set, and

```
 1: C ← ∅
 2: for each partition q ∈ P
 3:     A(q) ← {p ∈ P | p →* q}
 4: done ← false
 5: while not done
 6:     p ← an unchosen partition p′ with the highest quality(A(p′))
 7:     if
 8:             dead(C) not enough for current allocation request
 9:         or
10:             quality(C) < quality(C ∪ A(p))
11:         then
12:             C ← C ∪ A(p)
13:             for each partition q ∈ P
14:                 A(q) ← A(q) \ C
15:         else
16:             done ← true
17: return C
```

Figure 7.3: Greedy chooser algorithm.

Lines 13 and 14 update the sets of unchosen ancestors of all partitions to exclude the newly chosen partitions.

The description of the greedy chooser algorithm in Figure 7.3 omits some details on data structures for implementing it efficiently.

7.2.1 The greedy chooser is not optimal

Figure 7.4 shows an example of a partition dag annotated with the estimated *dead* and *live* functions. For this example, the greedy chooser will not find the optimal solution.

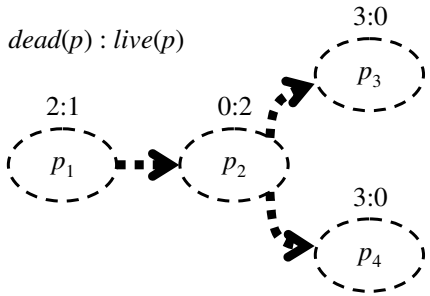


Figure 7.4: Example where the greedy chooser is not optimal.

Table 7.2 shows all closed subsets of the partition dag in Figure 7.4 and their quality. The greedy chooser would start by choosing $C_1 = \{p_1\}$, which has a quality of $2/1 = 2.00$. Then, it would consider adding another partition with its ancestor set.

But all of the three possibilities $C_2 = \{p_1, p_2\}$, $C_3 = \{p_1, p_2, p_3\}$, or $C_4 = \{p_1, p_2, p_4\}$ have worse qualities than what has already been chosen. The only better choice, which the flow-based chooser would find, is $C_{\text{opt}} = \{p_1, p_2, p_3, p_4\}$ with a quality of 2.67, but the greedy chooser does not consider it, since it involves adding more than the ancestor set of a single partition.

Table 7.2: Qualities of closed subsets of the partition dag in Figure 7.4.

Closed set $C \subseteq P$	Quality
p_1 p_2	$2/3 = 0.67$
(empty set)	$0/0 =$ undefined
p_1 p_2 p_3	$5/3 = 1.67$
p_1 p_2 p_4	$5/3 = 1.67$
p_1	$2/1 = 2.00$
p_1 p_2 p_3 p_4	$8/3 = 2.67$

7.3 Flow-Based Chooser

The flow-based chooser finds an optimal solution to the problem stated in 7.1. It does so by reducing it to a max-weight closed set problem, for which the literature has solutions using network flow algorithms. Section 7.3.1 describes the reduction, Section 7.3.2 reviews some basics on network flow, and Section 7.3.3 reviews the solution to the max-weight closed set problem from the literature.

7.3.1 Reduction to max-weight closed set problem

Section 7.3.1.1 proves two lemmas and makes some observations that allow reducing the CBGC partition selection problem to the max-weight closed set problem. Section 7.3.1.2, formulates the algorithm that does that. Section 7.3.1.3 describes how that algorithm uses geometry so it needs to solve only a logarithmic number of instances of the max-weight closed set problem.

7.3.1.1 Preparation

One difficulty of the problem that this section is trying to solve is that the properties of individual partitions do not simply add up ($\frac{d_1}{l_1} + \frac{d_2}{l_2} \neq \frac{d_1+d_2}{l_1+l_2}$). Therefore, the first goal is to reduce the problem to one where they *do* add up. To this end, define a family $w : \mathbb{R} \times P \to \mathbb{R}$ of weight functions on partitions. For each real number $x \in \mathbb{R}$, the weight function $w_x : P \to \mathbb{R}$ is given by $w_x(p) = \mathit{dead}(p) - x \cdot \mathit{live}(p)$. Table 7.3 shows the weights w_x for the example in Figure 7.2 and $x = \frac{21}{11}$.

Lemma 2 gives a feeling for how the weight-functions are related to the problem.

Table 7.3: The weight function $w_{(\frac{21}{11})}$ for the partitions in Figure 7.2.

Partition p	p_1	p_2	p_3	p_4	p_5
$\frac{dead(p)}{live(p)}$	$\frac{2}{2}$	$\frac{4}{3}$	$\frac{1}{12}$	$\frac{6}{3}$	$\frac{10}{3}$
$w_{(\frac{21}{11})}(p)$	$-\frac{20}{11}$	$-\frac{19}{11}$	$-\frac{241}{11}$	$\frac{3}{11}$	$\frac{47}{11}$

Lemma 2 *For all $x \in \mathbb{R}$ and all non-empty closed sets of partitions $C \subseteq P$,*
(i) $0 \leq \sum_{p\in C} w_x(p) \Leftrightarrow x \leq quality(C)$
(ii) $0 \geq \sum_{p\in C} w_x(p) \Leftrightarrow x \geq quality(C)$
(ii) $0 = \sum_{p\in C} w_x(p) \Leftrightarrow x = quality(C)$

Proof. Part (i) can be seen by symbol-pushing:

$$\begin{aligned}
&0 \leq \textstyle\sum_{p\in C} w_x(p) && \Leftrightarrow\\
&0 \leq \Big(\textstyle\sum_{p\in C} dead(p) - x \cdot live(p)\Big) && \Leftrightarrow\\
&0 \leq \Big(\textstyle\sum_{p\in C} dead(p)\Big) - x \cdot \Big(\textstyle\sum_{p\in C} live(p)\Big) && \Leftrightarrow\\
&x \leq \Big(\textstyle\sum_{p\in C} dead(p)\Big) / \Big(\textstyle\sum_{p\in C} live(p)\Big) && \Leftrightarrow\\
&x \leq quality(C)
\end{aligned}$$

This proof used the property that the domain of the functions *live* and *dead* is non-negative. The lemma required that C is non-empty, since otherwise $quality(C) = 0/0$ is undefined. The proof for the other two parts of the lemma is analogous. □

Using Lemma 2, one can show how the solution of maximizing a straight sum correlates with the solution of the problem at hand. Let K be the set of non-empty closed subsets of P.

Lemma 3 *For all $x \in \mathbb{R}$,*
(i) $0 \leq \max_{C\in K}\Big\{\sum_{p\in C} w_x(p)\Big\} \Rightarrow x \leq \max_{C\in K}\{quality(C)\}$
(ii) $0 \geq \max_{C\in K}\Big\{\sum_{p\in C} w_x(p)\Big\} \Rightarrow x \geq \max_{C\in K}\{quality(C)\}$
(iii) $0 = \max_{C\in K}\Big\{\sum_{p\in C} w_x(p)\Big\} \Rightarrow x = \max_{C\in K}\{quality(C)\}$

Proof. For part (i), let C_1 be a witness for the premise, in other words, let $C_1 \in K$ satisfy $0 \leq \sum_{p\in C_1} w_x(p)$. Then Lemma 2 implies $x \leq quality(C_1)$. Since in addition $quality(C_1) \leq \max_{C\in K}\{quality(C)\}$, that shows the conclusion $x \leq \max_{C\in K}\{quality(C)\}$.
A proof by contradiction shows part (ii). Assume that $0 \geq \max_{C\in K}\{\sum_{p\in C} w_x(p)\}$, but that $x < \max_{C\in K}\{quality(C)\}$. Then there exists a closed set of partitions $C_2 \in K$ for which $x < quality(C_2)$. By Lemma 2 that means that $\sum_{p\in C_2} w_x(p) > 0$. But that would mean $\sum_{p\in C_2} w_x(p) > \max_{C\in K}\{\sum_{p\in C} w_x(p)\}$, contradicting the maximality.

Part (iii) follows from the conjunction of parts (i) and (ii). □

Section 7.3.3 will show how to find a solution (along with a witness) to the max-weight closed set problem

$$\max_{C\in K\cup\{\emptyset\}}\Big\{\sum_{p\in C} w_x(p)\Big\}.$$

Here is a sketch how to use that know-how to find a solution (along with a witness) to the main problem

$$\max_{C\in K}\{quality(C)\}.$$

The basic idea is to try different values for x and to use Lemma 3 to search for an x_M that satisfies

$$x_M = \max_{C\in K}\{quality(C)\}.$$

First observe some properties of the search space.

1. For every partition q that has no predecessors ($\neg\exists p\,.\,(p,q) \in E_P$), the singleton set of partitions $\{q\}$ is closed ($\{q\} \in K$). Hence, $x_{\min} = \max\{quality(\{q\}) \mid \neg\exists p\,.\,(p,q) \in E_P\}$ is a lower bound on x_M. For example, in Figure 7.2, $x_{\min} = \max\{quality(\{p\}) \mid p \in \{p_1, p_2\}\} = quality(\{p_2\}) = \frac{4}{3}$.

2. Either $\sum_{p\in P} dead(p) = 0$, or the solution has positive quality, and hence contains at least one partition. One can ignore the case $\sum_{p\in P} dead(p) = 0$, since in that case, it would not make sense to attempt scavenging at all. Let $x_{\max} = \max\{quality(\{p\})\}$ be the best quality of any singleton set of partitions (including singleton sets of partitions with predecessor, which are not closed). Since the quality of a set of partitions is at most as high as the quality of its best member, $x_{\max}$ is an upper bound on x_M. For example, in Figure 7.2, $x_{\max} = \max\{quality(\{p\})\} = quality(\{p_5\}) = \frac{10}{3}$.

3. Let $D = \sum_{p\in P} dead(p)$ and $L = \sum_{p\in P} live(p)$. It is clear that x_M must be of the form d/l where $0 < d \leq D$ and $0 < l \leq L$. Hence, there are $O(DL)$ valid values for x_M. For example, in Figure 7.2, $D = 23$ and $L = 23$, so x_M must be of the form d/l with $0 < d \leq 23$ and $0 < L \leq 23$. Note that in general $D \neq L$.

7.3.1.2 Flow-based chooser algorithm

Figure 7.5 formulates the algorithm. If D is the total number of dead objects, L the total number of live objects, P the number of nodes and E the number of edges in the partition dag, then the total complexity of the algorithm is

$$O\Big(\log(DL)\cdot\Big(\min\{D,L\} + PE\log\Big(\frac{P^2}{E}\Big)\Big)\Big)$$

The algorithm works as follows. Lines 1 and 2 initialize $[low, high]$ to the range in which the solution of the partition selection problem must reside. The do-while-loop

	action	complexity
1	$low \leftarrow x_{\min} = \max\{quality(\{q\}) \mid \neg\exists p\,.\,(p,q) \in E_P\}$;	$P+$
2	$high \leftarrow x_{\max} = \max\{quality(\{p\})\}$;	$P+$
3	**do**{	(
4	choose $x = \frac{d}{l}$ such that $low \le x \le high \wedge 0 < d \le D \wedge 0 < l \le L$ and such that x halves the search space;	$\min\{D, L\}+$
5	find y_x, C_x such that $y_x = \max_{C \in K \cup \{\emptyset\}} \left\{ \sum_{p \in C} w_x(p) \right\} = \sum_{p \in C_x} w_x(p)$;	$PE \log(P^2/E)+$
6	**if**($C_x = \{\} \vee 0 > y_x$){	$\max\{$
7	$high \leftarrow \max\{\frac{d}{l} \mid \frac{d}{l} < x \wedge 0 < d \le D \wedge 0 < l \le L\}$;	$\min\{D, L\}$
8	}**else if**($0 < y_x$){	,
9	$low \leftarrow \min\{\frac{d}{l} \mid \frac{d}{l} > x \wedge 0 < d \le D \wedge 0 < l \le L\}$;	$\min\{D, L\}$
10	}	}
11	}**while**($C_x = \{\} \vee 0 \ne y_x$);	$) \cdot \log(DL)+$
12	**return** C_x;	1

Figure 7.5: Algorithm for flow-based chooser.

in lines 3 to 11 repeatedly solves max-weight closed set problems for values of x in the search space, and uses the solution y_x to either narrow the range $[low, high]$ or to determine that it has found the solution to the CBGC partition selection problem. Line 4 chooses an x that is a possible solution between *low* and *high* such that the search space is halved (see Section 7.3.1.3). Line 5 solves the max-weight closed set problem for the weight function $w = w_x$ (see Section 7.3.3), finding the maximum weight y_x and a witness set of partitions C_x that has the maximum weight. If C_x is empty or $0 > y_x$, then x was too large and line 7 sets *high* to a possible solution value $\frac{d}{l}$ just below x. If $0 < y_x$, then x was too small and line 9 sets *low* to a possible solution value $\frac{d}{l}$ just above x. When the algorithm has found a non-empty set C_x that maximizes $\sum_{p \in C_x} w_x(p)$, then according to Lemmas 2 and 3 the set C_x is also a solution to the CBGC partition selection problem, and the algorithm terminates.

7.3.1.3 Search space halving

The search space for solutions to the CBGC partition selection problem is $\{d/l \mid 0 < d \le D \wedge 0 < l \le L\}$. Figure 7.6 visualizes the search space for $D = L = 23$ as a two-dimensional grid of numbers.

Each number in the search space corresponds to a ray originating in $(0, 0)$. Figure 7.6 shows the rays corresponding to $low = \frac{4}{3}$ and $high = \frac{10}{3}$. The solution x_M must be one of the fractions in the area between the rays.

Without loss of generality assume that $\frac{D}{L} \le low$, in other words, both rays *low* and *high* are above the diagonal of the rectangular solution grid. Pick $d' = \text{lcm}(low.dead, high.dead)$; in Figure 7.6, this yields $d' = \text{lcm}(4, 10) = 20$. The rays *low* and *high* intersect the line $dead = d'$ at $l_l = low.live \cdot d'/low.dead$ and $l_h = high.live \cdot d'/high.dead$.

The goal is to find a point that halves the area of the triangle $((0, 0), (d', l_l), (d', l_h))$. Let A_l be the area of the triangle $((0, 0), (d', l_l), (d', 0))$, which is $\frac{1}{2}(d' \cdot l_l)$, and let A_h be the area of the triangle $((0, 0), (d', l_h), (d', 0))$, which is $\frac{1}{2}(d' \cdot l_h)$. The goal is

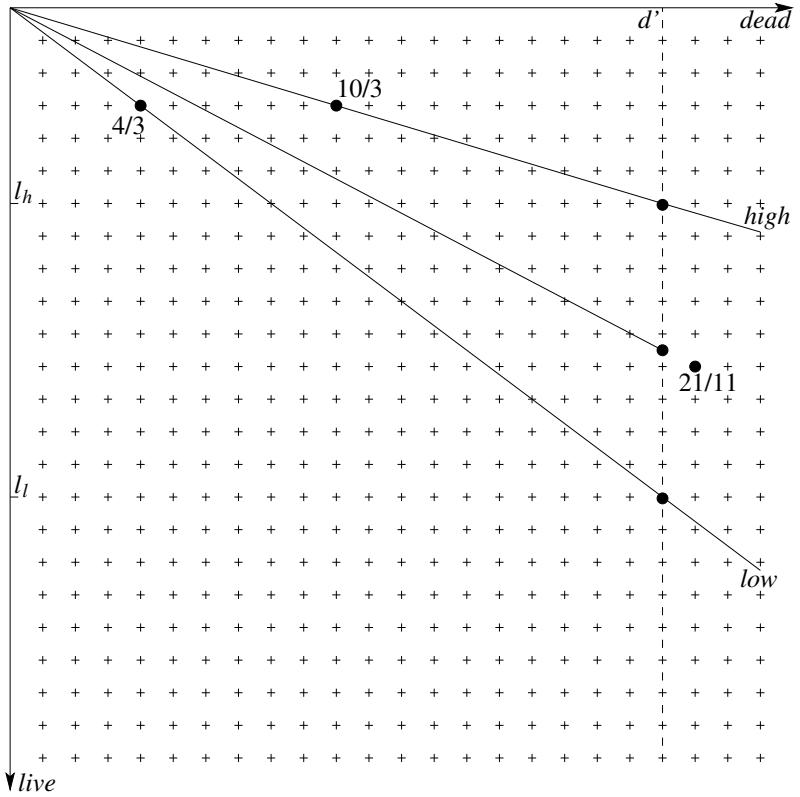


Figure 7.6: Solution space.

to find a point $x' = (d', l_x)$ such that $A_x = \frac{1}{2}(d' \cdot l_x) = \frac{1}{2}(A_l + A_h)$. It is easy to see that $l_x = \frac{1}{2}(l_l + l_h)$.

Line 4 of the algorithm in Figure 7.5 finds an $x' = d'/l_x$ as described above, and then rounds it to the closest $x = d/l$ that corresponds to a point in the grid, i.e. that satisfies $0 < d \le D$ and $0 < l \le L$. For example, for $low = \frac{4}{3}$ and $high = \frac{10}{3}$, the value of x' is $\frac{10.5}{20}$, and the closest legal x is $\frac{11}{21}$.

7.3.2 Flow networks

Before Section 7.3.3 looks at max-weight closed sets, this section reviews some folklore on flow networks, as described e.g. in [42] chapter 27.

A *flow network* consists of a set V of vertices, with two special vertices $s, t \in V$, the source s and the sink t, and a capacity function $c : V \times V \to \mathbb{R}^+$. It can be represented as a graph with edges $E = \{(p, q) \in V \times V \mid c(p, q) > 0\}$.

A *flow* is a function $f : V \times V \to \mathbb{R}$ on pairs of vertices in a flow network that satisfies the three flow properties

- Capacity constraint: for all $p, q \in V$, require $f(p,q) \leq c(p,q)$.
- Skew symmetry: for all $p, q \in V$, require $f(p,q) = -f(q,p)$.
- Flow conservation: for all $p \in V \setminus \{s,t\}$, require $\sum_{q \in V} f(p,q) = 0$.

The value $value(f)$ of a flow is the total flow coming out of the source, in other words $value(f) = \sum_{q \in V} f(s,q)$. A max-flow is a flow of maximum value for its network. A flow defines a residual capacity function $c_f(p,q) = c(p,q) - f(p,q)$ on pairs of vertices.

A cut is a partition (S,T) of the vertices of a flow network into a source side $S \subset V$ with $s \in S$ and a sink side $T = V \setminus S$ with $t \in T$. The capacity $c(S,T)$ of a cut is the total capacity of the edges that cross the partition, in other words, $c(S,T) = \sum_{p \in S, q \in T} c(p,q)$. A min-cut is a cut of minimum capacity for its network.

There is a duality between max-flows and min-cuts.

Theorem 1 (Max-flow min-cut.) $\max_{f \text{ flow}} \{value(f)\} = \min_{(S,T) \text{ cut}} \{c(S,T)\}$

Proof. See [42] page 593. This theorem is also constructive: given a max-flow, one can find a min-cut by choosing all vertices reachable from s via edges with positive residual capacities as S. □

The max-flow problem is to find a max-flow along with a witness. The fairly straight-forward lift-to-front algorithm from [42] page 621 solves it in $O(V^3)$. The Goldberg-Tarjan algorithm is an extension of the lift-to-front algorithm that solves the problem even faster, namely in $O(VE \log(V^2/E))$ [59].

7.3.3 Max-weight closed sets

Max-weight closed set problems and their solution using network flow are described on pages 719-721 of [4]. This section modifies the problem slightly by solving for sets that are closed under the predecessor relation, instead of closed under the successor relation as in [4].

A max-weight closed set problem consists of a partial order $(P, <)$ and a weight function $w : P \to \mathbb{R}$ on elements of P. The goal is to find a closed set $C \subseteq P$ with maximum weight $\sum_{p \in C} w(p)$. Here, a set C is closed if for all $q \in C$, the implication $p < q \Rightarrow p \in C$ holds. Note that this definition differs from [4].

For example, the partial order in Figure 7.2 together with the weight function in Table 7.3 defines a max-weight closed set problem.

The first step is to construct a flow network from the max-weight closed set problem. Partition the elements of P into $P^+ = \{p \in P \mid w(p) \geq 0\}$ and $P^- = \{p \in P \mid w(p) < 0\}$. For the vertices V of the flow problem, choose two additional vertices s, t as source and sink and set $V = P^+ \cup P^- \cup \{s,t\}$. The capacity function $c : V \times V \to \mathbb{R}^+ \cup \infty$ is defined as

- $c(s,p) = w(p)$ for each $p \in P^+$

- $c(p,t) = -w(p)$ for each $p \in P^-$
- $c(q,p) = \infty$ whenever $p < q$ (this differs from [4], since the sets that CBGC is interested in are closed under the predecessor relation)

For example, the weights from Table 7.3 partition $P = \{p_1, p_2, p_3, p_4, p_5\}$ into $P^+ = \{p_4, p_5\}$ and $P^- = \{p_1, p_2, p_3\}$, yielding the flow network in Figure 7.7.

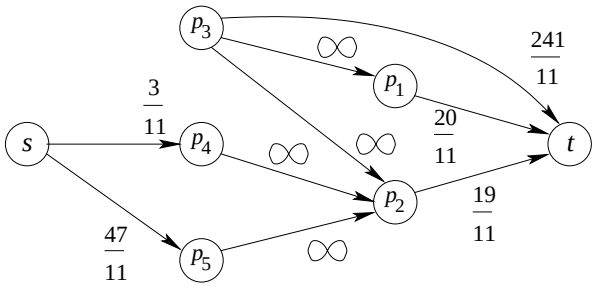


Figure 7.7: Example flow network.

The second step is to find a min-cut (S,T) in the flow network. The set $S \setminus \{s\}$ is a max-weight closed set. A few lemmas show why this is true.

Lemma 4 *A min-cut in a network constructed by the first step above has finite capacity.*

Proof. The cut $(\{s\}, V \setminus \{s\})$ has finite capacity, since there are no vertices $p \in V$ with $c(s,p) = \infty$. The capacity of a min-cut is less than or equal to the capacity of $(\{s\}, V \setminus \{s\})$. Hence, the capacity of a min-cut is finite. □

Lemma 5 *A subset $C \subseteq P$ of the partial order is closed if and only if $\{s\} \cup C$ gives a cut with finite capacity.*

Proof.

$$\begin{aligned} C \text{ is closed} &\Leftrightarrow (q \in C \wedge p < q) \Rightarrow p \in C \\ &\Leftrightarrow q \notin C \vee \neg(p < q) \vee p \in C \\ &\Leftrightarrow (q \in C \wedge p \notin C) \Rightarrow \neg(p < q) \\ &\Leftrightarrow (q \in C \wedge p \notin C) \Rightarrow c(q,p) < \infty \\ &\Leftrightarrow c(\{s\} \cup C, \{t\} \cup (P \setminus C)) < \infty \end{aligned}$$

□

Lemma 6 *If $c(S,T)$ is finite, then*

$$c(S,T) = \sum_{p \in P^+} w(p) - \sum_{p \in (S \setminus \{s\})} w(p).$$

Proof. Being finite, the cut (S,T) can only cross edges involving s or t. More precisely, the capacity is the total capacity of edges going from s to T, which is $\sum_{p\in T} c(s,p)$, plus the total capacity of edges going from S to t, which is $\sum_{p\in S} c(p,t)$.

$$\begin{aligned}
&c(S,T)\\
&= \sum_{p\in T} c(s,p) + \sum_{p\in S} c(p,t)\\
&= \sum_{p\in(P^+\setminus(P^+\cap S))} c(s,p) + \sum_{p\in(P^-\cap S)} c(p,t)\\
&= \sum_{p\in P^+} c(s,p) - \sum_{p\in(P^+\cap S)} c(s,p) + \sum_{p\in(P^-\cap S)} c(p,t)\\
&= \sum_{p\in P^+} w(p) - \sum_{p\in(P^+\cap S)} w(p) + \sum_{p\in(P^-\cap S)} (-w(p))\\
&= \sum_{p\in P^+} w(p) - \sum_{p\in(S\setminus\{s\})} w(p)
\end{aligned}$$

□

With these lemmas, one can show that the reduction from a max-weight closed set problem to a min-cut problem worked.

Theorem 2 *If (S,T) is a min-cut of the flow network, then $S\setminus\{s\}$ is a max-weight closed set of the partial order $(P,<)$.*

Proof. A min-cut minimizes $c(S,T)$, and according to Lemma 6 that is equivalent to minimizing $\sum_{p\in P^+} w(p) - \sum_{p\in(S\setminus\{s\})} w(p)$. Since $\sum_{p\in P^+} w(p)$ is constant, that means that the min-cut maximizes $\sum_{p\in(S\setminus\{s\})} w(p)$. Lemma 5 states that that has indeed maximized over all closed sets. □

When there are multiple min-cuts (S,T), one can choose the one with the largest S using a depth-first search starting at t.

Figure 7.8 shows a max-flow in the flow network from Figure 7.7. A min cut is $S = \{s, p_2, p_4, p_5\}, T = \{p_1, p_3, t\}$, since the residual capacities $c_f(p_2,p_3) = c_f(p_2,t) = 0$. (Note that $c_f(p_3,p_2) \neq 0$, but that is the wrong direction). The max-weight closed set is therefore $\{p_2,p_4,p_5\}$ with a weight of $-\frac{19}{11} + \frac{3}{11} + \frac{47}{11} = \frac{31}{11}$.

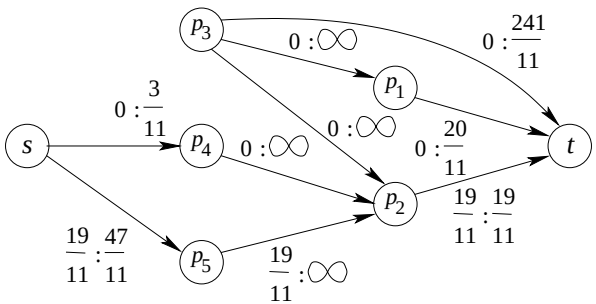


Figure 7.8: Max-flow in network from Figure 7.7.

7.4 Alternatives

This chapter introduced two choosers for CBGC: flow-based and greedy. In theory, the flow-based chooser is optimal, while the greedy chooser can make sub-optimal choices. In practice, the quality of the two choosers is usually almost the same. From a prototype implementation, the flow-based chooser appears to be too slow, whereas the greedy chooser is probably usable. Nevertheless, the flow-based chooser was important for experiments; at the very least, it helped demonstrate that the greedy chooser makes good choices in practice.

One alternative for the chooser is to add more constraints, for example a lower bound on the total size of dead objects that must be reclaimed, or an upper bound on the total size of live objects that should be traversed. In fact, the greedy chooser from Section 7.2 already employs such a lower bound. It has a similar effect for CBGC as Barret and Zorn's dynamic threatening boundary for generational GC [17].

So far, this dissertation has assumed that CBGC runs the estimator, the chooser, and the actual partial GC in that order. It followed a divide-and-conquer strategy by investigating these CBGC components separately. One could also imagine interleaving them, which may enable the estimator and chooser to make more informed decisions and may thus lead to synergy. For example, the estimator may revise its estimates based on exact survivor rates of some partitions in a GC in progress, and the chooser may revise its choice based on the updated estimates.

Another alternative is to change the problem statement so the chooser tries not only to maximize reclaimed memory while minimizing cost, but also tries to improve mutator locality. That would improve overall throughput without necessarily taking less work during garbage collection.

Chapter 8

Partial Garbage Collection

Lines 3 to 16 of Figure 4.2 show the abstract algorithm for partial garbage collection in a CBGC, and Section 4.2.1 describes it in detail. A member of the CBGC family has two choices for making that abstract algorithm concrete: how to implement the tricolor abstraction, and in which order to do the reachability traversal. In addition, each concrete CBGC algorithm has to make a number of design choices for the implementation; Chapter 11 discusses those.

Section 8.1 shows how a CBGC algorithm can make the tricolor abstraction concrete, and Section 8.2 shows concrete traversal orders that a CBGC algorithm can use.

8.1 Tricolor abstraction

The tricolor abstraction[p.183] assigns each object one of three colors white, gray, or black. During a garbage collection, white objects have not been reached yet; gray objects have been reached, but must yet be scanned for pointers to possibly unreached successors; and black objects have been reached along with all their successors.

A concrete connectivity-based garbage collector must somehow keep track of which objects are white, gray, and black. Section 8.1.1 specifies that problem more precisely. Section 8.1.2 is a case study of solving it in a copying CBGC. Section 8.1.3 discusses how CBGC can use other tricolor abstractions.

8.1.1 Problem statement

Figure 4.2 shows the algorithm for partial garbage collection in a CBGC, formulated abstractly in terms of the tricolor abstraction. Making it concrete entails giving concrete implementations of the operations referring to the tricolor abstraction listed in Table 8.1.

Table A.1 shows how a garbage collector can implement the tricolor operations when operating on the full heap. The solutions are mostly transferable to the partial GC situation. The differences are:

Table 8.1: Tricolor operations from Figure 4.2 that a concrete CBGC needs to implement.

Line(s)	Operation
5 and 13	$color(o) \leftarrow$ **gray**
8	$pickGray(p)$
9	$color(o) \leftarrow$ **black**
12	$color(o) =$ **white**
15	reclaim memory
16	$color(o) \leftarrow$ **white**

- Partition map maintenance. All recoloring operations must maintain the partition map. After changing the color of an object o, the map $partition(o)$ must still be defined, and must still yield the same partition as before the recoloring.
- Partition-specific $pickGray(p)$. Whereas in a full GC, $pickGray()$ takes no arguments, in CBGC, it is parameterized by the partition p, and must return a gray object o with $partition(o) = p$.
- Partition-specific memory reclamation. In order to execute Line 15 efficiently, Line 14 must be able to identify all objects of the current partition that are still white, without having to scan white objects in other partitions.
- Partition-specific color reset. In order to execute Line 16 efficiently, Line 14 must be able to identify all objects of the current partition that are black, without having to scan survivor objects in other partitions.

To implement the tricolor abstraction for CBGC, one can start with one of the well-known implementations of the tricolor abstraction for full GC from Table A.1, and then take care of the four differences outlined above.

8.1.2 Copying CBGC

This section is a case study for how to make the tricolor abstraction concrete for CBGC with semispace copying, using a generalization of Cheney scan[p.184]. The reason for picking that algorithm in this case study is that it is simple and efficient, and works naturally on lists of blocks, allowing partitions to be implemented by maintaining lists of blocks. However, CBGC also works with other tricolor abstractions, as discussed below.

Copying CBGC implements the tricolor operations as in Table A.1, with a few changes to take care of the differences in partial GC (Section 8.1.1):

- Partition map maintenance. In copying CBGC, each partition has two semispaces, each of which is a list of blocks. Before copying, objects reside in fromspace, and the partition map maps an object to its partition by mapping its address to a block, and then looking up the partition that that block belongs

to. When the color of an object changes to gray, it is copied into a block in to-space. The partition map is still defined: again, it maps the object address to the new block, and then looks up the partition that that block belongs to.

- Partition-specific *pickGray*(p). To allow picking a gray object of a given partition p, each partition maintains a scan-pointer in its own to-space. Just like in full-heap Cheney scan[p.184], objects from the start of to-space to the scan pointer are black, and objects from the scan pointer to the end of to-space are gray. The only difference is that to-space is not necessarily consecutive, its blocks may be scattered over memory. But that does not prevent picking the next gray object and advancing the scan pointer.

- Partition-specific memory reclamation. In copying Cbgc, each partition has a from-space, which is a list of blocks. To reclaim all white objects of a given partition, the collector releases all blocks of its from-space. The released blocks become available for use by other partitions. For example, they may be used to satisfy future allocation requests, or they may be used to grow the to-space of another partition that the partial GC algorithm has not finished processing yet.

- Partition-specific color reset. Whereas full copying gc resets the color of all objects in to-space to white by flipping the global from-space and to-space, copying Cbgc does the same for one partition by flipping that partition's from-space and to-space. This just means swapping the pointers to the two lists of blocks implementing from-space and to-space.

In copying Cbgc as described above, white objects are non-forwarded objects still in the from-space of their partition, whereas gray and black objects are forwarded and in to-space of their partition. Gray objects are before, black objects after the scan-pointer of their partition. Table 8.2 shows what that means in terms of the tricolor operations.

8.1.3 Alternatives

Using a different tricolor abstraction for Cbgc can be done with two steps: start from the tricolor operations of a full-heap collector, then address the four differences of partial GC in Cbgc listed in Section 8.1.1. Chapter 11 describes how a real Cbgc in Jikes RVM does that for the tricolor abstractions mark-sweep and treadmill, and for immortal objects. This not only demonstrates that Cbgc works with different tricolor abstractions, it also shows that it can use multiple tricolor abstractions in the same heap at the same time. Such hybrids are actually common: all JMTk collectors in Jikes RVM use multiple tricolor abstractions in the same heap [20], and prior work suggests the same thing [26, 98].

Table 8.2: Tricolor operations from Figure 4.2, and how copying Cbgc implements them.

Line(s)	Operation	Copying Cbgc
5 and 13	*color*(o) ← **gray**	copy to to-space of *partition*(o) and forward
8	*pickGray*(p)	at scan pointer in to-space of partition p
9	*color*(o) ← **black**	move scan pointer of *partition*(o) past o
12	*color*(o) = **white**	in from-space of *partition*(o) and not forwarded
15	reclaim memory	release all blocks of the from-space block list of the partition
16	*color*(o) ← **white**	flip meaning of from-space and to-space of *partition*(o)

8.2 Traversal order

Partial GC in a Cbgc performs a reachability traversal, but the algorithm in Figure 4.2 leaves the traversal order for this undefined. It is up to the specific algorithm to decide in which order Line 3 scans the roots, in which order Line 8 returns gray objects, and in which order Line 14 sweeps garbage and survivors.

In the common case, the order of the root scan in Line 3 is determined by what scanning order of globals and variables on thread stacks is the most efficient. All garbage collectors in JMTk in Jikes RVM (see Section 2.1.3) use the same root scan order. Globals are scanned in the order in which their classes declare them and in the order in which Jikes RVM loaded their classes. Threads are scanned from top to bottom, in the order in which Jikes RVM's scheduler stores the thread objects. Wilson, Lam, and Moher [136] discuss the order of root scanning, and warn against storing roots in a hash-table and scanning that in order of hash values.

In the common case, the order of processing gray objects in Line 8 of the algorithm in Figure 4.2 is a by-product of the chosen reachability traversal. For example, copying with Cheney scan leads to a breadth-first order, whereas mark-sweep with a mark stack leads to a depth-first order. A treadmill that keeps gray objects on a doubly-linked list can pick a breadth-first or depth-first order by the decision which end of the queue to retrieve objects from. Hierarchical decomposition is a traversal order that tries to improve collector locality [136]; one can probably generalize it to Cbgc. Alternatively, one could change the traversal order based on dynamic feedback to improve mutator locality [35].

In the common case, the order of sweeping garbage and survivors in Line 14 of the algorithm in Figure 4.2 is also a by-product of the chosen reachability traversal. For example, a semispace copying collector does not need to process white objects one by one in Line 15, but rather discards them en-masse when reclaiming the blocks of the from-space. A mark-sweep collector, on the other hand, will usually sweep

objects in address order.

An idea related to the traversal order is parallelism. CBGC could be extended by parallelizing Line 6 in Figure 4.2: two partitions that are incomparable by the partial topological order of the partition dag can be collected in parallel without any need for synchronization on object granularity. Doing that is future work.

Chapter 9

Design Space Exploration

This chapter explores the performance of garbage collectors in the CBGC family. Chapter 4 introduced the four components of this family: partitioning, estimator, chooser, and partial garbage collection. Chapters 5 to 8 dealt with the components one by one, and showed that there are several ways to instantiate each component. The design space of CBGC is the set of combinations of instantiations of its components.

Broadly speaking, there are three aspects of performance to evaluate in a garbage collector (see Section 1.3): cost in time, or throughput; cost in space, or memory efficiency; and pause times, or responsiveness. This chapter compares a range of CBGC collectors to each other and to the SEMI and APPEL garbage collectors, with respect to these three performance aspect. In this chapter, SEMI denotes a copying garbage collector that does not use any partitioning, but instead performs only full GCs[p.184]. APPEL [7] denotes Appel's flexible-size nursery copying generational collector with two partitions: a nursery partition containing all objects allocated since the last GC, and a mature partition containing the remaining objects[p.190]. Prior work has found APPEL to be one of the best performing generational collectors [21].

For CBGC, this chapter examines several different partitionings, estimators, and choosers. It abbreviates the configurations with CBGC*pec*, where

- $p \in \{\mathrm{H}, \mathrm{T}, \mathrm{A}\}$ is the Harris, Type-dynamic, or Allocsite-dynamic partitioning (Chapter 5);
- $e \in \{\mathrm{D}, \mathrm{R}, \mathrm{C}, \mathrm{O}\}$ is the Decay, Roots, Combined, or Oracle estimator (Chapter 6); and
- $c \in \{\mathrm{G}, \mathrm{F}\}$ is the Greedy or Flow-based chooser (Chapter 7).

This chapter considers only one instantiation of the partial GC component of CBGC, namely copying CBGC as described in Section 8.1.2. The two non-CBGC collectors in this chapter are also copying collectors, thus comparing with a copying CBGC makes he most sense.

Figure 9.1 shows the relative strength of the various configurations. Two configurations are connected by an edge if the upper configuration is theoretically "better"

than the lower one. For example, CbgcAOF is theoretically better than CbgcAOG, since CbgcAOF uses the flow-based chooser, which is optimal given a particular partitioning and estimator, whereas CbgcAOG uses the greedy chooser, which is only approximate.

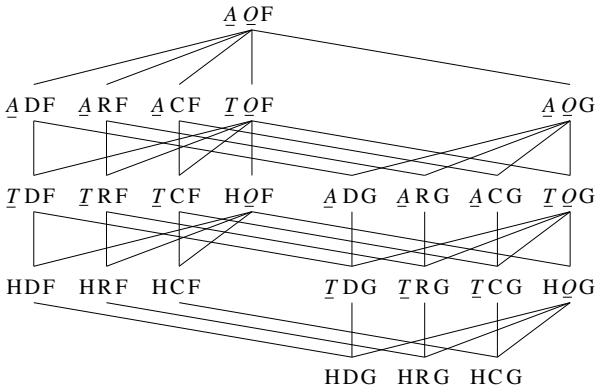


Figure 9.1: Cbgc Configurations. Higher positions represent a "stronger" or more optimal configuration. The configurations with an '$\underline{A}$', '$\underline{T}$', or '$\underline{O}$' use information from a benchmark run and are therefore not realistic.

Components in Figure 9.1 that are not realistic (e.g., they require information from a program run) are underlined. For example, $\underline{T}$DF uses an unrealistic type-dynamic partitioning, and so the 'T' is underlined. Including the unrealistic configurations allows an exploration of the limits of how well Cbgc could perform if one provided it with better instantiations of the components than those that are known today.

Section 9.1 discusses the experimental methodology for the design space exploration.

Sections 9.2, 9.3, and 9.4 evaluate the cost in time, cost in space, and pause times for just two points of the design space, namely CbgcAOG and CbgcHCG. CbgcHCG is a fully realistic configuration, giving a lower bound for what can be achieved in practice, whereas CbgcAOG is one of the strongest (albeit unrealistic) configurations, presenting the full potential of Cbgc.

Section 9.5 explores many of the other points in the design space in order to better understand the tradeoffs in Cbgc algorithms.

In the experiments, unless otherwise indicated, the collectors use a heap size of three times the high watermark of the benchmark, and a block size of 1KB. Section 9.6 shows that the results generalize to different heap sizes and block sizes.

9.1 Methodology

The methodology of the garbage collector performance experiments in this chapter is based on a trace-driven simulator. Section 9.1.1 describes the traces that drive the simulator. Section 9.1.2 outlines the design of the simulator itself. Section 9.1.3 discusses strengths and weaknesses of the methodology.

9.1.1 Garbage collection traces

This chapter uses traces from 13 Java benchmarks to drive garbage collection simulations. Section 2.2 described the benchmarks, and Section 2.3 described the traces. The traces are chronological recordings of every object allocation, pointer update, and object death over the execution of a program. Each of these events includes the information required to simulate it.

Table 9.1 lists the benchmarks. The table shows the total allocation and high watermark (maximum number of simultaneously reachable bytes) to give a feeling for the size of the runs. Benchmark null is an empty Java program, and thus it gives an indication of how much memory the virtual machine uses just for starting up. It is included to put the data in Table 9.1 into perspective, but of course subsequent sections do not present numbers for null. Jikes RVM allocates its own objects on the same heap as application objects, and the simulations in this chapter treat objects from all owners uniformly.

Table 9.1: Traces used in this evaluation; for a description of what the benchmarks do, see Tables 2.2 and 2.3.

Program	Total allocation		High water-mark bytes
	objects	bytes	
null	373,315	48,791,248	48,695,104
power	1,230,662	73,228,028	49,700,876
deltablue	1,303,984	77,502,904	49,305,572
bh	1,453,904	83,503,920	49,342,316
health	2,102,507	86,718,316	51,499,180
db	3,807,582	133,782,036	58,714,536
compress	388,832	159,951,240	56,684,200
mtrt	7,973,471	237,305,784	59,777,468
ipsixql	10,089,370	351,117,828	53,827,992
jess	12,345,040	437,641,308	54,330,928
jack	14,274,816	473,120,964	55,925,844
xalan	7,388,779	488,960,484	85,682,372
pseudojbb	18,063,813	566,361,852	78,049,072
javac	17,943,604	579,746,244	61,123,296

Table 9.1 shows the benchmarks ordered by increasing numbers of bytes allocated. Subsequent tables and figures use the same ordering, so that the bottom-most (if arranged vertically) or right-most (if arranged horizontally) benchmarks are the largest.

9.1.2 Garbage collection simulator

This chapter is based on a simulator, gcSim; gcSim is open-source and available at `http://www.cs.colorado.edu/~hirzel/gcSim`. It consists of implementations of the collectors described in this chapter, supported by a number of abstractions. These abstractions include models of the root set, the heap, and individual objects and a block manager (the heap is organized as a number of fixed-sized blocks[p.177]).

9.1.3 Strengths and weaknesses of the methodology

Using a simulator to evaluate CBGC had some advantages over a full implementation: it allowed abstracting from implementation details, it allowed comparing CBGC to other collectors in a controlled environment, and it allowed experimenting with various CBGC algorithms, some of which are not possible to implement in practice, but are interesting for limit evaluation.

There are also drawbacks to not using a full implementation in a Java virtual machine. The most important is that simulation can not give concrete timing numbers. Another drawback is the lack of cache-level locality numbers. However, previous work has shown that simulators can be useful for GC research: the older-first collector was first evaluated using a simulator similar to ours [117], and the results carried over to the real implementation [116].

Chapter 11 describes the design and implementation of a CBGC algorithm in Jikes RVM, demonstrating that CBGC works correctly in the presence of all real-world challenges of a full system.

9.2 Cost in time

This section compares SEMI, APPEL, CBGCHCG, and CBGCAOG with respect to the time cost of garbage collection. Section 9.2.1 compares the amount of work each collector does during garbage collection, while Section 9.2.2 considers the other time costs of garbage collection.

9.2.1 GC work per time

This chapter uses the gcWorkPerTime metric to compare the work that different garbage collection algorithms do during GC. The metric gcWorkPerTime is computed as the total number of bytes copied in all garbage collections (work), divided by the total allocation, in bytes, of the program (time). Measuring time in bytes allocated is common in memory management research[p.180]; Table 9.1 shows the total allocation of the benchmark programs.

As an example, if the gcWorkPerTime is 0.5, then for every 2 bytes that the application allocates, GC must perform 1 byte of copying work. The gcWorkPerTime metric is often called mark/cons ratio[p.185] in the literature.

Figure 9.2 shows gcWorkPerTime for SEMI, APPEL, the simplistic CBGCHCG, and the oracle-based CBGCAOG. In this and subsequent figures taller bars indicate worse performance.

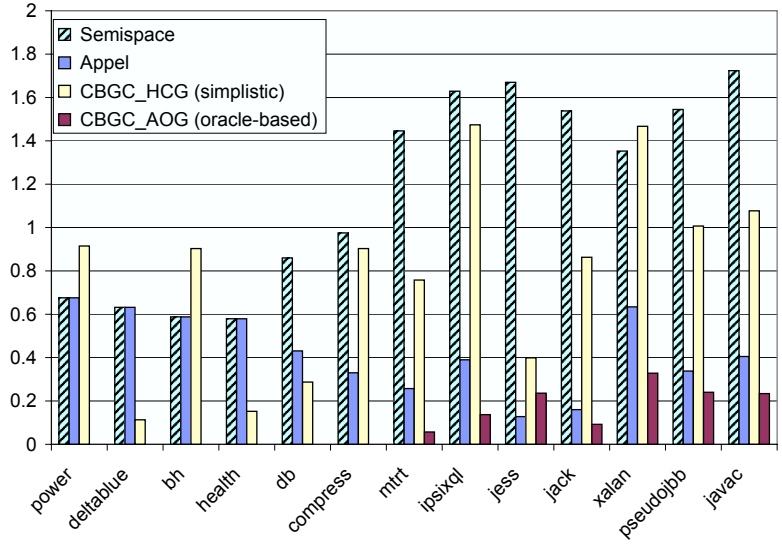


Figure 9.2: gcWorkPerTime (bytes copied / bytes allocated).

Comparing CBGCAOG to APPEL shows that except for one benchmark (jess), CBGCAOG outperforms APPEL. For the benchmarks that do relatively little allocation, the bar for CBGCAOG is not even visible. In other words, at each garbage collection CBGCAOG is able to choose only partitions where almost every object is a dead object. For the benchmarks that allocate more, CBGCAOG copies more, but typically still copies far fewer bytes than APPEL does.

On the other hand, the realistic CBGCHCG collector is usually worse than APPEL, but usually outperforms SEMI. CBGCHCG performs worse than SEMI for xalan because of weakness in the estimator (Figure 9.8); the combined estimator tells CBGCHCG that certain partitions have many dead objects when they do not. CBGCHCG performs worse than SEMI for bh and power because these benchmarks perform only one collection each with SEMI (Table 9.2) and thus the exact timing of the one collection ends up being significant.

9.2.2 Other time cost factors

Besides the time spent in garbage collection, there are many other costs of memory management [123]. Other time costs of CBGC include time to perform the partitioning analysis upon class loading, and running the estimator and chooser before GC. Time costs present in APPEL but not in CBGC include the time to compile and execute write barriers. In addition, the two collectors may have different memory system costs (Section 9.3.1 presents a preliminary exploration of these last costs).

9.2.3 Cost in time conclusions

- The oracle-based CBGCAOG usually performs much less GC work per time than APPEL. A good CBGC algorithm can potentially have lower cost in time than state-of-the-art collectors.
- The simplistic CBGCHCG usually performs more GC work per time than APPEL, but less than SEMI. Reducing CBGC's cost in time requires better realistic CBGC components.
- There are other time cost factors besides GC work per time. The design and implementation of a real-world CBGC in Chapter 11 is a step towards allowing more definitive comparisons.

9.3 Cost in space

There is an obvious space-time tradeoff with garbage collectors. Increasing the memory available to run an application reduces the time spent in garbage collection. At one extreme, if an application has an infinite amount of memory, it will never need to garbage collect. Conversely, if an application has little memory, it needs to perform more collections which causes an increase in the time costs of GC.

To allow controlled and fair comparison across garbage collectors, the experiments in this chapter use a fixed heap size[p.187] equal to three times the high watermark of the benchmark program. The high watermark is the maximum number of bytes that are reachable at the same time. Table 9.1 shows the high watermark for each benchmark, computed using the Merlin perfect death time traces. Most of this chapter uses a fixed heap size of three times this high watermark, because previous research shows it to be a heap size at which many algorithms perform well [7, 21]. Section 9.6 shows that the results hold across a range of heap sizes.

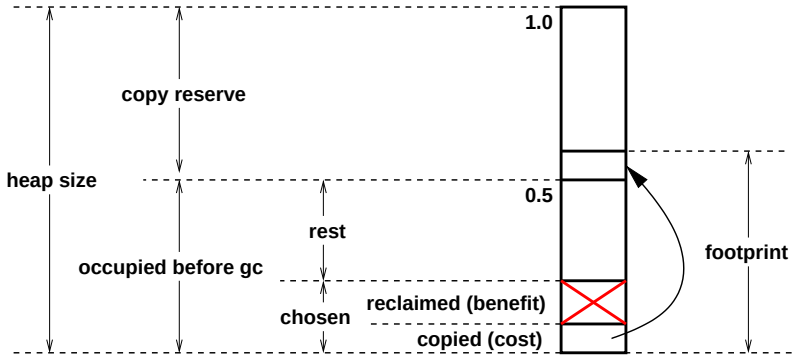


Figure 9.3: Heap Anatomy.

Even though the *heap size* is fixed, one can still evaluate the cost in space by measuring the *footprint* of memory in use. Figure 9.3 shows the space usage of garbage

collectors in the simulator (for simplicity, Figure 9.3 shows the memory regions as contiguous; in reality, they are represented as sets of fixed-sized blocks.) During program execution, the collectors maintain half of the heap as "copy reserve"[p.187] since in the worst case, all objects may survive GC. While SEMI collects the entire heap at each GC, CBGC and APPEL may perform a partial GC and examine only a few partitions ("chosen"). A partial GC will use only a subset of the copy reserve. The maximum footprint of a collector is the maximum fraction of the heap that is simultaneously in use[p.187]. A smaller footprint generally translates into better memory system performance (e.g., less paging).

9.3.1 Maximum footprint

Figure 9.4 shows the maximum footprint for SEMI, APPEL, the realistic, but simple CBGCHCG, and the oracle-based CBGCAOG.

Figure 9.4 shows that the realistic CBGCHCG consistently has a smaller footprint than APPEL. For some benchmarks (e.g., jess) the footprint of CBGCHCG is much smaller than that of APPEL.

The oracle-based CBGCAOG consistently has a much smaller footprint than APPEL. Oftentimes, the footprint of CBGCAOG is close to 0.5, indicating that CBGCAOG hardly uses the copy reserve at all.

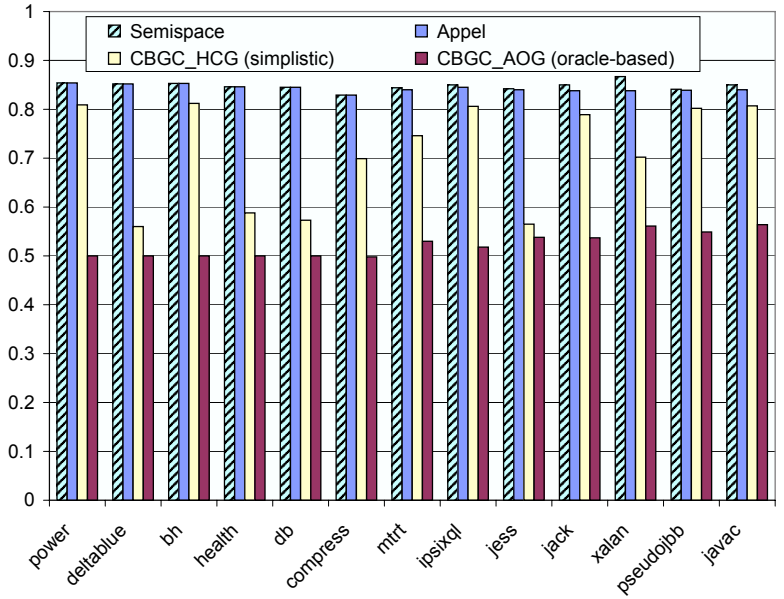


Figure 9.4: maxFootprint (maximum footprint / heap size in bytes).

There are two reasons why CBGC has such a good maximum footprint. First, in the experiments for this chapter, the pathological case where it would need to do the equivalent of a full GC in APPEL never occurred. Second, CBGC allows

early reclamation[p.186], reusing memory while a GC is still in progress as discussed in Section 4.5. Appel, on the other hand, usually has a maximum footprint close to 85% of the heap size. This is because most of the immortal data of a benchmark is allocated up front and survives the first garbage collection, where Appel's flexible-sized nursery occupies 50% of the heap. The immortal data includes the runtime system of the virtual machine, the application stacks, and compiled methods.

Both Cbgc and Appel may suffer from some amount of internal fragmentation. One might have expected this to be worse for Cbgc than for Appel, because Cbgc has more partitions. The simplistic CbgcHCG usually uses around 85 partitions, and on average 3.4 partitions contain 95% of the heap objects. The oracle-based CbgcAOG usually uses around 900 partitions, and on average 13.7 partitions contain 95% of the heap objects. But as Figure 9.4 shows, the differences in fragmentation due to many partitions has little impact on the footprint.

9.3.2 Other space cost factors

Besides the space occupied by objects, all garbage collectors maintain a number of data structures that also contribute to the space cost. For example, Cbgc needs space for the partition graph and Appel needs space for the remembered sets and the write barrier instructions (in the code). While this chapter presents no detailed experimental results for these costs, the above mentioned space costs for Cbgc appear to be insignificant.

9.3.3 Cost in space conclusions

- Even the simplistic CbgcHCG has a lower cost in space than Appel. Therefore, the paging and TLB activity of Cbgc are likely to be lower than that of Appel. That can lead to better memory system performance.
- Since Cbgc has a lower cost in space than Appel, it may enable programs to run in less memory.

9.4 Pause times

For stop-the-world collectors[p.191], the amount of work that a garbage collector performs during a collection determines the amount of time for which the application is paused. This chapter considers two measures for this: the amount of work the garbage collector performs on average during a collection (Section 9.4.1) and the maximum amount of work the garbage collector performs on any collection (Section 9.4.2).

9.4.1 Average work per GC

Figure 9.5 gives the average amount of copying performed by a collector as a fraction of heap size in bytes (avgWorkPerGc). For example, if the avgWorkPerGc is 0.01, then the average collection copies 1% of the heap size (see Figure 9.3).

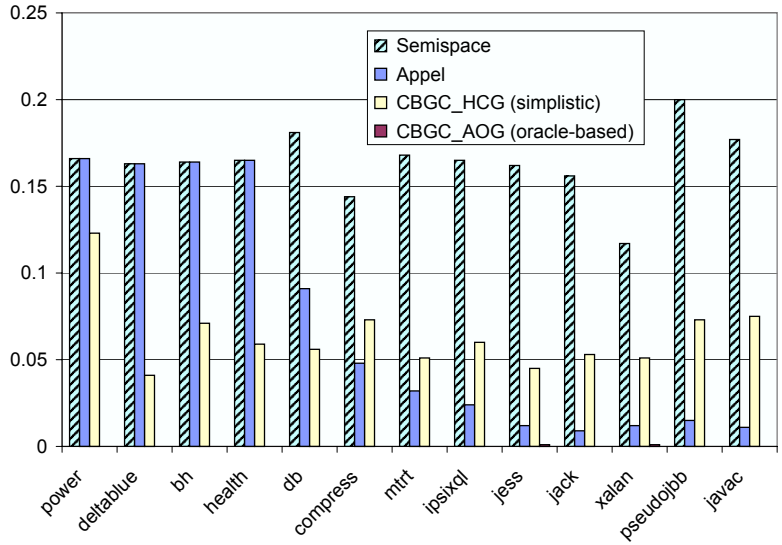


Figure 9.5: avgWorkPerGc (average copied / heap size in bytes).

Figure 9.5 shows that while Appel usually performs well (especially for the larger benchmarks), CbgcAOG performs much better. As a matter of fact, CbgcAOG performs so well that its bars are not visible for most of the benchmarks.

The realistic CbgcHCG performs well, even outperforming Appel for some benchmarks. As expected, Semi performs poorly, since at each collection it needs to copy all the reachable objects.

Table 9.2: Number of Garbage Collections.

Program	Semi	Appel		CbgcHCG	CbgcAOG
	Total	Total	Major	Total	Total
power	1	1	0	2	1
deltablue	1	1	0	1	55
bh	1	1	0	4	11
health	1	1	0	1	10
db	2	2	0	3	9
compress	3	3	0	5	5
mtrt	6	6	0	13	1,925
ipsixql	11	22	1	34	1,097
jess	14	15	0	17	692
jack	14	34	0	30	2,155
xalan	9	33	2	31	703
pseudojbb	12	39	1	27	2,619
javac	17	79	2	35	3,349

CbgcAOG has such a low avgWorkPerGc because it usually collects only partitions that contain mostly garbage *and* it performs many garbage collections (Table 9.2). Since there is an overhead to triggering a garbage collection (e.g., root scanning), it may be worthwhile to consider other choosers that pick more partitions to collect at each collection (Section 7.4).

9.4.2 Maximum work per GC

Even if a garbage collector has a low average pause time, it may still be disruptive if some pauses are much longer. Therefore, this section considers the maximum pause time.

Figure 9.6 shows maxWorkPerGc for Semi, Appel, the realistic CbgcHCG, and the oracle-based CbgcAOG. The maxWorkPerGc is the maximum number of bytes copied at any garbage collection divided by the heap size (see Section 9.3). For example, if maxWorkPerGc is 0.18, then the largest collection copies 18% of the heap size (see Figure 9.3).

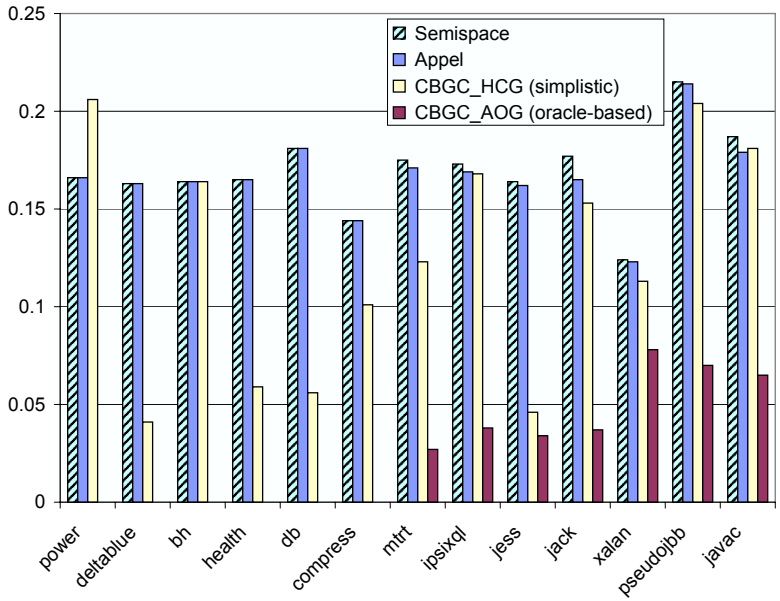


Figure 9.6: maxWorkPerGc (maximum copied / heap size in bytes).

Figure 9.6 shows that even the realistic CbgcHCG has a better maxWorkPerGc than Appel for all benchmarks except power and javac, where it is slightly worse. CbgcAOG is consistently the best configuration.

With respect to this metric, Appel does not perform any better than Semi. The reason for this is that Appel occasionally performs major collections that collect the entire heap. Also, the first collection with Appel is effectively a full heap collection.

9.4.3 Pause times conclusions

- Even the simplistic CbgcHCG tends to incur less work per GC than Appel.
- Cbgc never required full-heap collections in the experiments for this chapter. It always chose few enough and small enough partitions to collect. While one can construct a pathological situation that forces Cbgc to do a full-heap GC, it is unlikely to ever happen in practice. Appel, on the other hand, usually does full-heap GCs in long runs.

9.5 Other points in the Cbgc design space

The previous sections looked at whether or not Cbgc *can* outperform other garbage collectors. This section explores the Cbgc design space and tries to identify weaknesses in the realistic Cbgc implementations. Since cost in time appears to be the biggest challenge for the simplistic CbgcHCG (compared to Appel, it already has low cost in space and low work per GC), this section uses cost in time to compare Cbgc configurations.

This section uses the methodology of varying one component (e.g., partitioning) while fixing the other components at their strongest level (which may be unrealistic). This methodology allows evaluation of how different implementations of a component perform without worrying about interactions with poor implementations of the other components. For example, a poor implementation of a partitioner may obfuscate the differences between estimators.

9.5.1 Partitionings

This section explores the partitionings described in Chapter 5. The Harris partitioning is realistic, but simple. The allocsite-dynamic partitioning is a limit study for the finest-grained partitioning one can get if all objects allocated at the same allocation site must reside in the same partition. The type-dynamic partitioning falls in between the Harris partitioning and the allocsite-dynamic partitioning.

Figure 9.7 shows the gcWorkPerTime metric from Section 9.2 for CbgcHOG, CbgcTOG, and CbgcAOG. In other words, it keeps the estimator (oracle) and chooser (greedy) constant and varies the partitioner. Figure 9.1 shows the theoretical ordering between these alternatives.

Figure 9.7 shows that as the partitioning improves, so does the gcWorkPerTime metric. Using the type-dynamic partitioning is usually not much better than using the Harris partitioning. Using the allocsite-dynamic partitioning often reduces the amount of GC work by a factor of 2 or more over the other partitioners.

Compared to Figure 9.2, Figure 9.7 shows that the differences due to different partitionings are less than the total difference between the realistic CbgcHCG and the unrealistic CbgcAOG. Thus, the estimator also contributes to the difference.

To summarize, the quality of the partitioning makes a big difference in the performance of Cbgc. The results suggest that Cbgc should avoid type-based partitionings, and use a realistic partitioning based on allocation sites instead. That

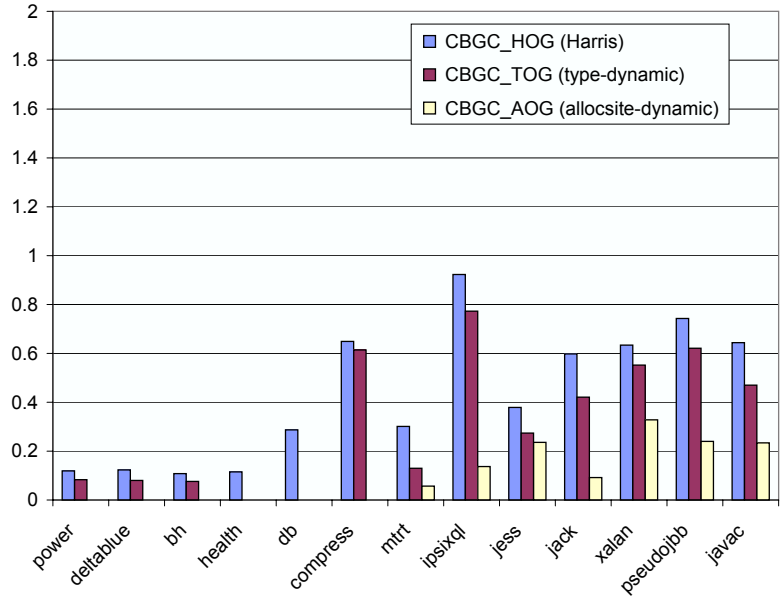


Figure 9.7: gcWorkPerTime (bytes copied / bytes allocated) for different partitionings. The CBGC_AOG bars are the same as in Figure 9.2, the other bars are new.

narrows down the choice to the bottom four partitionings in Table 5.1. Chapter 10, which is based on the paper [79], describes how to perform Andersen's analysis in a Java virtual machine, and Chapter 11 uses it for a realistic partitioning.

9.5.2 Estimators

This section explores the estimators described in Chapter 6. The roots and decay estimators are simple and realistic. The combined estimator is a hybrid of these two, so it is still realistic and a little more sophisticated. The oracle estimator is a limit study that always estimates correctly. A realistic estimator can perform at most as well as the oracle.

Figure 9.8 shows the gcWorkPerTime metric from Section 9.2 for CBGCARG, CBGCADG, CBGCACG, and CBGCAOG. Since the decay estimator needs some experience to learn the survival rates, it performs poorly for the smaller benchmarks that cause few collections (for those benchmarks, the decay estimator also has high cost in space and high pause times). The roots estimator performs poorly for larger benchmarks. Fortunately, the combined estimator combines the strengths of the roots and decay estimator to yield a much better estimator. The oracle estimator performs much better than the other estimators especially for the larger benchmarks.

To summarize, the combined estimator is the best of the realistic estimators implemented so far. However, comparison with the oracle estimator shows that there is still much room for improvement. Hence, it may be worthwhile to explore

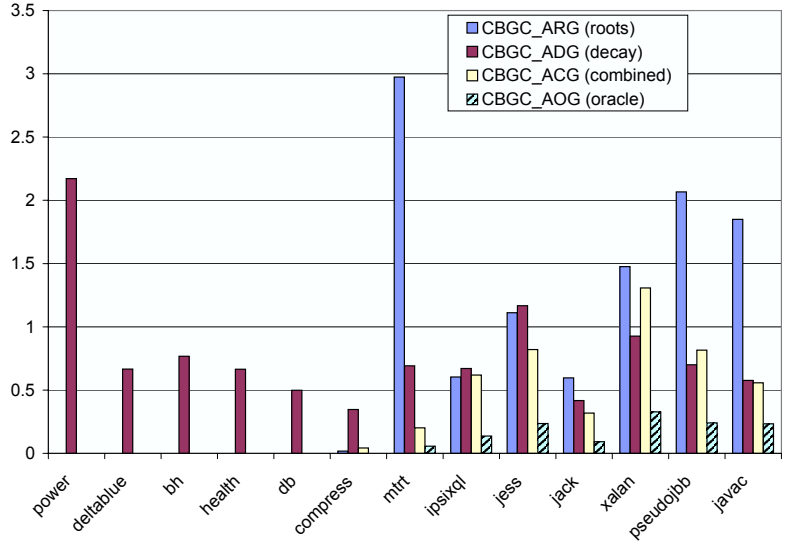


Figure 9.8: gcWorkPerTime (bytes copied / bytes allocated) for different estimators. The CBGC_AOG bars are the same as in Figure 9.2, the other bars are new.

some of the alternative estimators that Section 6.1.4 suggests.

9.5.3 Choosers

So far, this chapter reported results for the greedy chooser instead of the optimal flow-based chooser. This section compares the gcWorkPerTime, maxFootprint, avgWorkPerGc, and maxWorkPerGc metrics from Sections 9.2 to 9.4 for CBGCAOG and CBGCAOF. Chapter 7 describes how the two choosers work.

Table 9.3 shows the results. Column "Average time choose" is the average wall clock time for running the Java implementations of the choosers in gcSim, on a 1.4GHz Pentium 4 with 1GB of RAM. The unit is seconds.

The metrics "GC work per time", "maximum footprint", and "average and maximum work per GC" evaluate how the quality of the chooser affects overall GC performance. If the chooser makes good choices, the numbers are lower; if it makes bad choices, they are higher. In almost all cases, the flow-based and the greedy chooser make choices of identical quality. The only exceptions occur for the programs ipsixql, xalan, and pseudojbb. For ipsixql, the greedy chooser turns out to improve overall GC performance. At first glance, this seems to contradict the optimality of the flow-based chooser; but it can happen if the choosers make different choices of similar quality that lead to different fragmentation effects down the line.

The wall-clock times have to be taken with a grain of salt, since the choosers are not optimized for running quickly. The flow-based chooser implemented in gcSim uses the cubic lift-to-front flow algorithm described in [42], but in the literature, there are algorithms with better complexity (e.g. [59]). The greedy chooser is not

Table 9.3: Performance of different choosers.

Program	GC Work per Time		Maximum Footprint		Work per GC Average		Work per GC Maximum		Average Time Choose	
	AOF	AOG	AOF	AOG	AOF	AOG	AOF	AOG	AOF	AOG
power	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.11s	0.13s
deltablue	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	1.33s	0.10s
bh	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.19s	0.11s
health	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.62s	0.11s
db	0.000	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.34s	0.10s
compress	0.000	0.000	0.498	0.498	0.000	0.000	0.000	0.000	3.97s	0.04s
mtrt	0.057	0.057	0.530	0.530	0.000	0.000	0.037	0.037	2.14s	0.12s
ipsixql	0.151	0.137	0.532	0.518	0.000	0.000	0.054	0.049	4.81s	0.13s
jess	0.236	0.236	0.538	0.538	0.001	0.001	0.046	0.046	6.69s	0.18s
jack	0.092	0.092	0.537	0.537	0.000	0.000	0.047	0.047	3.03s	0.15s
xalan	0.333	0.328	0.536	0.561	0.001	0.001	0.045	0.130	12.62s	0.32s
pseudojbb	0.239	0.240	0.550	0.549	0.000	0.000	0.081	0.081	4.25s	0.18s
javac	0.234	0.234	0.564	0.564	0.000	0.000	0.076	0.076	5.02s	0.24s

very efficient either, it builds up auxiliary data structure for every choice that could be cached in a better implementation.

Table 9.3 shows that the wall-clock time of the flow-based chooser is quite high. In fact, given that a GC should take less than around 0.3 seconds to be imperceptible by the user, it can probably not afford the flow-based chooser, even if it is optimized by an order of magnitude. The greedy chooser performs much better, and with a little more optimization, its choice times will be acceptable.

9.6 Sensitivity to heap size and block size

So far this chapter reported results for a heap size of 3.0 times the high watermark and a block size of 1KB. This section considers how the results change when using different heap and block sizes.

9.6.1 Heap sizes

Assuming zero fragmentation, a copying collector needs at least a heap size of 2.0 times the high watermark to work, since it keeps a copy reserve. Previous research indicates that a heap size of 2.5 times the high watermark is tight and a heap size of 4.0 times the high watermark is loose [21].

Table 9.4 shows the gcWorkPerTime metric from Section 9.2 using three different heap sizes. For all heap sizes and all benchmarks except for jess the relative performance of CbgcAOG and Appel is the same. For benchmark jess, CbgcAOG performs worse than Appel at heap sizes 2.5 and 3.0.

While not all experiments in this chapter were repeated with a range of heap sizes, Table 9.4 gives some confidence that the results hold for other heap sizes as well.

Table 9.4: gcWorkPerTime (bytes copied / bytes allocated) for different heap sizes.

Program	HeapSizeFrac 2.5		HeapSizeFrac 3.0		HeapSizeFrac 4.0	
	Appel	CbgcAOG	Appel	CbgcAOG	Appel	CbgcAOG
power	0.679	0.000	0.676	0.000	0.000	0.000
deltablue	0.634	0.160	0.632	0.000	0.000	0.000
bh	0.593	0.075	0.588	0.000	0.000	0.000
health	0.596	0.071	0.579	0.000	0.000	0.000
db	0.433	0.000	0.431	0.000	0.430	0.000
compress	0.384	0.020	0.330	0.000	0.312	0.000
mtrt	0.277	0.150	0.257	0.057	0.248	0.029
ipsixql	1.078	0.342	0.390	0.137	0.173	0.049
jess	0.132	0.541	0.128	0.236	0.121	0.101
jack	0.397	0.202	0.160	0.092	0.134	0.050
xalan	0.570	0.373	0.634	0.328	0.467	0.126
pseudojbb	0.641	0.539	0.338	0.240	0.187	0.084
javac	0.734	0.565	0.405	0.234	0.183	0.113

9.6.2 Block sizes

All data presented in this chapter uses a block size of 1KB. The motivation for this small block size was to reduce potential internal fragmentation, especially when Cbgc uses a large number of partitions. Since Appel uses only two partitions, it is unlikely to be affected by a different block size.

Experiments with block size 4KB revealed that the numbers for each individual collector are virtually the same with block size 1KB as they are with block size 4KB. The small variations appear unrelated to the collector.

9.7 Conclusions

In the simulator, even a simplistic member of the Cbgc family outperforms Appel with respect to pause times and memory footprint. The experiments with oracle-based Cbgc reveal that Cbgc has potential to dramatically improve upon all performance aspects of existing garbage collectors.

A partitioning needs to be more fine-grained than types for Cbgc to yield good cost in time. The coarsest partitioning (putting all objects in one partition) would correspond to Semi.

The realistic combined estimator is a good start at obtaining low cost in time, but there is still much room for improvement with better estimators. A bad estimator can lead Cbgc totally astray.

The greedy chooser is simple and close to optimal.

Chapter 10

Pointer Analysis for Java

Chapter 9 indicates that connectivity-based garbage collection needs a partitioning based on allocation sites to perform well. One such analysis is Andersen's pointer analysis, and Section 5.3.5 describes how to use it for partitioning. This chapter describes how to make Andersen's analysis work for all of Java; it is based on the ECOOP 2004 paper "Pointer analysis in the presence of dynamic class loading".

Many optimizations need precise pointer analyses to be effective. Unfortunately, some Java features, such as dynamic class loading, reflection, and native methods, make pointer analyses difficult to develop. Hence, prior pointer analyses for Java either ignore these features or are overly conservative. This chapter presents the first non-trivial pointer analysis that deals with all Java language features.

This chapter identifies all problems in performing Andersen's pointer analysis for the full Java language, presents solutions to those problems, and uses a full implementation of the solutions in Jikes RVM for validation and performance evaluation. The results from this work should be transferable to other analyses and to other languages.

Pointer analysis benefits many optimizations, such as inlining, load elimination, code movement, stack allocation, and parallelization. Unfortunately, dynamic class loading, reflection, and native code make ahead-of-time pointer analysis of Java programs impossible.

This chapter presents the first non-trivial pointer analysis that works for all of Java. Most prior papers assume that all classes are known and available ahead of time (e.g., [91, 92, 105, 133]). The few papers that deal with dynamic class loading assume restrictions on reflection and native code [25, 87, 101, 103]. Prior work makes these simplifying assumptions because they are acceptable in some contexts, because dealing with the full generality of Java is difficult, and because the advantages of the analyses often outweigh the disadvantages of only handling a subset of Java.

This chapter describes how to overcome the restrictions of prior work in the context of Andersen's pointer analysis [6], so the benefits become available in the general setting of an executing Java virtual machine. This chapter:

(a) identifies all problems of performing Andersen's pointer analysis for the full Java language,

(b) presents a solution for each of the problems,

(c) reports on a full implementation of the solutions in Jikes RVM,

(d) validates, for a set of benchmark runs, that the list of problems is complete, the solutions are correct, and the implementation works, and

(e) evaluates the efficiency of the implementation.

The performance results show that the implementation is efficient enough for stable long-running applications. However, because Andersen's algorithm has cubic time complexity, and because Jikes RVM, which is itself written in Java, leads to a large code base even for small benchmarks, performance needs improvements for short-running applications. Such improvements are an open challenge; they could be achieved by making Andersen's implementation in Jikes RVM more efficient, or by using a cheaper analysis.

The contributions from this work should be transferable to

- Other analyses: Andersen's analysis is a whole-program analysis consisting of two steps: modeling the code and computing a fixed-point on the model. Several other algorithms follow the same pattern, such as VTA [121], XTA [126], or Das's one level flow algorithm [43]. Algorithms that do not require the second step, such as CHA [44, 51] or Steensgaard's unification-based algorithm [114], are easier to perform in an online setting. Andersen's analysis is flow-insensitive and context-insensitive. While this chapter should also be helpful for performing flow-sensitive or context-sensitive analyses online, those pose additional challenges (multithreading and exceptions, and multiple calling contexts) that need to be addressed.

- Other languages: This chapter shows how to deal with dynamic class loading, reflection, and native code in Java. Other languages have similar features, which pose similar problems for pointer analysis.

10.1 Motivation

Java features such as dynamic class loading, reflection, and native methods prohibit static whole-program analyses. This chapter identifies all Java features that create challenges for pointer analysis; this section focuses just on class loading, and discusses why it precludes static analysis.

10.1.1 It is not known statically *where* a class will be loaded from.

Java allows user-defined class loaders, which may have their own rules for where to look for the bytecode, or even generate it on-the-fly. A static analysis cannot analyze those classes. User-defined class loaders are widely used in production-strength commercial applications, such as Eclipse [125] and Tomcat [124].

10.1.2 It is not known statically *which* class will be loaded.

Even an analysis that restricts itself to the subset of Java without *user-defined* class loaders cannot be fully static, because code may still load statically unknown classes with the *system* class loader. This is done by invoking Class.*forName*(String *name*), where *name* can be computed at runtime. For example, a program may compute the localized calendar class name by reading an environment variable. One approach to dealing with this issue would be to assume that all calendar classes may be loaded. This would result in a less precise solution, if, for example, at each customer's site, only one calendar class is loaded. Even worse, the relevant classes may be available only in the execution environment, and not in the development environment. Only an online analysis could analyze such a program.

10.1.3 It is not known statically *when* a given class will be loaded.

If the classes to be analyzed are available only in the execution environment, but Class.*forName* is not used, one could imagine avoiding *static* analysis by attempting a whole-program analysis during JVM *start-up*, long before the analyzed classes will be needed. The Java specification says it should appear to the user as if class loading is lazy, but a JVM could just pretend to be lazy by showing only the effects of lazy loading, while actually being eager. This is difficult to engineer in practice, however. One would need a deferral mechanism for various visible effects of class loading. An example for such a visible effect would be a static field initialization of the form

static HashMap *hashMap* = **new** HashMap(Constants.CAPACITY);

Suppose that Constants.CAPACITY has the illegal value −1. The effect, an ExceptionInInitializerError, should only become visible when the class containing the static field is loaded. Furthermore, *hashMap* should be initialized after CAPACITY, to ensure that the latter receives the correct value. Loading classes eagerly and still preserving the proper (lazy) class loading semantics is challenging.

10.1.4 It is not known statically *whether* a given class will be loaded.

Even if one ignores the order of class loading, and handles only a subset of Java without *explicit* class loading, *implicit* class loading still poses problems for static analyses. A JVM implicitly loads a class the first time executing code refers to it, for example, by creating an instance of the class. Whether a program will load a given class is undecidable, as Figure 10.1 illustrates: a run of "`java Main`" does not load class C; a run of "`java Main anArgument`" loads class C, because Line 5 creates an instance of C. One can observe this by whether Line 10 in the static initializer prints its message. In this example, a static analysis would have to conservatively assume that class C will be loaded, and to analyze it. In general, a static whole-program analysis would have to analyze many more classes than necessary, making it inefficient (analyzing more classes costs time and space) and less precise (the code in those classes may exhibit behavior never encountered at runtime).


```
 1: class Main {
 2:     public static void main(String[] argv) {
 3:         C v = null;
 4:         if (argv.length > 0)
 5:             v = new C();
 6:     }
 7: }
 8: class C {
 9:     static {
10:         System.out.println("loaded class C");
11:     }
12: }
```

Figure 10.1: Class loading example.

10.2 Related work

This chapter shows how to enhance Andersen's pointer analysis to analyze the full Java programming language. Section 10.2.1 puts Andersen's pointer analysis in context. Section 10.2.2 discusses related work on online, interprocedural analyses. Section 10.2.3 discusses related work on using Andersen's analysis for Java. Finally, Section 10.2.4 discusses work related to the validation methodology.

10.2.1 Static pointer analyses

The body of literature on pointer analyses is vast [73]. At one extreme, exemplified by Steensgaard [114] and type-based analyses [48, 65, 126], the analyses are fast, but imprecise. At the other extreme, exemplified by shape analyses [71, 107], the analyses are slow, but precise enough to discover the shapes of many data structures. In between these two extremes there are many pointer analyses, offering different cost-precision tradeoffs.

The goal of the research presented in this chapter was to choose a well-known analysis and to extend it to handle all features of Java. This goal was motivated by the need to build a pointer analysis to support connectivity-based garbage collection, for which type-based analyses are too imprecise (Section 9.5.1). Liang et al. [93] report that it would be very hard to significantly improve the precision of Andersen's analysis without biting into the much more expensive shape analysis. This left a choice between Steensgaard's [114] and Andersen's [6] analysis. Andersen's analysis is less efficient, but more precise [74, 109]. We decided to use Andersen's analysis, because it poses a superset of the Java-specific challenges posed by Steensgaard's analysis, leaving the latter (or points in between) as a fall-back option.

10.2.2 Online interprocedural analyses

An *online* interprocedural analysis is an interprocedural analysis that occurs during execution, and thus, can correctly deal with dynamic class loading.

10.2.2.1 Demand-driven interprocedural analyses

A number of pointer analyses are demand-driven, but not online [2, 30, 32, 69, 90, 132]. All of these analyses build a representation of the static whole program, but then compute exact solutions only for parts of it, which makes them more scalable. None of these papers discuss issues specific to dynamic class loading.

10.2.2.2 Incremental interprocedural analyses

Another related area of research is incremental interprocedural analysis [27, 41, 62, 64]. The goal of this line of research is to avoid a reanalysis of the complete program when a change is made after an interprocedural analysis has been performed. This chapter differs in that it focuses on the dynamic semantics of the Java programming language, not programmer modifications to the source code.

10.2.2.3 Extant analysis

Sreedhar, Burke, and Choi [113] describe extant analysis, which finds parts of the static whole program that can be safely optimized ahead of time, even when new classes may be loaded later. It is not an online analysis, but reduces the need for one in settings where much of the program is available statically.

10.2.2.4 Analyses that deal with dynamic class loading

Below is a discussion of some analyses that deal with dynamic class loading. None of these analyses deals with reflection or JNI, or validate their analysis results. Furthermore, all are less precise than Andersen's analysis.

Pechtchanski and Sarkar [101] present a framework for interprocedural whole-program analysis and optimistic optimization. They discuss how the analysis is triggered (when newly loaded methods are compiled), and how to keep track of what to de-optimize (when optimistic assumptions are invalidated). They also present an example online interprocedural type analysis. Their analysis does not model value flow through parameters, which makes it less precise, as well as easier to implement, than Andersen's analysis.

Bogda and Singh [25] and King [87] adapt Ruf's escape analysis [106] to deal with dynamic class loading. Ruf's analysis is unification-based, and thus less precise than Andersen's analysis. Escape analysis is a simpler problem than pointer analysis because the impact of a method is independent of its parameters and the problem doesn't require a unique representation for each heap object [36]. Bogda and Singh discuss tradeoffs of when to trigger the analysis, and whether to make optimistic or pessimistic assumptions for optimization. King focuses on a specific client, a garbage collector with thread-local heaps, where local collections require no synchronization. Whereas Bogda and Singh use a call graph based on capturing call edges at their first dynamic execution, King uses a call graph based on rapid type analysis [14].

Qian and Hendren [103], in work concurrently with the work for this chapter, adapt Tip and Palsberg's XTA [126] to deal with dynamic class loading. The main contribution of their paper is a low-overhead call edge profiler, which yields a precise call graph on which XTA is based. Even though XTA is weaker than Andersen's analysis, both have separate constraint generation and constraint propagation steps, and thus pose similar problems. Qian and Hendren solve the problems posed by dynamic class loading similarly to the way this chapter solves them; for example, their approach to unresolved references is analogous to the approach in Section 10.3.5.

10.2.3 Andersen's analysis for static Java

A number of papers describe how to use Andersen's analysis for Java [91, 92, 105, 133]. None of these deal with dynamic class loading. Nevertheless, they do present solutions for various other features of Java that make pointer analyses difficult (object fields, virtual method invocations, etc.).

Rountev, Milanova, and Ryder [105] formalize Andersen's analysis for Java using set constraints, which enables them to solve it with Bane (Berkeley ANalysis Engine) [50]. Liang, Pennings, and Harrold [92] compare both Steensgaard's and Andersen's analysis for Java, and evaluate trade-offs for handling fields and the call graph. Whaley and Lam [133] improve the efficiency of Andersen's analysis by using implementation techniques from CLA [70], and improve the precision by adding flow-sensitivity for local variables. Lhoták and Hendren [91] present Spark (Soot Pointer Analysis Research Kit), an implementation of Andersen's analysis in Soot [131], which provides precision and efficiency tradeoffs for various components.

Prior work on implementing Andersen's analysis differs in how it represents constraint graphs. There are many alternatives, and each one has different cost/benefit tradeoffs. Section 10.3.2.1 discusses these alternatives.

10.2.4 Validation methodology

The validation methodology compares points-to sets computed by the analysis to actual pointers at runtime. This is similar to limit studies that other researchers have used to evaluate and debug various compiler analyses [48, 88, 93].

10.3 Algorithm

Section 10.3.1 presents the architecture for performing Andersen's pointer analysis online. The subsequent sections discuss parts of the architecture that deal with: constraint finding (10.3.2), call graph building (10.3.3), constraint propagation (10.3.4), type resolution (10.3.5), and other constraint generating events (10.3.6).

10.3.1 Architecture

As mentioned in Section 10, Andersen's algorithm has two steps: finding the constraints that model the code semantics of interest, and propagating these constraints

until a fixed point is reached. In an offline setting, the first step requires a scan of the program and its call graph. In an online setting, this step is more complex, because parts of the program are "discovered" during execution of various VM events. Figure 10.2 shows the architecture for performing Andersen's pointer analysis online. The events during virtual machine execution (left column) generate inputs to the analysis. The analysis (dotted box) consists of four components (middle column) that operate on shared data structures (right column). Clients (bottom) trigger the constraint propagator component of the analysis, and consume the outputs. The outputs are represented as points-to sets in the constraint graph. In an online setting, the points-to sets conservatively describe the pointers in the program until there is an addition to the constraints.

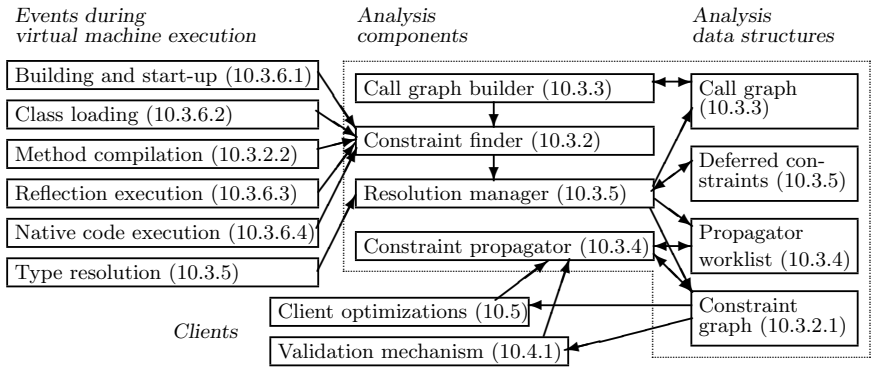


Figure 10.2: Architecture for performing Andersen's pointer analysis online. The numbers in parentheses refer to sections in this chapter.

When used offline, Andersen's analysis requires only a part of the architecture in Figure 10.2. In an offline setting, the only input comes from method compilation. It is used by the constraint finder and the call graph builder to create a constraint graph. After that, the constraint propagator finds a fixed-point on the constraint graph. The results are consumed by clients.

Four additions to the architecture make Andersen's analysis work online:

Building the call graph online. Andersen's analysis relies on a call graph for interprocedural constraints. This chapter uses an online version of CHA (class hierarchy analysis [44, 51]) for the call graph builder. CHA is an offline whole-program analysis, Section 10.3.3 describes how to make it work online.

Supporting re-propagation. Method compilation and other constraint-generating events happen throughout the execution. Where an offline analysis can propagate once after all constraints have been found, the online analysis has to propagate whenever a client needs points-to information and new constraints have been created since the last propagation. Section 10.3.4 describes how the propagator starts with its previous solution and a worklist of changed parts in the constraint graph to avoid incurring the full propagation cost every time.

Table 10.1: Constraint graph representation.

Node kind	Represents concrete entities	Flow sets	Points-to sets
h-node	Set of heap objects, e.g., all objects allocated at a particular allocation site	none	none
v-node	Set of program variables, e.g., a static variable, or all occurrences of a local variable	flowTo[v], flowTo[$v.f$]	pointsTo[h]
$h.f$-node	Instance field f of all heap objects represented by h	none	pointsTo[h]
$v.f$-node	Instance field f of all h-nodes pointed to by v	flowFrom[v], flowTo[v]	none

Supporting unresolved types. The constraint finder may find constraints that involve as-yet unresolved types. But both the call graph builder and the propagator rely on resolved types for precision; for example, the propagator filters points-to sets by types. Section 10.3.5 describes how the resolution manager defers communicating constraints from the constraint finder to other analysis components until the involved types are resolved.

Capturing more input events. A pointer analysis for Java has to deal with features such as reflection and native code, in addition to dynamic class loading. Section 10.3.6 describes how to handle all the other events during virtual machine execution that may generate constraints.

10.3.2 Constraint finder

Section 10.3.2.1 describes the constraint graph data structure, which models the data flow of the program. Section 10.3.2.2 describes how code is translated into constraints at method compilation time. The approach to representing the constraint graph and analyzing code in this chapter combines ideas from various earlier papers on offline implementation of Andersen's analysis.

10.3.2.1 Constraint graph

The constraint graph has four kinds of nodes that participate in constraints. The constraints are stored as sets at the nodes. Table 10.1 describes the nodes, introducing the notation that is used in the remainder of this chapter, and shows which sets are stored at each node. The node kinds in "[$\cdots$]" are the kinds of nodes in the set.

Flow-to sets (Column 3 of Table 10.1) represent a flow of values (assignments, parameter passing, etc.), and are stored with v-nodes and $v.f$-nodes. For example, if $v'.f \in \text{flowTo}(v)$, then v's pointer r-value may flow to $v'.f$. Flow-from sets are the inverse of flow-to sets. In the example, this would mean $v \in \text{flowFrom}(v'.f)$.

Points-to sets (Column 4 of Table 10.1) represent the set of objects (r-values) that a pointer (l-value) may point to, and are stored with v-nodes and $h.f$-nodes.

Since it stores points-to sets with $h.f$-nodes instead of $v.f$-nodes, the analysis is *field sensitive* [91].

The constraint finder models program code by v-nodes, $v.f$-nodes, and their flow sets. Based on these, the propagator computes the points-to sets of v-nodes and $h.f$-nodes. For example, if a client of the pointer analysis is interested in whether a variable p may point to objects allocated at an allocation site a, it checks whether the h-node for a is an element of the points-to set of the v-node for p.

Each h-node has a map from fields f to $h.f$-nodes (i.e., the nodes that represent the instance fields of the objects represented by the h-node). In addition to language-level fields, each h-node has a special node $h.f_{\text{td}}$ that represents the field containing the reference to the type descriptor for the heap node. A type descriptor is implemented as a TIB[p.188] object in Jikes RVM, and thus, must be modeled by the analysis. For each h-node representing arrays of references, there is a special node $h.f_{\text{elems}}$ that represents all of their elements. Thus, the analysis does not distinguish between different elements of an array.

There are many alternatives for storing the flow and points-to sets. For example, the analysis described here represents the data flow between v-nodes and $h.f$-nodes implicitly, whereas Bane represents it explicitly [54, 105]. Thus, the analysis saves space compared to Bane, but may have to perform more work at propagation time. As another example, CLA [70] stores reverse points-to sets at h-nodes, instead of storing forward points-to sets at v-nodes and $h.f$-nodes. The forward points-to sets are implicit in CLA and must therefore be computed after propagation to obtain the final analysis results. These choices affect both the time and space complexity of the propagator. As long as it can infer the needed sets during propagation, an implementation can decide which sets to represent explicitly. In fact, a representation may even store some sets redundantly: for example, to obtain efficient propagation, the representation for this chapter uses redundant flow-from sets.

Finally, there are many choices for how to implement the sets. The Spark paper evaluates various data structures for representing points-to sets [91], finding that hybrid sets (using lists for small sets, and bit-vectors for large sets) yield the best results. The shared bit-vector implementation from CLA [68] turns out to be even more efficient than the hybrid sets used by Spark.

10.3.2.2 Method compilation

The left column of Figure 10.2 shows the various events during virtual machine execution that invoke the constraint finder. This section is only concerned with finding intraprocedural constraints during method compilation; later sections discuss other kinds of events.

The intraprocedural constraint finder analyzes the code of a method, and models it in the constraint graph. It is a flow-insensitive pass of the optimizing compiler of Jikes RVM, operating on the high-level register-based intermediate representation (HIR). HIR decomposes access paths by introducing temporaries, so that no access path contains more than one pointer dereference.

Column "Actions" in Table 10.2 gives the actions of the constraint finder when it encounters the statement in Column "Statement". Column "Represent constraints" shows the constraints implicit in the actions of the constraint finder using mathematical notation.

Table 10.2: Intraprocedural constraint finder.

Statement		Actions	Represent constraints
$v' = v$	(move $v \to v'$)	flowTo(v).add(v')	pointsTo(v) $\subseteq$ pointsTo(v')
$v' = v.f$	(load $v.f \to v'$)	flowTo($v.f$).add(v')	$\forall h \in$ pointsTo(v) : pointsTo($h.f$) $\subseteq$ pointsTo(v')
$v'.f = v$	(store $v \to v'.f$)	flowTo(v).add($v'.f$), flowFrom($v'.f$).add(v)	$\forall h \in$ pointsTo(v') : pointsTo(v) $\subseteq$ pointsTo($h.f$)
ℓ: $v =$ **new**...	(alloc $h_\ell \to v$)	pointsTo(v).add(h_ℓ)	$\{h_\ell\} \subseteq$ pointsTo(v)

In addition to the actions in Table 10.2, the analysis needs to address some more issues during method compilation.

10.3.2.2.1 Unoptimized code

The intraprocedural constraint finder is implemented as a pass of the Jikes RVM optimizing compiler. However, as discussed in Section 2.1.1, Jikes RVM compiles some methods only with a baseline compiler, which does not use a representation that is amenable to constraint finding. The analysis handles such methods by running the constraint finder as part of a truncated optimizing compilation. Other virtual machines, where some code is not compiled at all, but interpreted, can take a similar approach.

10.3.2.2.2 Recompilation of methods

Many JVMs, including Jikes RVM, may recompile a method (at a higher optimization level) if it executes frequently. The recompiled methods may have new variables or code introduced by optimizations (such as inlining). Since each inlining context of an allocation site is modeled by a separate h-node, the analysis generates new constraints for the recompiled methods and integrates them with the constraints for any previously compiled versions of the method.

10.3.2.2.3 Magic

Jikes RVM has some internal "magic" operations, for example, to allow direct manipulation of pointers. The compilers expand magic in special ways directly into low-level code. Likewise, the analysis expands magic in special ways directly into constraints.

10.3.3 Call graph builder

For each call-edge, the analysis generates constraints that model the data flow through parameters and return values. Parameter passing is modeled as a move from actuals (at the call-site) to formals (of the callee). Each return statement in a method m is modeled as a move to a special v-node $v_{\text{retval}(m)}$. The data flow of the return value to the call-site is modeled as a move to the v-node that receives the result of the call.

The analysis uses CHA (Class Hierarchy Analysis [44, 51]) to find call-edges. A more precise alternative to CHA is to construct the call graph on-the-fly based on

the results of the pointer analysis. We decided against that approach because prior work indicated that the modest improvement in precision does not justify the cost in efficiency [91]. In work concurrent with the work for this chapter, Qian and Hendren developed an even more precise alternative based on low-overhead profiling [103].

CHA is a static whole-program analysis, but to support Andersen's analysis online, CHA must also run online, i.e., deal with dynamic class loading. The key to solving this problem is the observation that for each call-edge, either the call-site is compiled first, or the callee is compiled first. The constraints for the call-edge are added when the second of the two is compiled. This works as follows:

- When encountering a method $m(v_{\text{formal}_1(m)}, \ldots, v_{\text{formal}_n(m)})$, the call graph builder
  - creates a tuple $I_m = \langle v_{\text{retval}(m)}, v_{\text{formal}_1(m)}, \ldots, v_{\text{formal}_n(m)} \rangle$ for m as a callee,
  - finds all corresponding tuples for matching call-sites that have been compiled in the past, and adds constraints to model the moves between the corresponding v-nodes in the tuples, and
  - stores the tuple I_m for lookup on behalf of call-sites that will be compiled in the future.
- When encountering a call-site $c : v_{\text{retval}(c)} = m(v_{\text{actual}_1(c)}, \ldots, v_{\text{actual}_n(c)})$, the call graph builder
  - creates a tuple $I_c = \langle v_{\text{retval}(c)}, v_{\text{actual}_1(c)}, \ldots, v_{\text{actual}_n(c)} \rangle$ for call-site c,
  - looks up all corresponding tuples for matching callees that have been compiled in the past, and adds constraints to model the moves between the corresponding v-nodes in the tuples, and
  - stores the tuple I_c for lookup on behalf of callees that will be compiled in the future.

Besides parameter passing and return values, there is one more kind of interprocedural data flow that an analysis needs to model: exception handling. Exceptions lead to flow of values (the exception object) between the site that throws an exception and the catch clause that catches the exception. For simplicity, the initial prototype assumes that any throws can reach any catch clause; type filtering eliminates many of these possibilities later on. One could easily imagine making this more precise, for example by assuming that throws can only reach catch clauses in the current method or its (transitive) callers.

10.3.4 Constraint propagator

The propagator propagates points-to sets following the constraints that are implicit in the flow sets until the points-to sets reach a fixed point. In order to avoid wasted work, the algorithm maintains two pieces of information, a worklist of v-nodes and

isCharged-bits on $h.f$-nodes, that enable it to propagate only the changed points-to sets at each iteration (rather than propagating all points-to sets). The worklist contains v-nodes whose points-to sets have changed and thus need to be propagated, or whose flow sets have changed and thus the points-to sets need to be propagated to additional nodes. The constraint finder initializes the worklist.

The algorithm in Figure 10.3, which is a variation of the algorithm from SPARK [91], implements the constraint propagator component of Figure 10.2.

```
 1: while worklist not empty, or isCharged(h.f) for any h.f-node
 2:     while worklist not empty
 3:         remove node v from worklist
 4:         for each v' ∈ flowTo(v)                          // move v → v'
 5:             pointsTo(v').add(pointsTo(v))
 6:             if pointsTo(v') changed, add v' to worklist
 7:         for each v'.f ∈ flowTo(v)                        // store v → v'.f
 8:             for each h ∈ pointsTo(v')
 9:                 pointsTo(h.f).add(pointsTo(v))
10:                 if pointsTo(h.f) changed, isCharged(h.f) ← true
11:         for each field f of v,
12:             for each v' ∈ flowFrom(v.f)                  // store v' → v.f
13:                 for each h ∈ pointsTo(v)
14:                     pointsTo(h.f).add(pointsTo(v'))
15:                     if pointsTo(h.f) changed, isCharged(h.f) ← true
16:             for each v' ∈ flowTo(v.f)                    // load v.f → v'
17:                 for each h ∈ pointsTo(v)
18:                     pointsTo(v').add(pointsTo(h.f))
19:                     if pointsTo(v') changed, add v' to worklist
20:     for each v.f
21:         for each h ∈ pointsTo(v), if isCharged(h.f)
22:             for each v' ∈ flowTo(v.f)                    // load v.f → v'
23:                 pointsTo(v').add(pointsTo(h.f))
24:                 if pointsTo(v') changed, add v' to worklist
25:     for each h.f
26:         isCharged(h.f) ← false
```

Figure 10.3: Constraint propagator.

The propagator puts a v-node on the worklist when its points-to set changes. Lines 4-10 propagate the v-node's points-to set to nodes in its flow-to sets. Lines 11-19 update the points-to set for all fields of objects pointed to by the v-node. This is necessary because for the h-nodes that have been newly added to v's points-to set, the flow to and from $v.f$ carries over to the corresponding $h.f$-nodes. Line 12 relies on the redundant flow-from sets.

The propagator sets the isCharged-bit of an $h.f$-node to true when its points-to set changes. To discharge an $h.f$-node, the algorithm needs to consider all flow-to edges from all $v.f$-nodes that represent it (lines 20-24). This is why it does not keep a worklist of charged $h.f$-nodes: to find their flow-to targets, it needs to iterate over $v.f$-nodes anyway. This is the only part of the algorithm that iterates over all ($v.f$-) nodes: all other parts of the algorithm attempt to update points-to sets while

visiting only nodes that are relevant to the points-to sets being updated.

To improve the efficiency of this iterative part, the implementation uses a cache that remembers the charged nodes in shared points-to sets. The cache speeds up the loops at Lines 20 and 21 by an order of magnitude.

The propagator performs on-the-fly filtering by types: it only adds an h-node to a points-to set of a v-node or $h.f$-node if it represents heap objects of a subtype of the declared type of the variable or field. Lhoták and Hendren found that this helps keep the points-to sets small, improving both precision and efficiency of the analysis [91]. Our experiences confirm this observation.

The propagator creates $h.f$-nodes lazily the first time it adds elements to their points-to sets, in lines 9 and 14. It only creates $h.f$-nodes if instances of the type of h have the field f. This is not always the case, as the following example illustrates. Let A, B, C be three classes such that C is a subclass of B, and B is a subclass of A. Class B declares a field f. Let h_A, h_B, h_C be h-nodes of type A, B, C, respectively. Let v be a v-node of declared type A, and let v.pointsTo $= \{h_A, h_B, h_C\}$. Now, data flow to $v.f$ should add to the points-to sets of nodes $h_B.f$ and $h_C.f$, but there is no node $h_A.f$.

We also experimented with the optimizations partial online cycle elimination [50] and collapsing of single-entry subgraphs [104]. They yielded only modest performance improvements compared to shared bit-vectors [68] and type filtering [91]. Part of the reason for the small payoff may be that the data structures in this chapter do not put $h.f$-nodes in flow-to sets (á la Bane [50]).

10.3.5 Resolution manager

The JVM specification allows a Java method to have unresolved references to fields, methods, and classes [95]. A class reference is resolved when the class is instantiated, when a static field in the class is used, or when a static method in the class is called.

The unresolved references in the code (some of which may never get resolved) create two main difficulties for the analysis.

First, the CHA (class hierarchy analysis) that implements the call graph builder does not work when the class hierarchy of the involved classes is not yet known. Our current approach to this is to be conservative: if, due to unresolved classes, CHA cannot yet decide whether a call edge exists, the call graph builder adds an edge if the signatures match.

Second, the propagator uses types to perform type filtering and also for deciding which $h.f$-nodes belong to a given $v.f$-node. If the involved types are not yet resolved, this does not work. Therefore, the resolution manager defers all flow sets and points-to sets involving nodes of unresolved types, thus hiding them from the propagator:

- When the constraint finder creates an unresolved node, it registers the node with the resolution manager. A node is unresolved if it refers to an unresolved type. An h-node refers to the type of its objects; a v-node refers to its declared type; and a $v.f$-node refers to the type of v, the type of f, and the type in which f is declared.

- When the constraint finder would usually add a node to a flow set or points-to set of another node, but one or both of them are unresolved, it defers the information for later instead. Table 10.3 shows the deferred sets stored at unresolved nodes. For example, if the constraint finder finds that v should point to h, but v is unresolved, it adds h to v's deferred pointsTo set. Conversely, if h is unresolved, it adds v to h's deferred pointedToBy set. If both are unresolved, the points-to information is stored twice.

Table 10.3: Deferred sets stored at unresolved nodes.

Node kind	Flow	Points-to
h-node	none	pointedToBy[v]
v-node	flowFrom[v], flowFrom[$v.f$], flowTo[v], flowTo[$v.f$]	pointsTo[h]
$h.f$-node	there are no unresolved $h.f$-nodes	
$v.f$-node	flowFrom[v], flowTo[v]	none

- When a type is resolved, the resolution manager notifies all unresolved nodes that have registered for it. When an unresolved node is resolved, it iterates over all deferred sets stored at it, and attempts to add the information to the real model that is visible to the propagator. If a node stored in a deferred set is not resolved yet itself, the information will be added in the future when that node gets resolved.

With this design, some constraints will never be added to the model, if their types never get resolved. This saves unnecessary propagator work. Qian and Hendren developed a similar design independently [103].

Before becoming aware of the subtleties of the problems with unresolved references, we used an overly conservative approach: we added the constraints eagerly even when we had incomplete information. This imprecision led to very large points-to sets, which in turn slowed down the analysis prohibitively. Our current approach is both more precise and more efficient.

10.3.6 Other constraint-generating events

This section discusses the remaining events in the left column of Figure 10.2 that serve as inputs to the constraint finder.

10.3.6.1 VM building and start-up

As discussed in Section 2.1.2, Jikes RVM itself is written in Java, and begins execution by loading a *boot image* (a file-based image of a fully initialized VM) of pre-allocated Java objects for the JIT compilers, GC, and other runtime services. These objects live in the same heap as application objects, so an analysis must model them.

The analysis described here models all the *code* in the boot image as usual, with the intraprocedural constraint finder pass from Section 10.3.2.2 and the call

graph builder from Section 10.3.3. The analysis models the *data snapshot* of the boot image with special boot image h-nodes, and with points-to sets of global v-nodes and boot image $h.f$-nodes. The program that creates the boot image does not maintain a mapping from objects in the boot image to their actual allocation site, and thus, the boot image h-nodes are not allocation sites, instead they are synthesized at boot image writing time. Finally, the analysis propagates on the combined constraint system. This models how the snapshot of the data in the boot image may be manipulated by future execution of the code in the boot image.

The above techniques for correctly handling the boot image can be extended to form a general hybrid offline/online approach, where parts of the application are analyzed offline (as the VM is now) and the rest of the application is handled by the online analysis presented in this work. Such an approach could be useful for applications where the programmer asserts no use of the dynamic language features in parts of the application.

10.3.6.2 Class loading

Even though much of this chapter revolves around making Andersen's analysis work for dynamic class loading, most analysis actions actually happen during other events, such as method compilation or type resolution. The only action that does take place exactly at class loading time is that the constraint finder models the ConstantValue bytecode attribute of static fields with constraints [95, Section 4.5].

10.3.6.3 Reflection execution

Java programs can invoke methods, access and modify fields, and instantiate objects using reflection. Although approaches such as String analysis [37] could predict which entities are manipulated in special cases, this problem is undecidable in the general case. Thus, when compiling code that uses reflection, there is no way of determining which methods will be called, which fields manipulated, or which classes instantiated at runtime.

One solution is to assume the worst case. We felt that this was too conservative and would introduce significant imprecision into the analysis for the sake of a few operations that were rarely executed. Other pointer analyses for Java side-step this problem by requiring users of the analysis to provide hand-coded models describing the effect of the reflective actions [91, 133].

This chapter uses the solution of handling reflection when the code is actually executed. This is done by instrumentations of the virtual machine service that handles reflection with code that adds constraints dynamically. For example, if reflection stores into a field, the constraint finder observes the actual source and target of the store and generates a constraint that captures the semantics of the store at that time.

This strategy for handling reflection introduces new constraints when the reflective code does something new. Fortunately, that does not happen very often. When reflection has introduced new constraints and a client needs up-to-date points-to results, it must trigger a re-propagation.

10.3.6.4 Native code execution

The Java Native Interface (JNI) allows Java code to interact with dynamically loaded native code. Usually, a JVM cannot analyze that code. Thus, an analysis does not know (i) what values may be returned by JNI methods and (ii) how JNI methods may manipulate data structures of the program.

The approach for this chapter is to be imprecise, but conservative, for return values from JNI methods, while being precise for data manipulation by JNI methods. If a JNI method returns a heap allocated object, the constraint finder assumes that it could return an object from any allocation site. This is imprecise, but easy to implement. The constraint propagation uses type filtering, and thus, will filter the set of heap nodes returned by a JNI method based on types. If a JNI method manipulates data structures of the program, the manipulations must go through the JNI API, which Jikes RVM implements by calling Java methods that use reflection. Thus, JNI methods that make calls or manipulate object fields are handled precisely by the mechanism for reflection.

10.4 Validation

Implementing a pointer analysis for a complicated language and environment such as Java and Jikes RVM is a difficult task: the pointer analysis has to handle numerous corner cases, and missing any of the cases results in incorrect points-to sets. An automatic validation mechanism helped debug the pointer analysis to a high confidence level.

10.4.1 Validation mechanism

Validation of the pointer analysis results occurs at GC (garbage collection) time. As GC traverses each pointer, it checks whether the points-to set captures the pointer: (i) When GC finds a static variable p holding a pointer to an object o, the validation code finds the nodes v for p and h for o. Then, it checks whether the points-to set of v includes h. (ii) When GC finds a field f of an object o holding a pointer to an object o', the validation code finds the nodes h for o and h' for o'. Then, it checks whether the points-to set of $h.f$ includes h'. If either check fails, it prints a warning message.

To make the points-to sets correct at GC time, validation runs propagate the constraints (Section 10.3.4) just before GC starts. As there is no memory available to grow points-to sets at that time, Jikes RVM's garbage collector sets aside some extra space for this purpose.

The validation methodology relies on the ability to map concrete heap objects to h-nodes in the constraint graph. To facilitate this, it adds an extra header word to each heap object that maps it to its corresponding h-node in the constraint graph. For h-nodes representing allocation sites, the allocation routine installs this header word. Only validation runs use the extra word; except for validation, the pointer analysis does not require any change to the object header.

10.4.2 Validation anecdotes

The validation methodology helped find many bugs, some of which were quite subtle. Below are two examples. In both cases, there was more than one way in which bytecode could represent a Java-level construct. Both times, the analysis dealt correctly with the more common case, and the other case was obscure, yet legal. The validation methodology showed where the analysis missed something; without it, we might not even have suspected that something was wrong.

10.4.2.1 Field reference class

In Java bytecode, a field reference consists of the name and type of the field, as well as a class reference to the class or interface "in which the field is to be found" ([95, Section 5.1]). Even for a static field, this may not be the class that declared the field, but a subclass of that class. Originally, we had assumed that it must be the exact class that declared the static field, and had written our analysis accordingly to maintain separate v-nodes for static fields with distinct declaring classes. When the bytecode wrote to a field using a field reference that mentions the subclass, the v-node for the field that mentions the superclass was missing some points-to set elements. That resulted in warnings from our validation methodology. Upon investigating those warnings, we became aware of the incorrect assumption and fixed it.

10.4.2.2 Field initializer attribute

In Java source code, a static field declaration has an optional initialization, for example, "**final static** String s = `"abc"`;". In Java bytecode, this usually translates into initialization code in the class initializer method <*clinit*>() of the class that declares the field. But sometimes, it translates into a ConstantValue attribute of the field instead ([95, Section 4.5]). Originally, we had assumed that class initializers are the only mechanism for initializing static fields, and that we would find these constraints when running the constraint finder on the <*clinit*>() method. But our validation methodology warned us about v-nodes for static fields whose points-to sets were too small. Knowing exactly for which fields that happened, we looked at the bytecode, and were surprised to see that the <*clinit*>() methods didn't initialize the fields. Thus, we found out about the ConstantValue bytecode attribute, and added constraints when class loading parses and executes that attribute (Section 10.3.6.2).

10.5 Clients

This section investigates two example clients of the analysis, and how they can deal with the dynamic nature of the analysis.

Method inlining can benefit from pointer analysis: if the points-to set elements of v all have the same implementation of a method m, the call $v.m()$ has only one possible target. Modern JVMs [8, 38, 99, 120] typically use a dual execution strategy, where each method is initially either interpreted or compiled without optimizations.

No inlining is performed for such methods. Later, an optimizing compiler that may perform inlining recompiles the minority of frequently executing methods. Because inlining is not performed during the initial execution, the analysis does not need to propagate constraints until the optimizing compiler needs to make an inlining decision.

Since the results of the pointer analysis may be invalidated by any of the events in the left column of Figure 10.2, an inlining client must be prepared to invalidate inlining decisions. Techniques such as code patching [38] and on-stack replacement [52, 83] support invalidation. If instant invalidation is needed, the analysis must repropagate every time it finds new constraints. There are also techniques for avoiding invalidation of inlining decisions, such as pre-existence based inlining [45] and guards [10, 84], that would allow the analysis to be lazy about repropagating after it finds new constraints.

CBGC (connectivity-based garbage collection) is the topic of this dissertation. CBGC uses pointer analysis results to partition heap objects such that connected objects are in the same partition, and the pointer analysis can guarantee the absence of certain cross-partition pointers. CBGC exploits the observation that connected objects tend to die together [82], and certain subsets of partitions can be collected while completely ignoring the rest of the heap.

CBGC must know the partition of an object at allocation time. However, CBGC can easily combine partitions later if the pointer analysis finds that they are strongly connected by pointers (Section 5.2.3). Thus, there is no need to perform a full propagation at object allocation time. However, CBGC does need full conservative points-to information when performing a garbage collection; thus, CBGC needs to request a full propagation before collecting. Between collections, CBGC does not need conservative points-to information.

10.6 Performance

This section evaluates the efficiency of the pointer analysis implementation in Jikes RVM 2.2.1. Prior work (e.g., [91]) has evaluated the precision of Andersen's analysis. In addition to the analysis itself, the modified version of Jikes RVM includes the validation mechanism from Section 10.4. Besides the analysis and validation code, the implementation also includes number of profilers and tracers to collect the results presented in this section. For example, at each yield-point (method prologue or loop back-edge), a stack walk determines whether the yield-point belongs to analysis or application code, and counts it accordingly. All experiments use a 2.4GHz Pentium 4 with 2GB of memory running Linux, kernel version 2.4.

Since Andersen's analysis has cubic time complexity and quadratic space complexity (in the size of the code), optimizations that increase the size of the code can dramatically increase the constraint propagation time. In our experience, aggressive inlining can increase constraint propagation time by up to a factor of 5 for the benchmarks. In default mode, Jikes RVM performs inlining (and optimizations) only inside the hot application methods, but is more aggressive about methods in

Table 10.4: Benchmark programs.

Program	Analyzed methods	Loaded classes	Run time
null	*15,598*	*1,363*	*1s*
javalex	15,728	1,389	37s
compress	15,728	1,391	14s
db	15,746	1,385	28s
mtrt	15,858	1,404	14s
mpegaudio	15,899	1,429	27s
jack	15,962	1,434	21s
richards	15,963	1,440	4s
hsql	15,992	1,424	424s
jess	16,158	1,527	29s
javac	16,464	1,526	66s
xalan	17,057	1,716	10s

the boot image. For the experiments in this chapter, Jikes RVM is more cautious about inlining inside boot image methods by using a FastAdaptiveMarkSweep image and disabling inlining at build time. During benchmark execution, Jikes RVM does, however, perform inlining for hot boot image methods when recompiling them.

10.6.1 Benchmark characteristics

This chapter uses 12 of the Java benchmarks described in Section 2.2. Table 10.4 gives some benchmark characteristics relevant for pointer analysis. Column "Analyzed methods" gives the number of methods analyzed. A method is analyzed when it is part of the boot image, or when the program executes it for the first time. The analyzed methods include the benchmark's methods, library methods called by the benchmark, and methods belonging to Jikes RVM itself. The benchmark null provides a baseline: its data represents approximately the amount that Jikes RVM adds to the size of the application. This data is approximate because, for example, some of the methods called by the optimizing compiler may also be used by the application (e.g., methods on container classes). Column "Loaded classes" gives the number of classes loaded by the benchmarks. Once again, the number of loaded classes for the benchmark null provides a baseline. Finally, Column "Run time" gives the run time for the benchmarks using the same configuration of Jikes RVM as for the rest of this chapter, but without the pointer analysis.

The Jikes RVM methods and classes account for a significant portion of the code in the benchmarks. Thus, the analysis has to deal with much more code than it would have to in a JVM that is not written in Java. On the other hand, writing the analysis itself in Java had significant software engineering benefits; for example, the analysis relies on garbage collection for its data structures. In addition, the absence of artifical boundaries between the analysis, other parts of the runtime system, and the application exposes more opportunities for optimizations. Current trends show that the benefits of writing system code in a high-level, managed, language are

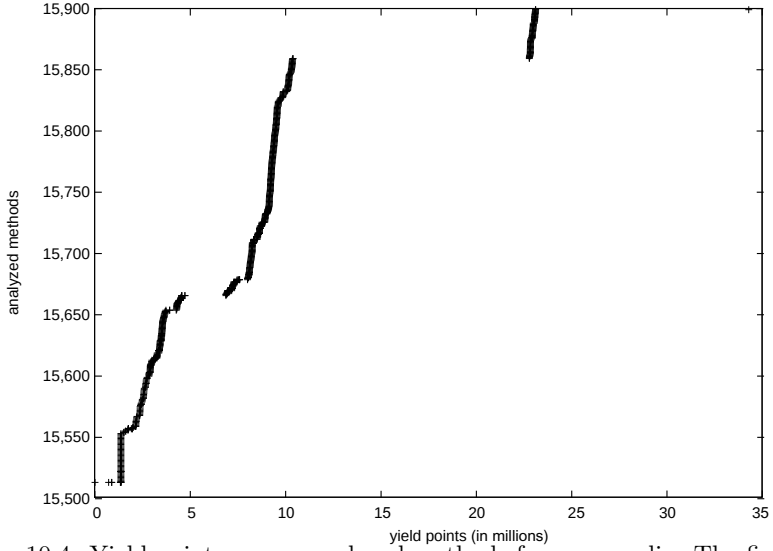


Figure 10.4: Yield-points versus analyzed methods for mpegaudio. The first shown data point is the *main*() method.

gaining wider recognition. For example, Microsoft is pushing towards implementing more of Windows in managed code.

Figure 10.4 shows how the number of analyzed method increase over a run of the mpegaudio benchmark. The x-axis represents time measured by the number of thread yield-points encountered in a run. There is a thread yield-point in the prologue of every method and in every loop. The figure ignores yield-points that occur in analysis code (this would be hard to do if one used real time for the x-axis). The y-axis starts at 15,500: all methods analyzed before the first method in this graph are in the boot image and are thus analyzed once for all benchmarks. The graphs for other benchmarks have a similar shape, and are therefore omitted.

Figure 10.4 shows that there are two significant stages (around the 10 and 25 million yield-point marks) when the application is executing only methods that it has encountered before. At other times, the application encounters new methods as it executes. We expect that for longer running benchmarks (e.g., a webserver that runs for days), the number of analyzed methods stabilizes after a few minutes of run time. That point may be an ideal time to propagate the constraints and use the results to perform optimizations.

10.6.2 Analysis cost

The analysis has two main costs: constraint finding and constraint propagation. Constraint finding happens whenever the virtual machine compiles a new method, loads a new class, etc. Constraint propagation happens whenever a client of the pointer analysis needs points-to information. Define *eager* propagation to be propagation after every event from the left column of Figure 10.2, if it generated new

Table 10.5: Total allocation (in megabytes).

Benchmark	Eager	At GC	Lazy	No propagation	No analysis
null	*48.5*	*48.1*	*48.8*	*13.5*	*9.7*
javalex	621.7	104.7	110.6	70.0	111.8
compress	416.2	230.0	167.0	129.3	130.2
db	394.4	213.8	151.0	112.7	113.6
mtrt	721.9	303.8	240.5	201.5	172.9
mpegaudio	755.9	145.8	83.1	42.8	137.0
jack	1,782.4	418.4	354.8	309.2	322.8
richards	1,117.8	61.3	67.7	26.6	12.6
hsql	4,047.0	3,409.6	3,343.8	3,291.1	3,444.6
jess	4,694.8	458.0	394.4	341.4	398.3
javac	2,023.0	450.4	381.3	328.2	429.3
xalan	6,074.9	166.4	200.4	131.5	37.6

constraints. Define *lazy* propagation to be propagation that occurs just once at the end of the program execution.

10.6.2.1 Cost in space

Table 10.5 shows the total allocation for the benchmark runs. Column "No analysis" gives the number of megabytes allocated by the program without the analysis. Column "No propagation" gives the allocation when the analysis generates, but does not propagate, constraints. Thus, this column gives the space overhead of just representing the constraints. Columns "Eager", "Lazy", and "At GC" give the allocation when using eager, lazy, and at GC propagation. The difference between these and the "No propagation" column represents the overhead of representing the points-to sets. Sometimes doing more work actually reduces the amount of total allocation (e.g., mpegaudio allocates more without any analysis than with lazy propagation). This phenomenon occurs because the analysis is interleaved with the execution of the benchmark program, and thus the Jikes RVM adaptive optimizer optimizes different methods with the analysis than without the analysis.

Finally, since the boot image needs to include constraints for the code and data in the boot image, the analysis inflates the boot image size from 31.5 megabytes to 73.4 megabytes.

10.6.2.2 Cost of constraint finding

Table 10.6 gives the percentage of overall execution time spent in generating constraints from methods (Column "Analyzing methods") and from resolution events (Column "Resolving classes and arrays"). These executions did not run any propagations. Table 10.6 shows that generating constraints for methods is the dominant part of constraint generation. Also, as the benchmark run time increases, the percentage of time spent in constraint generation decreases. For example, the time

Table 10.6: Percent of execution time in constraint finding.

Program	Analyzing methods	Resolving classes and arrays
null	*69.16%*	*3.68%*
javalex	2.02%	0.39%
compress	5.00%	1.22%
db	1.77%	0.39%
mtrt	7.68%	1.70%
mpegaudio	6.23%	6.04%
jack	6.13%	2.10%
richards	21.98%	5.88%
hsql	0.29%	0.09%
jess	5.59%	1.24%
javac	3.20%	1.60%
xalan	26.32%	8.66%

spent in constraint finding is a negligible percentage of the run time for the longest running benchmark, hsql.

10.6.2.3 Cost of propagation

Table 10.7 shows the cost of propagation. Columns "Count" give the number of propagations that occur in the benchmark runs. Columns "Time" give the arithmetic mean ± standard deviation of the time (in seconds) it takes to perform each propagation. Table 10.7 includes the lazy propagation data to give an approximate sense for how long the propagation would take if one were to use a static pointer analysis. Recall, however, that these numbers are still not comparable to static analysis numbers of these benchmarks in prior work, since, unlike them, the analysis in this chapter also analyzes the Jikes RVM compiler and other system services.

Table 10.7 shows that the mean pause time due to eager propagation varies between 3.8 and 16.8 seconds for the real benchmarks. In contrast, a full (lazy) propagation is much slower. Thus, the algorithm is effective in avoiding work on parts of the program that have not changed since the last propagation.

Other results (omitted) showed that the propagation cost did not depend on which of the events in the left column of Figure 10.2 generated new constraints that were the reason for the propagation.

Figure 10.5 presents the spread of propagation times for the javac benchmark. A point (x,y) in this graph says that propagation "x" took "y" seconds. Out of 1,107 propagations in javac, 524 propagations take under 1 second. The remaining propagations are much more expensive (10 seconds or more), thus increasing the average. The figure also shows that more expensive propagations occur later in the execution. The omitted graphs for other benchmarks have a similar shape. Although the figure presents the data for eager propagation, clients of the analysis do not necessarily require eager propagation (Section 10.5).

As expected, the columns for propagation at GC in Table 10.7 show that if the

Table 10.7: Propagation statistics (times in seconds).

	Eager		At GC		Lazy	
Program	Count	Time	Count	Time	Count	Time
null	*1*	*135.6±0.0*	*1*	*120.7±0.0*	*1*	*137.8±0*
javalex	166	13.6±22.0	1	120.7±0.0	1	158.4±0
compress	127	8.6±18.7	3	104.7±23.6	1	142.8±0
db	140	10.0±20.2	3	106.1±24.8	1	144.5±0
mtrt	262	5.5±14.4	3	106.8±24.7	1	148.0±0
mpegaudio	317	5.5±13.4	3	105.4±24.1	1	144.3±0
jack	392	10.9±17.8	3	114.4±33.8	1	161.8±0
richards	410	3.8±10.9	1	120.8±0.0	1	134.8±0
hsql	391	10.1±20.6	6	76.6±94.8	1	426.7±0
jess	734	16.8±20.5	3	117.7±38.5	1	182.4±0
javac	1,103	12.5±22.9	5	114.3±97.6	1	386.7±0
xalan	1,726	11.2±21.4	1	120.5±0.0	1	464.6±0

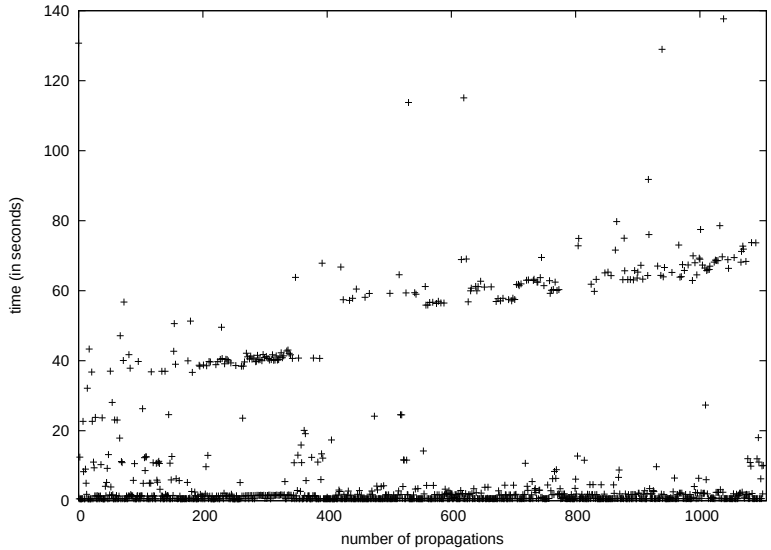


Figure 10.5: Propagation times for javac (eager).

analysis propagates less frequently, the individual propagations are more expensive; they are still on average cheaper than performing a single full propagation at the end of the program run. Recall that, for Java programs, performing a static analysis of the entire program is not possible because what constitutes the "entire program" is not known until it executes to completion.

10.6.3 Understanding the costs of constraint propagation

The speed of the constraint propagator (a few seconds to update points-to information) may be adequate for long-running clients, but may not be feasible for short-running clients. For example, a web server that does not touch new methods after a few minutes of running can benefit from the current analysis: once the web server stops touching new methods, the propagation time of the analysis goes down to zero. Since the benchmark suite for this chapter does not include a server application, we confirmed this behavior by running two benchmarks (javac and mpegaudio) multiple times in a loop: after the first run, there was little to no overhead from constraint finding or constraint propagation (well under 1%). On the other hand, an application that only runs for a few minutes may find the analysis to be prohibitively slow. Profiling the analysis showed that the worklist part (lines 2 to 19 in Figure 10.3) takes up far more of the propagation time than the iterative part (lines 20 to 26 in Figure 10.3). Thus, future work should first focus on the worklist part to improve propagator performance.

10.7 Conclusions

This chapter describes and evaluates the first non-trivial pointer analysis that handles all of Java. Java features such as dynamic class loading, reflection, and native methods introduce many challenges for pointer analyses. Some of these prohibit the use of static pointer analyses. This chapter validates the results of the analysis against actual pointers created during program runs. It evaluates the analysis by measuring many aspects of its performance, including the amount of work the analysis must do at run time. The results show that the analysis is feasible and fast enough for server applications.

Coming up with this analysis was probably the largest hurdle on the way towards performing connectivity-based garbage collection in a Java virtual machine. Chapter 11 describes CBGC in Jikes RVM using the analysis from this chapter.

Chapter 11

CBGC in a Virtual Machine

This chapter describes the design and prototype implementation of a CBGC algorithm in Jikes RVM. It discusses the various challenges that a CBGC in a virtual machine needs to overcome, presenting an engineering solution for each of these challenges. The prototype is a proof of concept: it demonstrates that CBGC can work in a real-world system.

Sections 11.1 to 11.4 describe instances of the four building blocks of any collector in the family of connectivity-based garbage collectors. The prototype of CBGC in Jikes RVM uses an online version of Andersen's pointer analysis for the partitioning (Section 11.1); the greedy chooser (Section 11.3; the combined estimator (Section 11.2); and mostly mark-sweep for partial garbage collection (Section 11.4). Section 11.5 shows experimental results evaluating the performance of the prototype. Section 11.6 concludes this chapter.

11.1 Partitioning

This section describes how the CBGC implemented in Jikes RVM comes up with partitions (Section 11.1.1), and how it represents the components of the partitioning, namely the partition dag (Section 11.1.2) and the partition map (Section 11.1.3).

11.1.1 Coming up with partitions

The online version of Andersen's analysis from Chapter 10 comes up with the partitioning for the prototype CBGC in Jikes RVM. As Section 9.5.1 shows, an analysis based on types is too imprecise to yield a partitioning for good CBGC performance. The analysis from Section 10 is the first pointer analysis for Java that is based on allocation sites instead of types, and may therefore be precise enough for CBGC. Of course, CBGC is a client of the online analysis that triggers propagations at garbage collection time, since that is when it needs conservative analysis results.

Since propagation can happen at garbage collection time, and propagation usually allocates objects, CBGC in Jikes RVM must allow for allocation at garbage collection time. Usually, this is not possible, because the collection is triggered exactly because no memory is available to perform allocation. To work around this

difficulty, objects allocated on behalf of the analysis reside in a different area of the heap that is not "full" at the collection. This segregation of regular objects and analysis objects into different heap areas also has the useful side-effect of facilitating separate measurements of the heap usage of the two kinds of objects.

Whereas the analysis as described in Chapter 10 uses an extra header word in each object to keep track of its h-node id for validation, CBGC in Jikes RVM does not use this extra header word. This makes it possible to build a boot image of Jikes RVM where no objects in the boot image or allocated at runtime have the extra header word. Of course, the configuration with the extra header word is still available, and it is useful for regression tests that validate the pointer analysis in case new bugs crop up.

11.1.2 Partition dag

Conceptually, the partition dag is a directed acyclic graph where partitions are nodes, and edges represent may-points-to information of objects in the partitions. CBGC in Jikes RVM uses a straight-forward adjacency list representation [42, Section 23.1] based on Java objects: it maintains an array of partition nodes, each of which is an object. Each partition has a list of partitions that are targets of its outgoing edges.

When the online pointer analysis discovers new points-to relations that lead to cycles in the partition dag, those are collapsed into a single partition. In the representation, partitions reside in a fast union-find data structure [42, Chapter 22]. In other words, partitions are referenced via handles. When two partitions are merged, one of them is chosen as the representative. An access of either handle finds the representative, caches it, and returns it.

11.1.3 Partition map

The partition map maps objects to partitions. At allocation time, it determines into which partition the new object goes. At garbage collection time, it tells the garbage collector which partition the object belongs to; among other things, this determines whether the collector even considers the object in the partial reachability traversal.

At allocation time, the partition map is based on the h-node id of the allocation site. The pointer analysis in the compiler picks the h-node id and hardcodes it into the allocation sequence. When the allocation sequence executes, it maps the h-node id to a partition, and allocates the new object into that partition.

At garbage collection time, the partition map works differently for objects in the boot image than for objects allocated at run time. Hence, as a first step, it determines whether the object belongs to the boot image based on its address.

For objects in the boot image, the partition map then proceeds to look up the type from the object header. As discussed in Section 10.3.6.1, the online pointer analysis is flexible in which h-node ids it assigns to objects in the data snapshot of the boot image. To enable the partition map, it just uses the same h-node id for all objects of a given type in the boot image. Thus, at runtime, the type maps to a

unique h-node, which in turn yields the partition.

For objects allocated at runtime, the partition map first determines the block containing the address of the object. At runtime, blocks are dedicated to partitions, and an array maps block ids to the partition they belong to. The partition map uses this as the second step for mapping an object to its partition.

In all cases, the partition map employs the fast union-find data structure of partitions from Section 11.1.2 to return the correct partition of an object even after its original partition may have merged with others.

11.2 Estimator

CBGC in Jikes RVM can use one of two estimators: the combined estimator or the oracle estimator. Per default, it uses the combined estimator. But for validation or as a point of reference, a build can opt to compile the oracle estimator into the boot image instead, so it is used at every runtime garbage collection.

The combined estimator yields the best results among the realistic estimator compared by Section 9. In CBGC in Jikes RVM, it works exactly as discussed earlier, except that it takes into account that some objects are immortal. Immortal objects include objects in the boot image, as well as certain objects allocated at runtime, such as TIBs[p.188]. Of course, the estimator "estimates" that all immortal objects survive and none die.

In contrast to Section 9, where the oracle estimator was based on perfect Merlin death times computed from a trace [72], the oracle estimator in Jikes RVM does not have the luxury of a trace or any other device for predicting the future. Instead, it obtains complete knowledge of the present (which objects are dead and which are live) by performing two full reachability traversals. The first traversal marks and counts all reachable objects, whereas the second traversal unmarks all reachable objects again.

This two-traversal mechanism of the oracle estimator works because both traversals reach exactly the same set of objects. Before the first traversal, all objects are unmarked. At the end of the first traversal, all reachable objects are marked, whereas all unreachable objects are still unmarked. The second traversal views marked objects as white, and thus, unmarks objects when it reaches them. By definition, it never reaches an unreachable object, and thus, never gets confused that unmarked objects can be either unreachable (not reached by the first traversal) or reachable (reached by the first traversal, but also already reached by the second traversal). At the end of the second traversal, all objects are unmarked again, which means that everything looks exactly like before the oracle estimator started.

While the oracle estimator takes a long time and is not useful for realistic CBGC, where the goal is to avoid the cost of full traversals, it is useful for validation. In a build with the oracle estimator, the reclamation phase of the garbage collector asserts that it reclaims exactly as many objects as the oracle estimator predicted. If that is not the case, either the estimator or the garbage collector have bugs. Since the estimator code base is stable, it helps quickly detecting bugs introduced by changes to the code base of the rest of the garbage collector. In addition to this GC validation, the implementation of CBGC in Jikes RVM also piggy-backs the validation of the pointer analysis onto the oracle estimator.

A limitation of the oracle estimator implementation is that while it perfectly predicts how many bytes are live and dead in each partition, that does not necessarily translate directly into reclaimed memory due to fragmentation. For example, if 1,000 bytes of a 4,096-byte block are dead, but 12 bytes survive, the block does not become available for allocation of objects belonging to different partitions than the partition that owns the block.

In Jikes RVM, both the combined estimator and the oracle estimator must carefully avoid any allocation. If an estimator were to allocate memory, it would change the amount of live memory, which is exactly what it is trying to predict in the first place. Of course, for the same reason an estimator must also avoid making any objects unreachable.

11.3 Chooser

The prototype of CBGC in Jikes RVM uses the greedy chooser, just as described in Section 7.2. As the results in Section 9 show, the greedy chooser comes up with near-optimal choices more quickly than the flow-based chooser.

CBGC in Jikes RVM segregates objects into regular objects and analysis objects. Analysis objects are allocated at garbage collection time, from a different area of the heap than regular objects. The chooser uses different lower bounds for how much memory must at least be reclaimed among regular and among analysis objects. That way, if the estimates are accurate, the chooser makes sure that after the collection, enough memory is available for the regular allocation that triggered the allocation in the first place.

In Jikes RVM, the chooser must be careful to avoid allocating any objects or making any objects unreachable. This is because the chooser relies on the accuracy of the estimates; accurate estimates will become inaccurate if the amount of live objects changes during chooser execution.

11.4 Partial garbage collection

Partial garbage collection is the second most difficult component of the prototype CBGC in Jikes RVM (the most difficult component is the online pointer analysis). It can be described as mark-sweep CBGC only to the same extent that that term applies to the JMTk mark-sweep collector; in reality, it manages many objects with different tricolor abstractions. The exact implementation details of partial garbage collection for CBGC in Jikes RVM are what shows that CBGC works in a real system. The material of this section is what the simulator of Section 9 abstracts from: challenges that need to be solved for running CBGC in a virtual machine.

Partial GC for CBGC in Jikes RVM poses the following challenges:

- Boot image objects are not segregated by partition, posing two challenges: the partition map can not use the object address like for runtime-allocated objects, and it is not easy to sweep all objects belonging to a given partition. Section 11.1.3 described how the partition map deals with the first challenge by using the type of boot image objects; Section 11.4.4.1 discusses how it solves the second challenge with per-partition black object data structures.
- The propagator allocates objects at garbage collection time, which is problematic because garbage collections are triggered precisely because there is no memory available to allocate objects into. As mentioned earlier, CBGC in Jikes RVM solves this challenge by segregating regular from analysis objects; Section 11.4.1 describes how that works in the implementation.
- In Jikes RVM, the fields of scalars reside at lower memory addresses than their headers, whereas the elements of arrays reside at higher addresses than their headers (see Figure 2.1). However, unless small objects are segregated by size, a sweep needs to find the header of an object from its address. Section 11.4.1 describes how to solve this challenge by segregating small scalars from small arrays, and using the highest address of the one and the lowest address of the other during sweeps.
- In Java, pointer analysis must run online to deal with dynamic class loading, and new pointer analysis results may lead to merging of partitions. When that happens, a fast union-find data structure allows retrieving the representative partition, as discussed in Section 11.1.2. However, the challenge remains that the state of mark bits in objects of merged partitions may be inconsistent; Section 11.4.3 discusses how to avoid that by making sure mark bits remain consistent in the presence of partial garbage collection.
- The abstract CBGC algorithm from Figure 4.2 relies on the operation $pickGray(p)$ to pick a gray object in a given partition p, but JMTk provides only one global shared work queue to keep track of all gray objects in all partitions. The solution to this challenge is to maintain per-partition gray object sets, as discussed in Section 11.4.4.
- Finalizers may resurrect objects, which creates the challenge that a garbage collector may not be able to reclaim an object even though it is unreachable. Section 11.4.5 describes how to change the abstract CBGC algorithm from Figure 4.2 to make it correct even in the presence of finalizers.

The rest of this section describes partial garbage collection for CBGC in Jikes RVM in the following steps. Section 11.4.1 discusses how the prototype CBGC organizes its heap. Section 11.4.2 explains when and how it triggers garbage collections. Section 11.4.3 introduces global data structures that the collector relies on. Section 11.4.4 provides details on how it maintains the tricolor abstraction in data structures local to spaces of partitions. Section 11.4.5 describes how to change the abstract CBGC algorithm from Figure 4.2 to make it correct in the presence of finalizers.

11.4.1 Heap organization

Figure 11.1 shows how the prototype CBGC in Jikes RVM organizes the heap into areas, partitions, and spaces. Areas are large contiguous ranges of virtual memory addresses. Partitions are the unit of garbage collection of a CBGC, and may span different areas; the memory belonging to a partition is, in general, not contiguous. Each space belongs to exactly one (area, partition) combination, and contains objects segregated into blocks by how the collector deals with them.

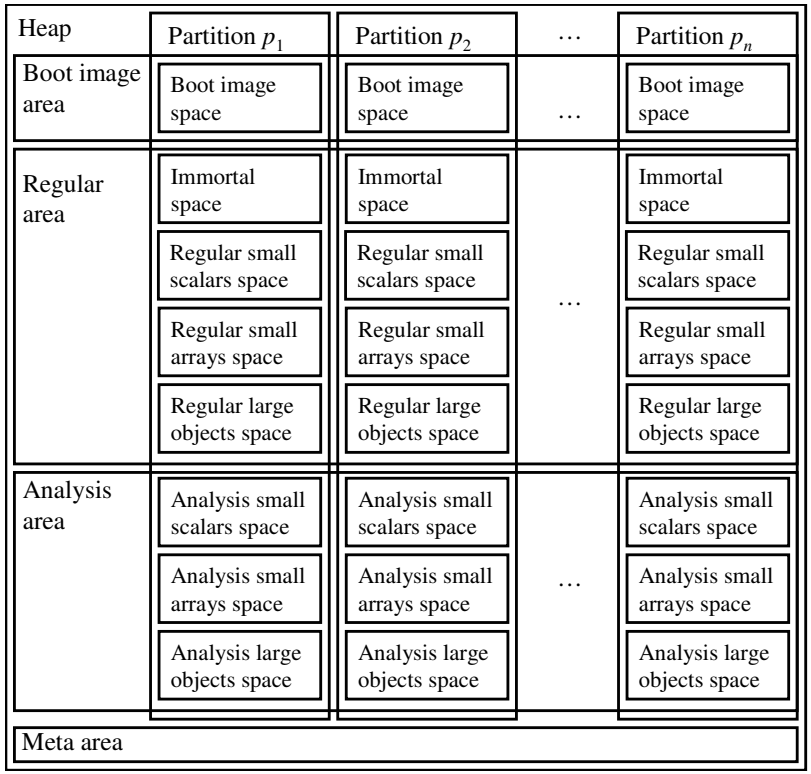


Figure 11.1: Heap organization into areas, partitions, and spaces.

There are four areas, or contiguous ranges of virtual memory addresses: the boot area, the regular area, the analysis area, and the meta area. The boot area contains the boot image. The regular area contains normal objects allocated by the mutator, whereas the analysis area contains objects allocated by the pointer analysis, which may propagate constraints at garbage collection time. The meta area contains data structure for maintaining raw addresses to gray or black objects during garbage collection. In a garbage collector requiring a write barrier, the meta area would also contain remembered sets, but in CBGC, the meta area is empty between garbage collections. The meta area is unique in that the data it contains is managed with

explicit memory management, and does not belong to any of the garbage collected partitions or spaces.

Each object resides in a block, or in the case of a large object, in multiple consecutive blocks. A block is part of exactly one space. For example, a small object might occupy the memory addresses from $124 \times 4096 + 12$ to $124 \times 4096 + 28$, and thus reside in the 4096-bytes block number 124. That block might belong to a regular small scalar space. Only in the boot image are blocks shared among the boot image spaces of different partitions; but even for the boot image, each object is part of exactly one space of one partition. A space belongs to exactly one (area, partition) combination. In the above example, the regular small scalar space containing block 124 would belong to the regular area and to some partition p_i.

The prototype CBGC in Jikes RVM uses three kinds of spaces: immortal object spaces, mark-sweep spaces, and treadmill spaces. Each space kind manages different kinds of objects with a different tricolor abstraction. Each of the 8 spaces of a partition shown in Figure 11.1 is an instance of one of these three space kinds.

Immortal spaces manage boot image objects, which Jikes RVM treats as immortal, and runtime objects such as TIBs, which also never die. Immortal objects must not die, because the Jikes RVM runtime system may keep using them even if no Java-level pointer refers to them anymore. Also, immortal objects must not move (change their address); for example, the stack of the garbage collector thread is immortal, and if it could move, that would make the stack variables that the collector relies on inconsistent. An immortal space of objects allocated at runtime consists of a set of not necessarily consecutive blocks that are dedicated only for objects of that space.

Mark-sweep spaces manage small mortal objects allocated at runtime. Other CBGC algorithms could also use copying for these objects. A mark-sweep space consists of a set of not necessarily consecutive blocks that are dedicated only for objects of that space.

Treadmill spaces manage large mortal objects allocated at runtime. It is not advisable to move those objects, since for large objects without outgoing pointers, a copy would require work proportional to the object size, whereas a non-copying collector only needs to expend a constant amount of work on it. A treadmill space consists of a set of superblocks, each of which is a sequence of consecutive blocks occupied by a large object spanning all of them. Blocks for different objects are not necessarily consecutive.

As shown in Figure 11.1, each partition has 8 spaces. Both the boot image space and the immortal space belong to the immortal space kind. Each of the regular and analysis small array and scalar spaces is a mark-sweep space. Finally, both the regular and the analysis large object space are treadmill spaces.

The prototype of CBGC in Jikes RVM manages scalars and arrays with different spaces, because they have different layouts. Figure 2.1 shows that the fields of scalars reside at lower memory addresses than their headers, whereas the elements of arrays reside at higher addresses than their headers. This means that the collector can sweep arrays in ascending address order only, and can sweep arrays in descending address order only. On the one hand, when sweeping in ascending address order,

after processing an object, the collector knows the next higher base address of an object, and for an array, that is sufficient to find the header, whereas for a scalar, there may be an unknown number of fields before the header. On the other hand, when sweeping in descending address order, after processing an object, the collector knows the next lower end address of an object, and for a scalar, that is sufficient to find the header, whereas for an array, there may be an unknown number of elements behind the header.

JMTk's mark-sweep collector solves the problem of sweeping scalars or arrays by segregating them by size, and using a parallel array of mark-bits and in-use bits. Since the bits are stored in parallel, the sweep does not need to find the object header to manipulate them; and since the objects are segregated by size, the sweep does not need to find the object header to determine the size. CBGC in Jikes RVM chooses not to segregate objects by size, because they are already segregated by partition and by space, and thus an addition level of segregation might lead to too much indirection, too much fragmentation, and too much meta-data. Instead, it segregates small objects by whether they are scalar or arrays, and then sweeps them in descending or ascending address order. This has the additional benefit of being a step towards allowing a Cheney scan in a copying collector; JMTk does not support Cheney scans, and uses an explicit data structure for the set of gray objects even in copying garbage collection.

11.4.2 Garbage collection triggering

The prototype CBGC in Jikes RVM triggers a garbage collection when the regular area is full, i.e., when the sum of blocks in use in the regular area has reached the specified heap size. The analysis area, on the other hand, is dimensioned large enough so that CBGC never needs to trigger a garbage collection on behalf of the analysis. This is necessary because the analysis may perform constraint propagation at the start of a garbage collection, and that must not trigger a recursive garbage collection by filling up the analysis space. In the prototype, the analysis area spans much more virtual memory than any of the benchmarks ever needed. Pages from that memory are only mapped on demand.

Sometimes, the garbage collector does not reclaim enough memory in the regular area to satisfy the pending allocation request. This may happen for example because the estimator's guess was off, or because of fragmentation. When the collector does not reclaim enough memory, it tries again, and during the retry, the estimator knows that all partitions chosen in previous attempts will have 100% survivors in the current attempt. To limit the number of retries, the chooser exponentially increases its lower bound of how much memory needs to be freed at least by each new attempt. If a fixed number of retries, for example four, still did not reclaim enough memory, CBGC in Jikes RVM falls back to doing a full garbage collection.

In addition to the implicit triggering of garbage collection because memory in the regular area is exhausted, it is also possible to explicitly trigger a full collection. For example, the experiments described at the end of this chapter invoke the benchmark's *main*() method twice, explicitly triggering a full GC between the two

runs. That way, the second run is largely unperturbed by the analysis, since no new constraints are generated, and the analysis area contains the live objects of a fixed set of constraints that is not mutated anymore.

11.4.3 Global data structures

While CBGC maintains most of its state on the level of partitions and spaces, it also uses some global data structures. They include the partition dag and partition map, the mark state and mark bits in the object headers, and some per-area data.

Section 11.1.2 describes the data structures implementing the partition dag, and Section 11.1.3 describes the data structures implementing the partition map.

CBGC in Jikes RVM uses one global mark state to support merging of partitions. The mark bit of all objects in all partitions always means the same thing: it is 0 between garbage collections, and 1 during garbage collection for gray or black objects. That means that when partitions are merged after propagation at the beginning of a garbage collection, the mark bits of all objects in the merged partitions are 0. Maintaining the global mark state includes sweeping all survivor objects after garbage collection and resetting their mark bits to 0.

CBGC in Jikes RVM uses a two-word object header. Only one bit out of those two words is dedicated to GC, namely the mark bit. The rest of the object header stores the pointer to the TIB, and bits to support locking and hashing [12]. In other words, CBGC does not require any extra header word; both the online pointer analysis and the partition map work without relying on additional information in the object header.

A global array maps block ids to integers identifying which of the 8 spaces of its partition each block belongs to (see Figure 11.1), or to a constant for the meta area. All areas except the boot image area also have a free list of blocks, supporting explicitly acquiring or releasing blocks for individual spaces. Finally, each area keeps track of the number of blocks in use in that area. The number of blocks in use in the regular area triggers garbage collection, as described in Section 11.4.2.

11.4.4 Concrete per-space CBGC operations

This section describes how CBGC in Jikes RVM implements the tricolor operations of the abstract CBGC algorithm from Figure 4.2. Regardless of which space an object belongs to, the operation that checks whether $color(o) = \textbf{white}$ uses the mark bit of the object: object o is white if and only if it is not marked. The remaining operations work differently for objects in different spaces, and are discussed per space kind below. This means that the abstract algorithm in Figure 4.2 delegates each action to the appropriate space of the appropriate partition.

The format of the following discussion of per-space tricolor operations follows Table A.1, which shows the tricolor operations for the canonical full garbage collection algorithms.

11.4.4.1 Space kind of immortal spaces

The boot image space and the immortal space of each partition belong to this space kind. For allocation, each immortal space includes a pointer to the current block into which it allocates objects, along with an allocation bump-pointer to the next free address at which to allocate an object. For tracing, each boot image space and each immortal space includes a mark buffer storing addresses of gray and black objects, and two indices in the mark buffer: scan and top. All objects from the bottom to scan are black, and all objects from scan to top are gray.

Table 11.1 shows how spaces containing immortal objects implement the operations of the algorithm in Figure 4.2. Since boot image objects of different partitions may be mixed in the same block, the algorithm can not sweep all boot image objects belonging to a given space in address order. Instead, action $color(o) \leftarrow \textbf{black}$ keeps black objects in the mark buffer for future sweep with the action $color(o) \leftarrow \textbf{white}$. When the garbage collector is done processing the current partition, the mark buffer can be discarded.

Table 11.1: Concrete connectivity-based stop-the-world garbage collector operations for the space kind of immortal spaces. The numbers in parentheses in the middle column refer to the lines in the algorithm in Figure 4.2 that use the operations in the left column.

$alloc(p, size)$		increase bump pointer, and return old value (for boot image, this happens globally, whereas at runtime, it happens per space)
$merge(p_1, p_2)$		nothing to do
$color(o) \leftarrow \textbf{gray}$	(5,13)	mark and push on mark buffer
$pickGray(p)$	(8)	at scan index in mark buffer
$color(o) \leftarrow \textbf{black}$	(9)	increment scan index of mark buffer
reclaim memory	(15)	nothing to do: by definition, immortals never die
$color(o) \leftarrow \textbf{white}$	(16)	unmark during sweep of mark buffer

11.4.4.2 Space kind of mark-sweep spaces

The space kind of mark-sweep spaces encompasses the regular and analysis small scalar and array spaces of each partition. Each space contains a set of doubly-linked free-lists for different size classes. While objects are not segregated by size, free objects are chained into lists by size to allow quick retrieval of a given size object for allocation. In addition, a mark-sweep space steals an in-use bit from the object header to keep track of which objects are in-use and which are free (whereas the mark-bit of an in-use object keeps track of whether it is white or not). For tracing, each mark-sweep space includes a mark-stack storing addresses of gray objects, and a singly-linked list of all its blocks.

Table 11.2 shows how mark-sweep spaces implement the operations of the al-

gorithm in Figure 4.2. The sweep for performing the memory reclamation and unmarking operations processes objects of one block at a time; as discussed earlier, within a block, the sweep works either in ascending or in descending address order, depending on whether the objects are arrays or scalars. Most of the operations are almost the same as in the canonical concrete full mark-sweep collector from Table A.1, except that the mark stack is local to the given space, instead of global for the whole heap.

Table 11.2: Concrete connectivity-based stop-the-world garbage collector operations for the space kind of mark-sweep spaces. The numbers in parentheses in the middle column refer to the lines in the algorithm in Figure 4.2 that use the operations in the left column.

alloc(p, *size*)		take off free list for *size* if the free list is empty, try splitting a larger free object, or acquire a block
merge(p_1, p_2)		concatenate block lists and free lists
color(o) ← **gray**	(5,13)	mark and push on mark stack
pickGray(p)	(8)	top of mark stack
color(o) ← **black**	(9)	pop o from mark stack
reclaim memory	(15)	put white objects on free lists, release empty blocks
color(o) ← **white**	(16)	during sweep, unmark survivors

11.4.4.3 Space kind of treadmill spaces

The space kind of treadmill spaces encompasses the regular and analysis large object spaces of each partition. Each space maintains three doubly linked lists, one each for white, gray, and black objects. Large objects can span multiple, consecutive blocks. The first three words of the first block of an object are used as extra header words to maintain the previous and next pointer of its doubly-linked list, and to store the object reference itself, to allow accessing the header regardless of whether the object is an array or a scalar. The extra 3 header words per object incur only a moderate per-object overhead: in the current implementation, the smallest large objects have size 4,096 bytes, so 12 bytes of header use up less than 0.3% of the object size.

Table 11.3 shows how treadmill spaces implement the operations of the algorithm in Figure 4.2. Most of the operations are almost the same as in the canonical concrete full-heap treadmill collector from Table A.1, except that the three lists for white, gray, and black objects are local to the given space, instead of global for the whole heap.

11.4.5 Finalizers

A finalizer[p.186] is a *finalize*() method of an object, which the runtime system automatically invokes when the object has become unreachable, but before its memory

Table 11.3: Concrete connectivity-based stop-the-world garbage collector operations for the space kind of treadmill spaces. The numbers in parentheses in the middle column refer to the lines in the algorithm in Figure 4.2 that use the operations in the left column.

alloc(p, *size*)		acquire consecutive blocks large enough for *size*, and put the new object on the white list
merge(p_1, p_2)		concatenate the white lists (the gray and black lists are empty)
color(o) ← **gray**	(5,13)	mark and move from white to gray list
pickGray(p)	(8)	at end of gray list
color(o) ← **black**	(9)	move from gray to black list
reclaim memory	(15)	release all blocks of the white list
color(o) ← **white**	(16)	flip meaning of white and black lists

is reclaimed. To finalize an object means invoking its *finalize*() method, and the act of doing that for all objects that require it is called finalization.

Finalizers pose a challenge to garbage collectors, because they may resurrect objects. When invoking *finalize*(), the runtime system passes a pointer of the object in the implicit variable **this**. The *finalize*() method can then proceed to store **this** into a global or into the field of another object, thus making the finalized object reachable again, along with all objects it points to, directly or transitively. The Java language specification states that should the object become unreachable again in the future, the runtime system does not invoke its finalizer again, so each object can be resurrected at most once [60, Section 12.6].

Garbage collectors can treat objects with finalizers by preemptively making them gray. Whenever an object becomes unreachable, the garbage collector checks whether it has a non-empty finalizer (if *finalize*() has an empty method body, it can not possibly resurrect any objects). If so, it continues its reachability traversal from this object as if it were itself still reachable, and thus, reaches all objects that the finalizer could resurrect. Furthermore, it schedules the object for finalization. Finalization happens at any time before the next collection. It might indeed resurrect the object, in which case it is reachable at the next collection. If it does not resurrect the object, the next collection reclaims it.

When JMTk allocates an object, it checks whether it has a non-empty finalizer, and if yes, adds it to a global data structure *finalizerSet*. Then, at collection time, it scans all objects in *finalizerSet*, and for each objects that is still white, removes it from the set and makes it black. Unfortunately, in CBGC, this strategy would prevent early reclamation.

Instead, CBGC in Jikes RVM maintains a separate *finalizerSet*(p) for each partition p. When it allocates an object, it puts it into the *finalizerSet*(p) of its partition. When two partitions p_1 and p_2 are merged due to new results of the online pointer analysis, the representative keeps the union *finalizerSet*(p_1) ∪ *finalizerSet*(p_2).

Figure 11.2 shows how CBGC in Jikes RVM treats finalizers at garbage collection time. At the end of the reachability traversal for a given partition (Line A), it makes all white objects of that partition that have finalizers gray (Lines B-F). Then it repeats the reachability traversal starting from these gray objects (Line G). This algorithm ensures that none of the objects reachable from the object with the finalizer are reclaimed at the current collection: they become gray and eventually black either in the second reachability traversal, or the reachability traversal of one of the successor partitions. If one of the successor partitions is not chosen at the current collection, its objects will not be reclaimed anyway.

```
 1: estimate how many objects are dead and live in each partition
 2: choose a set C of partitions
                        Partial garbage collection
 3: for each root v
 4:     if v ≠ null and partition(target(v)) ∈ C
 5:         color(target(v)) ← gray
 6: for each chosen partition p ∈ C in topological order
 A:     process gray objects in p (see Lines 7-13 of Figure 4.2)
 B:     for each object o ∈ finalizerSet(p)
 C:         if color(o) = white
 D:             remove o from finalizerSet(p)
 E:             schedule o for finalization
 F:             color(o) ← gray
 G:     process gray objects in p (see Lines 7-13 of Figure 4.2)
14:     for each heap object o with partition(o) = p
15:         if color(o) = white, reclaim its memory
16:         else color(o) ← white
```

Figure 11.2: Abstract connectivity-based stop-the-world garbage collector in the presence of finalizers.

11.5 Results

This section investigates the performance of the prototype CBGC implementation in Jikes RVM, and gives optimization recommendations for improving the performance. The prototype uses the Andersen partitioning, but to put those numbers into perspective, this section also experiments with other partitionings implemented for the system in Jikes RVM. Section 11.5.1 describes the experimental methodology. Section 11.5.2 investigates the performance of a CBGC with just one partition, which is equivalent to a full-heap collector[p.189]. Sections 11.5.3 and 11.5.4 look at the performance of CBGC in Jikes RVM with partitioning based on Harris's and Andersen's analysis, respectively. Section 11.5.5 concludes with optimization recommendations derived from the experimental results.

11.5.1 Methodology

This section uses a 2.4GHz Intel Pentium 4 with 2GB of memory running Linux (kernel version 2.4) for its experiments. It uses Jikes RVM 2.2.1 with the "FastAdaptive" compilers configuration, which uses the optimizing compiler for the boot image, and the adaptive system at runtime. Like Section 10.6, this section disables inlining for boot image methods to keep the work load for the pointer analysis manageable. In addition, it runs Jikes RVM with deterministic yield-points to facilitate profiling.

All numbers in this section measure the second invocation of the application's *main*() method. After the first invocation, the virtual machine disables the adaptive recompilation system and performs a full-heap garbage collection. This way, there is no compilation activity during the second invocation of *main*(), and thus, the numbers in this chapter are less perturbed by background activity of the virtual machine.

This section experiments with the three example benchmarks jess, db, and pseudojbb; Section 2.2 describes the benchmark suite. The prototype also works for many other programs, but this section presents detailed results focusing on just three representative programs. The harness of the two SPECjvm98 benchmarks jess and db explicitly triggers one garbage collection at the beginning and one at the end of the run, and the current prototype CBGC treats such explicitly triggered collections as full-heap collections that collect all active partitions. For this section, jess uses a heap size of 40MB, db uses a heap size of 30MB, and pseudojbb uses a heap size of 100MB. A recent paper with in-depth results on these three benchmarks [19] characterizes jess as a program with high GC load and generational behavior, db as a program with lower GC load, and pseudojbb as a program with high GC load, but also high nursery survival rate when run with generational collectors.

All experiments with CBGC in this chapter use the combined estimator, the greedy chooser, and the mostly mark-sweep partial collector described earlier. The only component that varies is the partitioning: in addition to the partitioning based on Andersen's analysis, this section looks at a trivial partitioning with just one partition containing all objects, and at the Harris partitioning. The data structures supporting the Harris or Andersen analysis reside in the analysis heap, which is separate from the regular heap, and thus does not count towards the heap size. Of course, CBGC may still need to traverse analysis objects in its reachability traversal to find out which regular objects are garbage.

11.5.2 Full-heap collections

The individual operations of the reachability traversal and reclamation phases of a CBGC are more complex than those of a full-heap collector. For example, CBGC needs to look up the partition and the space of each pointer it traverses, and based on the result, decide how to treat the object. To compare the cost of the basic GC operations of CBGC and other collectors, this section looks at full-heap collectors: that way, any performance differences stem from GC operations, not from partial or opportunistic collection.

This section compares two collectors: "JMTk/MS" is the mark-sweep full heap collector that comes with Jikes RVM; and "CBGC/trivial" is a connectivity-based collector with the trivial partitioning. CBGC/trivial runs the general combined estimator and greedy chooser before every GC, even though the outcome is obvious. Also, CBGC/trivial uses the general partition map and the general mostly mark-sweep partial garbage collector, even though it is clear which partition each object maps to.

Table 11.4 shows the arithmetic mean of garbage collection pause times for JMTk/MS and CBGC/trivial. Since both collectors collect the full heap every time, the pause times hardly vary from collection to collection. The table shows that CBGC/trivial incurs much longer pauses than JMTk/MS, even though both collect the same amount of memory (the full heap).

Table 11.4: Average full-heap collection pauses, in seconds.

Benchmark	JMTk/MS	CBGC/trivial
jess	0.54	1.75
db	0.60	1.73
pseudojbb	1.04	3.24

To explain why the CBGC/trivial collections take longer than the JMTk/MS collections, Table 11.5 shows how long each phase of the CBGC/trivial garbage collection takes. This table uses the second-to-last garbage collection of each benchmark as an example; the time break-down in other collections is similar. The numbers show that the cost of the actual collection dominates. Both the reachability traversal and the reclamation phase are expensive. For pseudojbb, the reclamation phase is actually more expensive than the reachability traversal, probably because the survivor rate is high [19]. The costs of the estimator (around 16ms) and root scan (19ms) do not vary much from benchmark to benchmark. Both phases scan the roots, but the root scan for the reachability traversal is slightly more expensive, since it makes target objects gray. With the trivial partitioning, the chooser costs less than 0.3ms, since the lone partition is the only choice.

Table 11.5: Full-heap garbage collection pause breakdown, in seconds. The numbers in parentheses refer to lines in the algorithm in Figure 4.2.

Benchmark	Estimator (1)	Chooser (2)	Root scan (3-5)	Rest of reachability traversal (7-13)	Reclamation phase (14-16)
jess	0.016,400	0.000,296	0.021,251	1.097,837	0.637,922
db	0.014,999	0.000,121	0.018,144	0.971,414	0.608,978
pseudojbb	0.015,179	0.000,118	0.018,367	1.463,552	1.924,101

The experiments with full-heap collectors show that each individual operation of the reachability traversal and the reclamation phase of garbage collection is more expensive with the current version of the prototype CBGC than with JMTk/MS.

11.5.3 Partitioning based on Harris's analysis

Chapter 9 used a simulator to experiment with CBGCHCG, which uses the Harris partitioning, the combined estimator, and the greedy chooser. This section looks at a very similar collector in the prototype implementation in Jikes RVM. It uses the same partitioning, estimator, and chooser. The most important aspect in which it differs is that it uses mostly mark-sweep GC, whereas CBGCHCG in Chapter 9 uses copying GC. Another difference is that the Harris analysis runs online in Jikes RVM, and allocates some auxiliary data structure. While the objects representing those data structures incur additional work for the garbage collector, that overhead is much lower than for Andersen's more sophisticated pointer analysis. Hence, this section demonstrates how truly partial CBGC in Jikes RVM works with less perturbation by analysis data structures.

Table 11.6 shows the arithmetic mean of garbage collection pause times for CBGC/Harris, and also repeats the numbers for JMTk/MS and for CBGC/trivial from Table 11.4 for comparison. CBGC/Harris incurs about the same cost as CBGC/trivial for jess, a higher cost for db, and a lower cost for pseudojbb.

Table 11.6: Average collection pauses with Harris partitioning, in seconds.

Benchmark	JMTk/MS	CBGC/trivial	CBGC/Harris
jess	0.54	1.75	1.76
db	0.60	1.73	1.98
pseudojbb	1.04	3.24	2.63

Table 11.7 shows how long each phase of the CBGC/Harris garbage collection takes. This table uses the second-to-last garbage collection of each benchmark as an example; the time break-down in other collections is similar. The estimator cost is usually around 34ms, up from 16ms for CBGC/trivial. The chooser cost is usually around 19ms, instead of less than 0.3ms as with CBGC/trivial. Without the increased costs of the estimator and chooser, CBGC/Harris would outperform CBGC/trivial for both jess and pseudojbb, and perform similarly for db.

Table 11.7: Garbage collection pause breakdown with Harris partitioning, in seconds. The numbers in parentheses refer to lines in the algorithm in Figure 4.2.

Benchmark	Estimator (1)	Chooser (2)	Root scan (3-5)	Rest of partial GC (6-16)
jess	0.037,670	0.035,106	0.016,121	1.626,075
db	0.032,324	0.010,839	0.019,815	1.967,683
pseudojbb	0.032,797	0.011,926	0.019,743	2.536,604

To explain why the estimator and chooser take longer with CBGC/Harris than with CBGC/trivial, Table 11.8 investigates how many partitions there are in the Harris partitioning on average at a garbage collection. An active partition is a partition that contains at least one objects at the beginning of a garbage collection,

and must therefore be considered by the estimator and the chooser. There are on the order of 120 to 220 active partitions. For jess, which has more partitions, the chooser is more expensive than for db or pseudojbb. This indicates that the chooser takes noticeably more time for larger partition dags. To a smaller degree, that is also true for the estimator: it is slower for jess than for the benchmarks with fewer partitions. This indicates that the estimator scales better than the chooser when the size of the partition dag increases.

Table 11.8: Average size of Harris partitioning, in number of partitions.

Benchmark	Active	Chosen
jess	219.7	163.8
db	122.5	84.5
pseudojbb	128.4	73.2

In the simulator experiments from Chapter 9, CbgcHCG improved more clearly upon Semi than Cbgc/Harris improves over Cbgc/trivial. As mentioned earlier, one reason for that is that the estimator and the chooser cost time. Another reason is that the partitioning costs space. More specifically, the Harris analysis has auxiliary data structures that the garbage collector may need to traverse. The size of the boot image of Cbgc/Harris is 38.3MB, whereas the size of the boot image of Cbgc/trivial is only 34.8MB.

The additional 3.5MB of boot image are occupied by three data structures for the Harris analysis. For each type, the Harris analysis maintains a set of its supertypes, a set of its subtypes, and a set of fields whose contents are declared to be of that type. The Harris analysis uses these data structures to find all may-point-to edges from and to previously resolved types when processing a newly resolved type. In the current implementation of the Harris analysis in Jikes RVM, those sets are not optimized for space; one could probably change their representation so they use far less than 3.5MB of the boot image.

Since Cbgc performs partial collections, it could theoretically ignore the additional analysis data structures, if they reside in partitions that it can avoid collecting. Unfortunately, this is not the case. The additional data structures do increase the work of the traversal and reclamation phase. This becomes clear from an inspection of the partition dag. Cbgc/Harris has one big "pivot" partition that is a predecessor of most of the remaining partitions.

Table 11.9 characterizes the topology of the Harris partitioning at the second-to-last garbage collection of a benchmark run with verbose output. The pivot is a predecessor of all but a hand full of other partitions. Any garbage collection that tries to collect a partition behind the pivot has to collect the pivot as well.

When the pivot partition is small, Cbgc can perform well even if it has to choose the pivot at most collections. Table 11.10 shows the size of the pivot before and after the second-to-last garbage collection, and shows the size of the boot image space of the pivot partition. The pivot includes around 7.5MB of the boot image. A majority of the objects that survive each collection in the pivot belong to the boot image. The size of the boot image space of the pivot varies slightly between

Table 11.9: Number of partitions in given position relative to pivot partition, for Harris. Column "Behind" gives the number of active partitions that are immediate or transitive successors of the pivot; Column "Total" gives the total number of active partitions at that garbage collection.

Benchmark	Pivot	Behind	Total
jess	1	210	218
db	1	113	121
pseudojbb	1	120	128

benchmarks due to partition merging when Harris finds new may-point-to edges.

Table 11.10: Size of pivot partition in megabytes, for Harris. Column "Before" gives the total number of bytes in dead or live objects before the collection, Column "After" gives the total number of bytes in surviving objects after the collection, and Column "Boot" gives the total number of bytes in the boot image space of the pivot.

Benchmark	Before	After	Boot
jess	22.0	9.2	7.4
db	23.4	9.2	7.5
pseudojbb	30.6	12.0	7.4

To conclude, this section experimented with Cbgc/Harris (Cbgc in Jikes RVM using the Harris partitioning), and found that it performs slightly better than Cbgc/trivial (which uses just one partition). The reason why it does not perform significantly better is that the garbage collector needs to traverse the data structures of the Harris analysis itself, and that the chooser becomes more expensive due to the increased number of partitions. The performance improvements due to opportunistic partial collections are clearly not enough to overcome the overheads of the current Cbgc prototype implementation compared to a full-heap JMTk collector.

11.5.4 Partitioning based on Andersen's analysis

Because the simulator results from Chapter 9 showed that type-based partitionings are too weak, this chapter focuses on a Cbgc with a partitioning based on Andersen's analysis, a non-trivial pointer analysis that works on the granularity of allocation sites. This section explores the performance with that partitioning. In the experiments for this section, the benchmark jess crashed due to race conditions, hence this section presents numbers for db and pseudojbb only.

Table 11.11 shows the pause times of collections with the Andersen partitioning. The distribution turns out to form three clusters. There are many fast collections, some collections of medium duration, and a few slow collections. The fast collections of Cbgc/Andersen are in fact faster than garbage collections of Cbgc/Harris. However, there are too many collections, and each individual fast or medium collection reclaims too little memory to support enough allocation requests from the mutator, so the collector soon has to run again.

Table 11.11: Garbage collection pauses with Andersen partitioning. There are three clusters of collections "Fast", "Medium", and "Slow". For each cluster, "#" shows how many such collections there are, "Range" shows the range of their pause times in seconds, and "Avg." shows the average pause times of collections of that kind.

Benchmark	Fast			Medium			Slow		
	#	Range	Avg.	#	Range	Avg.	#	Range	Avg.
db	56	0.88-1.12	0.97	2	2.33-2.58	2.46	5	9.92- 9.94	9.50
pseudojbb	12	1.55-1.82	1.68	13	3.25-3.70	3.41	3	10.34-11.25	10.91

Table 11.12 shows how long each phase of the CBGC/Andersen partitioning takes. This table uses the second-to-last garbage collection of db and pseudojbb as an example; in both cases, that collection belongs to the fast cluster. The estimator cost increased further from Harris, to 239ms. The chooser cost increased even worse than the estimator cost, to 958ms. The rest of the partial GC, on the other hand, became extraordinarily fast: these collections spend hardly any time on their reachability traversal and reclamation phases.

Table 11.12: Garbage collection pause breakdown with Andersen partitioning, in seconds. The numbers in parentheses refer to lines in the algorithm in Figure 4.2.

Benchmark	Estimator (1)	Chooser (2)	Root scan (3-5)	Rest of partial GC (6-16)
db	0.216,623	0.658,441	0.019,450	0.020,927
pseudojbb	0.260,567	1.258,146	0.019,657	0.002,110

To see why the chooser and estimator became so much slower, Table 11.13 investigates how many partitions there are in the Andersen partitioning on average at a garbage collection. A comparison with Table 11.8 shows that the Andersen partitioning leads to significantly more partitions than the Harris partitioning. Also, a comparison with Table 11.12 shows that the number of partitions and the performance of the chooser are clearly correlated: the chooser is much slower for pseudojbb than it is for db, since pseudojbb has more partitions.

Table 11.13: Average size of Andersen partitioning, in number of partitions.

Benchmark	Active	Chosen
db	749.9	86.0
pseudojbb	976.1	133.7

Another difference to CBGC/Harris is that CBGC/Andersen has to contend with much bigger analysis data structures. Where the boot image size was 34.8MB with the trivial partitioning and 38.3MB with the Harris partitioning, the data structures for Andersen's partitioning almost double the boot image size to 76.5MB. Section 10.6.2.1 investigates the space cost of the online Andersen analysis in more detail.

Just like for the Harris partitioning, the Andersen partitioning also contains a "pivot" partition that dominates large parts of the partition dag. The fast and medium collections from Table 11.11 are those collections that do not choose the pivot partition. However, those collections do not reclaim enough, so the collector must occasionally fall back to a GC that chooses the pivot. While that garbage collection is not a full GC, it is slower than a GC in CBGC/trivial due to the large analysis data structures in the heap.

To conclude, this section found that the chooser does not scale well enough, and that the analysis data structures for Andersen are too large. Andersen's analysis itself is not precise enough to reliably allow CBGC to ignore the analysis data structures in partial collections. On the bright side, on average, the reachability traversal and reclamation phases with the Andersen partitioning are much faster than with the Harris partitioning.

11.5.5 Optimization recommendations

The previous sections investigated the performance of the prototype CBGC in Jikes RVM, and found that for various reasons, it falls short of the performance of JMTk/MS, a well-optimized, full-heap garbage collector in Jikes RVM.

This is a discrepancy to the simulator results from Chapter 9, where a simplistic CBGC outperformed a full-heap collector. One of the reasons for this discrepancy is that individual operations of CBGC's reachability traversal and reclamation phase are more costly than in JMTk/MS, whereas the simulator assumes uniform cost for all operations. But even when taking these costs into account, a partial CBGC gains less over a full-heap collector than the simulator predicted. One reason for this is that the greedy chooser takes more time when there are many partitions, whereas the simulator assumed that the chooser's cost in time was insignificant. Another reason is that the program analyses for the partitioning take a significant amount of heap space, whereas the simulator assumed an ahead-of-time static analysis where those objects are not around anymore at runtime.

To improve the performance of the prototype CBGC in Jikes RVM, one needs to attack the problem on multiple fronts simultaneously. One needs to:

- Reduce the constant cost of individual CBGC operations. Two concrete steps in that direction would be to treat the boot image more uniformly with the other heap areas (speeding up the reachability traversal), and to use copying GC for small mortal objects (speeding up the reclamation phase).

- Optimize the greedy chooser for the case with many partitions. One way to do that might be to pick an even simpler chooser algorithm; it might be that that yields choices of similar quality, just like the greedy chooser yields choices of similar quality to the optimal flow-based chooser.

- Improve the precision of the pointer analysis. When the pointer analysis is more precise, its points-to sets are smaller, which saves analysis space, reducing the load on the garbage collector. Ideally, a more precise pointer analysis

would also allow a partitioning where objects for the pointer analysis reside in partitions that the garbage collector can ignore when collecting regular objects.

11.6 Conclusions

The prototype CBGC in Jikes RVM described in this chapter works, it is fully implemented and successfully runs a large suite of benchmarks and regression tests. All benchmarks from Table 2.2 for which Column "Used in" includes Chapter 11 execute successfully and produce correct outputs; this is also the case for a variety of regression tests targeting key virtual machine components that are not listed in Table 2.2. The fact that the prototype CBGC works proves that connectivity-based garbage collection can be implemented in a Java virtual machine, and can deal with all real-world challenges that that poses.

Unfortunately, at the time of the writing of this chapter, the prototype CBGC implementation is too slow to compete with other Jikes RVM garbage collectors in terms of performance. A lot of this inefficiency is due to the fact that the implementation is not tuned for performance. With more time spent on performance tuning its performance will most likely improve, and may even become better than the performance of the most efficient competitor garbage collectors in Jikes RVM. Section 11.5 investigates the performance of CBGC in Jikes RVM.

11.6.1 Future work

The first item of future work is to improve the performance of the prototype CBGC implementation. When the performance reaches acceptable levels, the next item of future work is to do more detailed experiments comparing the performance of CBGC and of other collectors in Jikes RVM. Those experiments need to measure metrics for throughput, space efficiency, and responsiveness. More specifically, they would measure throughput at a given heap size by time in seconds it takes to execute each benchmark, and measure space efficiency by performing the throughput experiments for a range of heap sizes. The experiments would measure responsiveness by tracing the start time and end time of each garbage collection, and then computing MMU[p.182] for a range of time intervals. Since the prototype CBGC relies on the online pointer analysis from Chapter 10, which itself takes a lot of time and space resources, the performance experiments for CBGC would separate out the time and space taken by the analysis, and report numbers both with and with out it. More specifically, running the benchmark's *main*() method twice and measuring the performance in the second run factors out the time cost of pointer analysis, whereas the segregation of regular and analysis objects described above factors out the space cost of pointer analysis.

11.6.1.1 Space kind for copying spaces

Another item of short-term future work is to add an additional space kind for copying spaces, and to use that for small mortal runtime objects. The current CBGC in Jikes RVM uses mark-sweep for small mortals, but in fact, CBGC can use any tricolor abstraction for its spaces; in fact, the experiments in Section 9 used copying for small mortals. Just like the space kind for mark-sweep spaces from Section 11.4.4.2, the space kind for copying spaces would segregate arrays from scalars due to their different object layout (see Section 11.4.1).

Each copying space contains two singly-linked lists of blocks for the two semispaces from-space and to-space. To support allocation, it maintains an allocation pointer in from-space; and to support copying during garbage collection, it maintains an allocation pointer in to-space. To support garbage collection, a copying space uses forwarding pointers in the original copies of objects left in from-space after copying them to to-space; and a scan-pointer on to-space to enable Cheney scan[p.184], as well as clearing mark-bits of survivors to maintain the global mark-state.

Table 11.14 shows how copying spaces implement the operations of the algorithm in Figure 4.2. Most of the operations are almost the same as in the canonical concrete full copying collector from Table A.1, except that the semispaces are local to the given space, instead of global for the whole heap.

Table 11.14: Concrete connectivity-based stop-the-world garbage collector operations for the space kind of copying spaces. The numbers in parentheses in the middle column refer to the lines in the algorithm in Figure 4.2 that use the operations in the left column.

$alloc(p, size)$		increase bump pointer, and return old value
$merge(p_1, p_2)$		concatenate the from-space lists of blocks (the to-space list of blocks is empty)
$color(o) \leftarrow$ **gray**	(5,13)	copy to to-space and forward
$pickGray(p)$	(8)	at scan pointer in to-space
$color(o) \leftarrow$ **black**	(9)	move scan pointer past o
reclaim memory	(15)	release all blocks of the from-space block list
$color(o) \leftarrow$ **white**	(16)	flip meaning of from-space and to-space

11.6.1.2 Longer-term future work

A short-term future work item was to improve the performance of the garbage collector implementation. Equally important is the future work of improving the performance of the online pointer-analysis. Right now, it is the first non-trivial pointer analysis that works for Java; performance improvements could turn it into the first efficient non-trivial pointer analysis that works for all of Java.

Depending on the shape of the partition dag, connectivity-based garbage collection may not ever need copy reserve for all from-semispaces of all copying spaces. An item of future work would be to modify the chooser algorithm and the partial collector to exploit this property, by reserving only as much memory as necessary to ensure that garbage collections never fail because of insufficient copy reserve.

In the context of a virtual machine, a lot of information is available that could help the estimator come up with better estimates. Future work in this area would use techniques from low-overhead profiling and dynamic adaptive optimization to make the estimator more accurate.

An important way in which garbage collectors affect application throughput is by changing mutator locality. CBGC probably already improves mutator locality by allocating connected objects together. It would be interesting to study whether this is indeed the case, to quantify it, and to make CBGC locality-aware so it can actively augment this effect.

The prototype CBGC in Jikes RVM described in this chapter runs only on uniprocessor machines. But there is no fundamental reason why CBGC should not work also on multi-processor machines. Making any garbage collector parallel is, however, a major engineering effort. In the case of CBGC, it might pay off because the partition dag makes some independence guarantees for partitions that a parallel collector could exploit to avoid object-level synchronization in many cases.

11.6.2 Discussion

This chapter describes how to implement CBGC in a virtual machine. It is based on an actual working implementation that uses a given design, and demonstrates that the design and implementation are correct. The collector is completely realistic, it uses no oracles and works in a real-world system. However, it does not perform well yet; improving the performance is subject for future work.

Chapter 12

Conclusions

This dissertation introduces a new family of garbage collection algorithms (CBGC) that are based on object connectivity properties. CBGC segregates objects into partitions based on their connectivity, and also uses the connectivity information to decide which partitions to collect at each collection. CBGC is motivated by experiments that demonstrate that there is a strong correlation between connectivity and the lifetime and deathtime characteristics of programs. This dissertation describes a number of collectors from the CBGC family and evaluates them using a simulator. Furthermore, it describes the first non-trivial pointer analysis for all of Java, and an implementation of CBGC in a Java virtual machine that uses that pointer analysis.

Appendix A

Definitions

This chapter defines terminology for use in the rest of this dissertation. The reader may skip it and refer to it on demand when chapters use terms that the reader is unfamiliar with, or when they use familiar terms in unfamiliar ways. When a chapter refers to a definition in this appendix, it uses the notation $\text{term}^{\text{p.}n}$, where n is the number of the page with the definition for the term.

Garbage collection literature uses some terminology differently than is common in other parts of computer science, such as "heap" or "object". Furthermore, even among different garbage collection papers, words such as "block" or "incremental" do not always mean the same thing. This chapter defines terminology to avoid confusion in the remainder of the dissertation. It also gives some background to avoid cluttering the subsequent discussion with material that may already be familiar to the reader.

A.1 Data structures

stack:

1. LIFO (last-in-first-out) buffer.
2. The LIFO buffer of activation records of routines for one thread of execution. Among other things, a stack stores local variables that may point to heap objects.

heap:

1. All of the memory of a process holding dynamically allocated objects. This is the default meaning of "heap" in this dissertation.

   This definition follows: [86, Section 1.7].
2. Binary tree where each node carries a key, and the nodes are partially ordered such that the relation $key(parent(n)) \leq key(n)$ holds for each non-root node n.

   This definition follows: [42, Section 7.1].

dag: Directed acyclic graph.

This definition follows: [42, page 89].

SCC: Strongly connected component, a maximal set S of nodes in a directed graph such that for all pairs of members $n_1 \in S$ and $n_2 \in S$, there is a path from n_1 to n_2 and a path from n_2 to n_1. All nodes in an SCC are reachable from each other.

This definition follows: [42, Section 23.5]

WCC: Weakly connected component, a maximal set W of nodes in a directed graph such that in the undirected version of the graph, for all pairs of members $n_1 \in W$ and $n_2 \in W$, there is a path from n_1 to n_2 and a path from n_2 to n_1. All nodes in a WCC would be reachable from each other if one ignored the directions of edges.

This definition follows: [82, Section 2]

A.2 Virtual machines

VM:

1. Virtual machine, engine for executing intermediate language code on concrete hardware, providing runtime support such as garbage collection. A well-known example is the Java virtual machine.
2. Virtual memory, memory addresses as seen by running processes, which are mapped to physical addresses by the operating system.

In this dissertation, VM always stands for virtual machine, not virtual memory.

JVM: Java Virtual Machine. Engine for executing a Java program in form of bytecode on top of hardware and an operating system. A JVM includes runtime support such as a garbage collector, a thread scheduler, and JNI, and is tied closely to a set of libraries. Many JVMs include JIT compilers.

JIT: A JIT is a just-in-time compiler in a virtual machine. In Java, JIT compilers compile Java bytecode instructions into machine code for the hardware that the virtual machine is running on.

This definition follows: [95, Section 3.13].

JNI: The Java Native Interface, which allows Java code to interact with compiled code that is native to the hardware and operating system on which the Java virtual machine runs. Usually, the native code is compiled from C or C++. JNI provides facilities for Java code to call native code in the form of native methods, and for native code to call Java code in the form of a fixed set of C functions.

This definition follows: [96].

reflection: A language feature that allows data to determine code to be executed by stepping outside of the usual syntax of the language. More specifically, in Java, reflection allows manipulating methods, fields, and classes as explicit entities. The example code in Figure A.1 writes the value 123 to a field that is not known statically, but determined by reflection based on data from user input.

```
1: String fieldName = readFieldName();
2: Field f = A.class.getField(fieldName);
3: A a = new A();
4: f.setInt(a, 123);
```

Figure A.1: Reflection example.

Jikes RVM: An open-source research virtual machine for Java from IBM Research [5, 29]. All components of the runtime system, including JIT compilers, garbage collectors, and the thread system, are written in Java.

Jalapeño: Original name of Jikes RVM.

A.3 Objects and GC

object:

1. Contiguous chunk of memory allocated by one dynamic allocation on the level of the programming language. In the case of Java, an array or a scalar allocated by the keyword **new**. Usually, objects include a header. This is the default meaning of "object" in this dissertation.

   This definition follows: [86, Section 1.7] and [135, Section 1].

2. An instance of a class in object-oriented programming.

scalar: Non-array object. In Jikes RVM, all scalars are instances of classes, and their size is determined by the number of fields in the class.

This definition follows: [5].

pointer: A variable or a field containing the address of an object. Low-level languages like C also allow pointers to things other than objects. In Java terminology, pointers are called "references", and their targets can be scalars or arrays.

block: Contiguous chunk of memory. In this dissertation, for a given garbage collector, all blocks have the same size, which is a power of 2 (for example, $2^{12} = 4096$ bytes). All blocks are aligned to the block size. A block in a garbage collector does not necessarily have to be the same as a page in an operating system. A block may contain one or more small objects, or be part of a large object.

This definition follows: [78, Section 3.1], [86, Page 98].

This definition differs from: [135], where a block is the exact memory used to store one object.

garbage collection: Depending on the context, garbage collection can mean one of the following.

1. Automatic memory reclamation.
2. One cycle of executing a garbage collector. Often shortened to "collection". Each garbage collection consists of a reachability traversal and a reclamation phase, where the reachability traversal includes a root scan.
3. The research discipline of studying garbage collectors.

garbage collector: Automatic memory manager, supports dynamic allocation and automatic reclamation of heap objects. As discussed in Section 1.2, the two main kinds of collectors are reference counting and tracing. Another approach to automatic memory management without a garbage collector is regions. This dissertation focuses on tracing garbage collectors only.

GC: This acronym is overloaded and can mean either garbage collection or garbage collector, depending on the context in which it is used.

This definition follows: [18, 23].

reference counting GC: Garbage collection based on counting references to an object as they are written, as opposed to tracing GC which performs a reachability traversal. As discussed in Section 1.2, reference counting garbage collection maintains a counter with each object that counts how many references (i.e., pointers) there are to it. Writing a pointer to the object increments the counter, and deleting a pointer to the object decrements the counter. When the counter drops to zero, the object is dead and will not be used anymore, so it is freed. In addition, the reference counters of all successor objects are decremented.

This definition follows: [40].

mutator: The part of a program that is not the garbage collector. From the point of view of a tracing garbage collector, everything that is not GC just mutates the object graph, possibly making objects unreachable. For Java applications executing on top of Jikes RVM, the mutator includes the application itself, as well as all Jikes RVM runtime system components except for the GC. For example, the JIT compilers are part of the mutator.

This definition follows: [47].

A.4 Reachability

root:

1. Stack or global variable that may point to an object. The roots are the starting point for a garbage collector's reachability traversal: all objects not reachable from roots are dead. This is the default meaning of "root" in this dissertation.

   This definition follows: [86, Section 1.2].

2. In a directed graph, a node without incoming edges.

global: A statically allocated variable that can in principle be used by any method during the entire program execution. In Java, a global is a static field of a class, which is shared among all instances of the class, and can also be used from other classes if it is visible from them.

reachable: Figure A.2 illustrates the following three notions of "reachable".

1. An object o_n is reachable from an object o_1 if there exist objects $o_2, \ldots, o_{n-1}$ and pointer fields $f_1, \ldots, f_{n-1}$ such that $target(o_i.f_i) = o_{i+1}$ for $i = 1, \ldots, n-1$.
2. An object o is reachable from the root r if o is reachable from the object $target(r)$.
3. An object o is reachable if there exists a root r such that o is reachable from r.

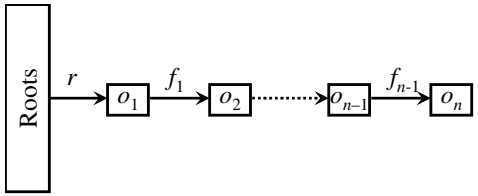


Figure A.2: Reachability example.

live:

1. A live object is an object that is reachable from the roots by following pointers.

   This definition follows: [86, Section 1.2].

2. A live variable is a variable that will be used in the future.

   This definition follows: [3, Section 10.2].

dead:

1. A dead object is an object that is not reachable from the roots by following pointers.

   This definition follows: [86, Section 1.2].

2. A dead variable is a variable that will not be used in the future.

   This definition follows: [3, Section 10.2].

garbage: One or more dead objects.

This definition follows: [86, Section 1.3].

A.5 Time

time:

1. Time in seconds is the usual, physical notion of wall-clock time.
2. Time in bytes is the total number of bytes that a program has allocated up to a given point. Every allocation of an object of size s byte advances time by s. In the experiments for this dissertation, all benchmarks are run on a uniprocessor machine; if there are multiple threads, their allocations are serialized, so the notion of time in bytes is still well-defined. Time in bytes is the traditional notion of time in memory management research. In the usual case, garbage collection only happens if an allocation request can not be satisfied without reclaiming some memory first, and hence, discrete values of time in bytes are the only possible collection points. This is true for the experiments in this dissertation.

   This definition follows: [86, Page 145].

throughput: Speed at which a program gets its work done, see Section 1.3.1.

responsiveness: Absence of disruptive periods where a program responds sluggishly or not at all, see Section 1.3.3.

age: The age of an object is the time in bytes since it was allocated.

birth time: The birth time of an object is the time in bytes when it is allocated.

death time: The death time of an object is the time in bytes at which it transitions from being live to being dead. In other words, the death time is when the object first becomes unreachable because the mutator deletes a pointer on the last path from roots to the object. For the kind of garbage collectors that this dissertation considers, the death time of an object is the earliest time at which it can be reclaimed.

This definition follows: [72, Section 6.1], which describes how to compute precise death times efficiently.

lifetime: The lifetime of an object is the duration of its life, measured in bytes from its birth time to its death time.

This definition follows: [18, 67]

immortal: An immortal object is an object that survives until close to the end of the program execution. A garbage collector should not traverse immortal objects if it can avoid it, since they do not become garbage.

This definition follows: [18].

truly immortal: An object is truly immortal if its deathtime coincides with the program end time, where time is measured in bytes.

This definition follows: [82].

quasi immortal: An object is quasi immortal if it is not truly immortal, but the time (in bytes) from its death to the end of the program execution is at most as long as its lifetime. Figure A.3 illustrates the distinction between quasi immortal and truly immortal.

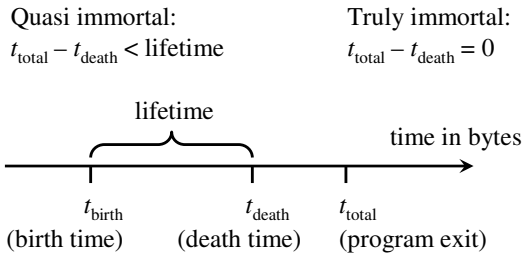


Figure A.3: Terminology for immortal objects.

This definition follows: [82].

utilization: For the time interval $[t_1, t_2]$, $utilization(t_1, t_2)$ is the fraction of $[t_1, t_2]$ spent in the mutator and not the garbage collector. In this definition, time means time in seconds. Figure A.4 illustrates utilization.

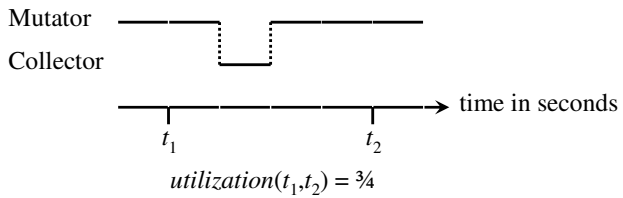


Figure A.4: Utilization example.

This definition follows: [33].

MMU: Minimum mutator utilization in all time intervals of a given length Δt. In this definition, time means time in seconds. MMU is a measure of responsiveness for non-concurrent collectors, where higher values of MMU indicate better responsiveness. If the program starts running at $t = 0$ and runs for a total time of t_{total},

$$\text{MMU}(\Delta t) = \min\{utilization(t, t + \Delta t) \mid 0 \le t \textbf{ and } t + \Delta t \le t_{\text{total}}\}$$

Figure A.5 shows an MMU plot. For an example interval of length $\Delta t = d_1$ on the x axis, the minimum mutator utilization is the y-value MMU(d_1), indicating that all intervals of length d_1 anywhere in the program execution have a utilization of at least MMU(d_1). The intercept with the x-axis is the maximum pause time, because for smaller intervals, the interval fits into a pause where the mutator does not run at all.

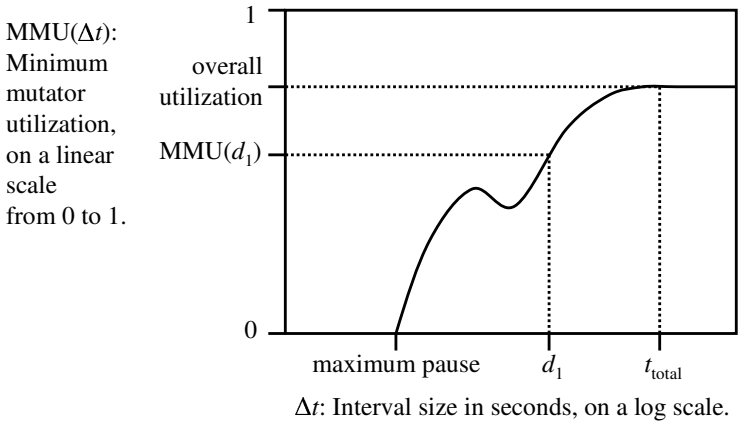


Figure A.5: Example MMU plot.

An easy way to determine the MMU experimentally is to trace all start and end times of garbage collections (time in seconds), and then compute the minimum of all $utilization(t_{\text{gcStart}}, t_{\text{gcStart}} + \Delta t)$ values for all garbage collection start times t_{gcStart}.

This definition follows: [33].

This definition differs from: [21], who alter it to make it monotonous. For them, MMU(Δt) is the minimum mutator utilization in all time intervals of *at least* the given length Δt.

A.6 Accuracy

type accuracy: Ability of a garbage collector to distinguish pointers from non-pointers. A collector that is not fully type accurate is conservative. A strong type system prevents pointer stores into non-pointer variables. In languages without a strong type system, on the other hand, the collector has to assume

that all non-pointers whose values look like pointers are pointers. Even in languages with a strong type system, the compiler may lose type accuracy for simplicity, making the collector type inaccurate for some or all pointers.

This definition follows: [86, Section 9.1].

liveness accuracy: Ability of a garbage collector to distinguish live variables from dead variables. A collector can obtain imperfect liveness accuracy from a compiler analysis, but perfect liveness accuracy is undecidable. A collector with liveness accuracy can ignore dead pointers during its reachability traversal, and thus potentially identify more objects as garbage, making it more effective.

This definition follows: [1], [80].

accuracy: Type accuracy and liveness accuracy.

This definition follows: [77].

conservative GC: A garbage collector that is not fully type accurate, and may interpret some non-pointer values as pointers during its reachability traversal.

This definition follows: [24], [86, Section 9.1].

A.7 Tracing GC

tricolor abstraction: A reasoning device for reachability traversals that comes from basic graph traversal algorithms such as depth-first search and breadth-first search, and is useful for proving correctness and termination of tracing garbage collectors.

The tricolor abstraction assigns each object that participates in a collection one of three colors white, gray, and black. Figure A.6 shows the algorithm for full tracing garbage collection using the tricolor abstraction. All objects start out white, encoding that they have not been reached yet. When the reachability traversal encounters a white object, it colors it gray, encoding that it has been reached. When the reachability traversal has traversed all outgoing pointers of a gray object, it colors it black, encoding that it has been reached and all of its successors have also been reached.

The algorithm in Figure A.6 maintains the invariant that no black objects point to white objects. Since it is a stop-the-world GC, no pointers change throughout the algorithm. It is easy to see that at the end of the loop in Lines 3-9, all live objects have been colored black, and that Lines 10-12 reclaim all dead objects.

This definition follows: [42, Sections 23.2, 23.3], [47, Section 4], and [86, Section 6.1].

white: Color of an object that has not been reached yet.

gray: Color of an object that has been reached, but may have white successors.

	Reachability traversal
Invariant:	{All objects are white.}
1:	**for each** root v
2:	**if** $v \neq$ **null**, $color(target(v)) \leftarrow$ **gray**
3:	**while** there are gray objects
4:	$o \leftarrow pickGray()$
5:	$color(o) \leftarrow$ **black**
6:	**for each** field f of o
7:	**if** $o.f \neq$ **null**
8:	**if** $color(target(o.f)) =$ **white**
9:	$color(target(o.f)) \leftarrow$ **gray**
Invariant:	{All live objects are black, all dead objects are white.}
	Reclamation
10:	**for each** heap object o
11:	**if** $color(o) =$ **white**, reclaim its memory
12:	**else** $color(o) \leftarrow$ **white**
Invariant:	{All objects are live and white.}

Figure A.6: Abstract full stop-the-world tracing garbage collector.

black: Color of an object that has been reached, along with all its successors.

tracing GC: A garbage collector that performs a reachability traversal to find which objects are live, as opposed to a reference-counting garbage collector. Figure A.6 shows the abstract algorithm for a tracing collector using the tricolor abstraction.

The canonical concrete algorithms for tracing collectors include copying, mark-sweep, and the treadmill (see Table A.1). More sophisticated collectors, such as aged-based or connectivity-based GCs, are usually extensions of tracing GC.

copying GC: Garbage collection where the reachability traversal copies live objects into a target memory area, and the reclamation phase reclaims all memory of the originating memory area. The originating area is called from-space, and the target area is called to-space. Since in a full collector, from-space and to-space occupy half of the heap each, they are called semispaces. After an object has been copied, the original memory location in from-space contains a forwarding pointer to the copy. Copying collectors usually maintain gray objects with a Cheney scan. Table A.1 shows how a copying full stop-the-world collector using Cheney scan keeps track of the tricolor abstraction, and implements the actions of the abstract full tracing collector from Figure A.6.

semispace: From-space or to-space in a copying collector.

Cheney scan: Algorithm for performing copying collections that maintains gray objects by keeping a scan-pointer in to-space, rather than requiring a separate mark-stack [31]. Figure A.7 illustrates the Cheney scan. When the collector

Table A.1: Concrete full stop-the-world tracing garbage collectors. The numbers in parentheses after actions in the left column refer to the lines in the algorithm in Figure A.6 that use these actions.

	Copying	Mark-Sweep	Treadmill
white	in from-space and not forwarded	not marked	not marked and on white list
gray	in to-space before scan pointer	marked and on mark stack	marked and on gray list
black	in to-space after scan pointer	marked and not on mark stack	marked and on black list
color(o) ← **gray** (2, 9)	copy to to-space and forward	mark and push on mark stack	mark and move from white to gray list
pickGray() (4)	at scan pointer in to-space	top of mark stack	at end of gray list
color(o) ← **black** (5)	move scan pointer past o	pop o from mark stack	move from gray to black list
color(o) ← **white** (12)	flip meaning of from- and to-space	unmark	flip meaning of white and black list

copies a live object from from-space to to-space, it places it at the allocation pointer, and advances the allocation pointer, making the object gray. When the collector scans a gray object to reach all its successors, it advances the scan pointer, making the object black.

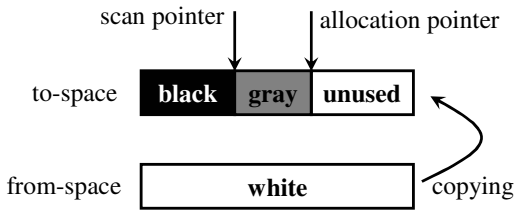


Figure A.7: Cheney scan.

mark-sweep GC: Garbage collection where the reachability traversal marks live objects by setting a mark-bit, and the reclamation phase sweeps all objects, reclaiming the unmarked ones. The sweep takes time proportional to the total number of objects. Mark-sweep collectors usually maintain gray objects with a mark stack. Table A.1 shows how a mark-sweep full stop-the-world collector using a mark stack keeps track of the tricolor abstraction, and implements the actions of the abstract full tracing collector from Figure A.6.

mark/cons ratio: Metric for characterizing garbage collector throughput; the lower the mark/cons ratio, the better the throughput. In the ratio $\frac{mark}{cons}$, "mark" stems from the action that colors white objects gray in mark-sweep garbage collection, and "cons" stems from the Lisp function for dynamic allocation. Thus, "mark" measures GC cost: most collectors perform work proportional to how much their reachability traversal encounters; and "cons" measures GC benefit: the purpose of a collection is to reclaim memory to satisfy future allocation (cons) requests. If $\frac{mark}{cons}$ is low, the collector spends little cost to achieve high benefit, indicating good throughput.

As a side note, one could claim that the throughput of a mark-sweep collector should be measured as $\frac{mark+cons}{cons}$, to account for the cost of the sweep phase. However, the sweep phase is often performed lazily during subsequent allocations; and, comparisons between collectors yield the same result with either metric, since

$$\left(\frac{mark_1}{cons_1} < \frac{mark_2}{cons_2}\right) \textbf{ if and only if } \left(\frac{mark_1+cons_1}{cons_1} < \frac{mark_2+cons_2}{cons_2}\right)$$

treadmill GC: Garbage collection where the reachability traversal puts live objects on a doubly-linked list of survivors, and the reclamation phase reclaims all objects remaining on the original doubly-linked list of objects [15]. Treadmill collectors usually maintain gray objects by keeping them on a separate list from black objects. That makes for three lists (for white, gray, and black objects), commonly maintained by extra header words with pointers to the previous and next object. Table A.1 shows how a treadmill full stop-the-world collector using a gray list keeps track of the tricolor abstraction, and implements the actions of the abstract full tracing collector from Figure A.6.

finalizer: Method invoked on an instance of a class after the garbage collector has determined that it is unreachable, but before its memory is reclaimed. The code of the finalizer may resurrect the object and any objects reachable from it, complicating the garbage collector.

This definition follows: [60, Section 12.6].

reachability traversal: Lines 1-9 of the algorithm in Figure A.6. The first part of the reachability traversal is the root scan in Lines 1-2. The reachability traversal colors all objects reached from roots gray, and transitively colors white successors of gray objects gray, and gray objects black, until all live objects are black, and all remaining white objects are dead.

root scan: Lines 1-2 of the algorithm in Figure A.6, the initial part of the reachability traversal that colors all targets of root pointers gray. Since the roots include pointers stored on thread stacks, the root scan needs stack walking support to be able to identify those pointers.

reclamation: Lines 10-12 of the algorithm in Figure A.6. The reclamation phase frees up the memory of dead white objects and makes it available for future allocation requests.

early reclamation: Early reclamation happens when the reclamation phase starts before the reachability traversal is finished. Harris describes early reclamation in the context of a full-heap incremental treadmill collector [65]. Connectivity-based garbage collection also supports early reclamation, which reduces the footprint of a copying partial collector [78].

A.8 Space

space efficiency: Smallness of the memory that a program runs in, see Section 1.3.2.

heap size: Amount of memory available for the heap. In this dissertation, heap size is a constant determined upon program start. The garbage collector has to satisfy all allocation requests of the mutator without ever exceeding the heap size; this may require setting aside a part of the heap for copy reserve.

Traditionally, static variables and stacks are not part of the heap. In Jikes RVM, on the other hand, they are also represented as Java objects residing on the heap.

footprint: Complete memory usage at a moment in time, including live and dead objects, headers, fragmentation, and auxiliary data structures such as remembered sets or mark stacks, but not vacant copy reserve.

time × space product: Product of the time in bytes for which memory is occupied with the number of bytes of memory that are occupied, measured in square bytes (byte^2). This is an integral under the curve of space usage over time, see Figure A.8.

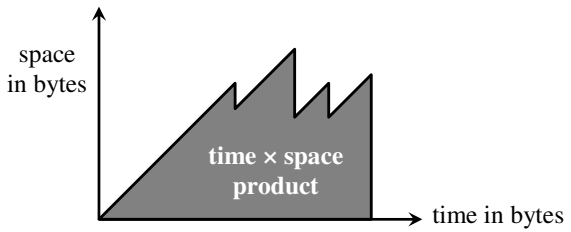


Figure A.8: Time × space product.

copy reserve: In a copying collector, the copy reserve is the amount of memory set aside to copy live objects into. In the absence of fragmentation, a pure copying collector requires one byte of copy reserve for each allocated byte in the heap, because in the worst case, all objects are live and are temporarily stored in both semispaces.

This definition follows: [21, Section 3.1].

space rental: The space required by an object over time, measured as a time × space product. In a non-moving collector, the space rental of an object with lifetime

ℓ byte and size s byte is $\ell \times s$ byte^2. In a copying collector, it is usually $2 \times \ell \times s$ byte^2 due to copy reserve.

Copying collectors can improve space efficiency by not providing a copy reserve for immortal objects. Whether the object is quasi immortal or truly immortal, its space rental is lower if it is never reclaimed than if it has a copy reserve so it may be reclaimed.

This definition follows: [18, Section 3.1].

fragmentation: Fragmentation is the part of the heap size that is not part of the space rental of objects, but the garbage collector is unable to use it to satisfy allocation requests either.

1. External fragmentation: There is an unused chunk of memory large enough to fit the object to be allocated, but the collector is unable to use it.
2. Internal fragmentation: All unused chunks of memory are too small to fit the object to be allocated, but their total size would be large enough if they were adjacent.

This definition follows: [135, Section 2].

header: Part of each object used by the runtime system. For example, in Java, object headers usually support determining the size, type, hash value, and lock status of the object, and also contain information for use by the garbage collector, such as a mark bit for mark-sweep collectors.

This definition follows: [12], [135, Section 3.2].

TIB: Type information block. Each object can be mapped to a TIB, usually by a pointer in the header. The TIB describes the type of the object, and allows virtual method dispatch.

This definition follows: [5].

A.9 Partial GC

partial GC:

1. A partial garbage collection is a garbage collection of only a part of the heap: the reachability traversal walks only a part of the object graph, and the reclamation phase reclaims only dead objects in that part. A garbage collection that is not partial is full.
2. A partial garbage collector is a garbage collector for which some, or even all, collections are partial. Most collectors require occasional full collections for completeness. The train collector is an exception [85], it is complete without ever requiring full collections.

full GC:

1. A full garbage collection is a garbage collection of the entire heap: the reachability traversal walks the entire object graph, and the reclamation phase reclaims all dead objects. A collection that is not full is partial.
2. A full garbage collector performs only full garbage collections.

complete GC: A complete garbage collector is a garbage collector that is able to reclaim all dead objects if needed. Examples for collectors that are not complete include pure reference counting GC [40], conservative GC [24], and some partial collectors [117].

This definition follows: [21, Section 2.1].

remembered set: Data structure with which a write barrier communicates additional roots to a partial collection.

write barrier: Bookkeeping mechanism required to allow partial collections in some collectors, such as generational GC. The write barrier performs additional actions upon mutator writes, and stores information into a remembered set that allows a future partial collection to treat some objects in the uncollected part of the heap as roots. For example, the generational GC write barrier records all old objects into which a write stores a pointer to a young object. Those objects serve as additional roots in a the next minor collection. Connectivity-based garbage collectors do not require write barriers, even though they perform partial collections.

nepotism: A phenomenon of partial garbage collectors that rely on write barriers, where a dead "uncle" object in the uncollected part of the heap points to a dead "nephew" object, and since the uncle is treated like a root, it keeps the nephew from being reclaimed.

This definition follows: [17, Section 3.1].

A.10 Age-based GC

age-based GC: Partial garbage collector that segregates objects by age, and uses age to decide which objects to collect. Age-based GC includes generational collectors [94, 130], older-first collectors [39, 116], and generalizations [21, 117].

generational hypothesis:

1. Weak generational hypothesis: "Newly created objects have a much lower survival rate than objects that are older."
2. Strong generational hypothesis: "Even if the objects in question are not newly created, the relatively younger objects have a lower survival rate than the relatively older objects."

This definition follows: [67].

generational GC: Younger-first age-based partial garbage collector. Divides the heap into generations $G_1, \ldots, G_n$ by age, where the youngest generation G_1 is also known as nursery. Usually, a generational collector only collects generation G_i if collecting generations $G_1, \ldots, G_{i-1}$ did not reclaim enough garbage to satisfy the next allocation request.

generation: A space used by a generational garbage collector to store objects of a certain age range.

nursery: Youngest generation in a generational GC.

mature space: Oldest generation of a generational GC.

Appel's collector: Flexible-size nursery copying generational collector with two generations (nursery and mature space), and en-masse promotion of nursery objects that survived one collection [7]. Let h be the heap size, and m be the current total size of all objects in the mature space. The mature space requires copy reserve, which leaves $h - 2m$ for the nursery, half of which is copy reserve for the nursery. That means the nursery will be collected when the total size of young objects reaches $h/2 - m$. Since m grows after each collection, the nursery size is flexible and shrinks after each collection. Occasionally, a full GC will reclaim some of the mature space memory, reducing m and thus increasing the size available for the nursery again.

major GC: A major garbage collection is a full garbage collection of a generational collector, i.e., it collects all generations.

minor GC: A minor garbage collection is a partial garbage collection of a generational collector, i.e., it collects only (some of the) younger generations. A philosophical question is whether the first collection of Appel's GC should be considered major or minor.

A.11 On-the-fly GC

on-the-fly GC: An on-the-fly collector allows the mutator to make progress during its collection, rather than stopping the world until it completes the reachability traversal and reclamation phase. As shown in Figure A.9, on-the-fly GC includes incremental GC (on a uniprocessor machine) as well as concurrent GC (on a multiprocessor machine).

concurrent GC: A concurrent garbage collector performs its collection in a thread on one processor at the same time as the mutator executes on one or more other processors. As shown in Figure A.9, concurrent GC is on-the-fly GC on a multiprocessor machine.

parallel GC: A parallel collector performs its collection on multiple threads simultaneously, with each collector thread running on its own processor. Most

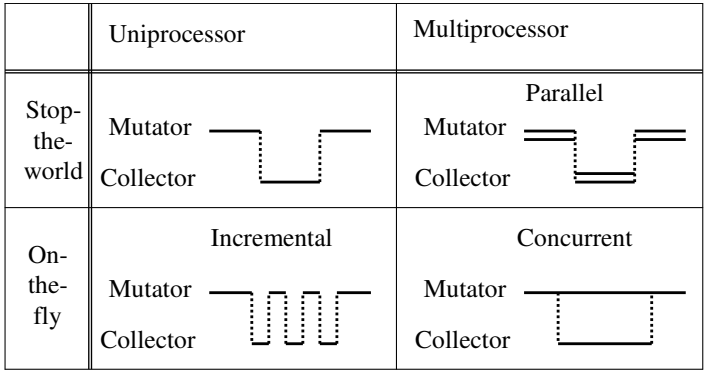


Figure A.9: Terminology for mutator/collector interleaving. Each picture shows one garbage collection cycle (including reachability traversal with root scan, and reclamation phase). Time runs from left to right, and horizontal lines are threads of execution.

garbage collectors in Jikes RVM are parallel, but stop-the-world, occupying the top right quadrant of Figure A.9.

This definition follows: [11].

This definition differs from: [22], what they call "parallel" is called "concurrent" in this dissertation, following [16].

incremental GC: An incremental garbage collector performs its collection interleaved with the mutator execution, performing only a few collector actions at a time. As shown in Figure A.9, incremental GC is on-the-fly GC on a uniprocessor machine.

This definition follows: [86, Chapter 8].

This definition differs from: [85]; what they describe as "incremental" is called "partial" in this dissertation. Table A.2 shows how, with the terminology in this dissertation, the distinction between full and partial GC is orthogonal to the distinction between stop-the-world and incremental GC.

Table A.2: Examples for combinations of full/partial and stop-the-world/incremental garbage collection.

	Full	Partial
Stop-the-world	[31]	[7]
Incremental	[15]	[22]

stop-the-world GC: A stop-the-world garbage collector sends the mutator to sleep before it starts its collection, and only wakes it up again after completing the reachability traversal and reclamation phase. As shown in Figure A.9, stop-

the-world GC may happen on a uniprocessor machine, as well as in the form of parallel GC on a multiprocessor machine.

A.12 Connectivity

CBGC: Connectivity-based garbage collection, the subject of this dissertation.

connectivity: Manner in which pointers connect objects to each other and to roots, either directly or transitively.

Bibliography

[1] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Programming Languages Design and Implementation (PLDI)*, 1998.

[2] Gagan Agrawal, Jinqian Li, and Qi Su. Evaluating a demand driven technique for call graph construction. In *International Conference on Compiler Construction (CC)*, 2002.

[3] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[4] Ravindra Ahuja, Thomad Magnanti, and James Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[6] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.

[7] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software – Practice and Experience (SPE)*, 19(2), 1989.

[8] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter Sweeney. Adaptive optimization in the Jalapeño JVM. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[9] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Programming Languages Design and Implementation (PLDI)*, 2001.

[10] Matthew Arnold and Barbara G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *European Conference for Object-Oriented Programming (ECOOP)*, 2002.

[11] Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith. A comparative evaluation of parallel garbage collector implementations. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2001.

[12] David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the Java object model. In *European Conference for Object-Oriented Programming (ECOOP)*, 2002.

[13] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *European Conference for Object-Oriented Programming (ECOOP)*, 2001.

[14] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1996.

[15] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, 1991. Also appeared in *SIGPLAN Notices*, March 1992.

[16] Katherine Barabash, Yoav Ossia, and Erez Petrank. Mostly concurrent garbage collection revisited. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[17] David A. Barrett and Benjamin G. Zorn. Garbage collection using a dynamic threatening boundary. In *Programming Languages Design and Implementation (PLDI)*, 1995.

[18] Stephen Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[19] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2004.

[20] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *International Conference on Software Engineering (ICSE)*, 2004.

[21] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: getting around garbage collection gridlock. In *Programming Languages Design and Implementation (PLDI)*, 2002.

[22] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Programming Languages Design and Implementation (PLDI)*, 1991.

[23] Hans-J. Boehm, Alan J. Demers, and Mark Weiser. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[24] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience (SPE)*, 1988.

[25] Jeff Bogda and Ambuj Singh. Can a shape analysis work at run-time? In *Java Virtual Machine Research and Technology Symposium (JVM)*, 2001.

[26] Richard Brooksby and Nicholas Barnes. The memory pool system. Unpublished paper, 2002.

[27] Michael Burke and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *Transactions on Programming Languages and Systems (TOPLAS)*, 1993.

[28] Dante Cannarozzi, Michael Plezbert, and Ron Cytron. Contaminated garbage collection. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[29] IBM T.J. Watson Research Center. Jikes Research Virtual Machine (RVM). http://www.ibm.com/developerworks/oss/jikesrvm.

[30] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Principles of Programming Languages (POPL)*, 1999.

[31] C. J. Cheney. A non-recursive list compaction algorithm. *Communications of the ACM (CACM)*, 1970.

[32] Ben-Chung Cheng and Wen-mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[33] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Programming Languages Design and Implementation (PLDI)*, 2001.

[34] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Programming Languages Design and Implementation (PLDI)*, 1998.

[35] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *International Symposium on Memory Management (ISMM)*, 1998.

[36] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[37] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Static Analysis Symposium (SAS)*, 2003.

[38] Michał Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[39] William D. Clinger and Lars Thomas Hansen. Generational garbage collection and the radioactive decay model. In *Programming Languages Design and Implementation (PLDI)*, 1997.

[40] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM (CACM)*, 1960.

[41] Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *Transactions on Programming Languages and Systems (TOPLAS)*, 1986.

[42] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT press, 1990.

[43] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[44] Jeffey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference for Object-Oriented Programming (ECOOP)*, 1995.

[45] David Detlefs and Ole Agesen. Inlining of virtual methods. In *European Conference for Object-Oriented Programming (ECOOP)*, 1999.

[46] Sylvia Dieckmann and Urs Hölzle. A study of allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference for Object-Oriented Programming (ECOOP)*, 1999.

[47] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM (CACM)*, 1978.

[48] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Using types to analyze and optimize object-oriented programs. *Transactions on Programming Languages and Systems (TOPLAS)*, 2001.

[49] Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[50] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Programming Languages Design and Implementation (PLDI)*, 1998.

[51] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Programming Languages Design and Implementation (PLDI)*, 1995.

[52] Stephen J. Fink and Feng Qian. Design, implementation, and evaluation of adaptive recompilation with on-stack replacement. In *Code Generation and Optimization (CGO)*, 2003.

[53] Robert Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In *International Symposium on Memory Management (ISMM)*, 2000.

[54] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical report, University of California at Berkeley, 1997.

[55] David Gay and Alex Aiken. Memory management with explicit regions. In *Programming Languages Design and Implementation (PLDI)*, 1998.

[56] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC)*, 2000.

[57] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: a practical interprocedural heap analysis for C. *International Journal of Parallel Programming (IJPP)*, 1996.

[58] GNU. Classpath, essential libraries for Java. `http://www.gnu.org/software/classpath/classpath.html`.

[59] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, pages 921–940, 1988.

[60] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[61] Christian Grothoff. Recycling garbage theory. Technical Report CSD TR# 04-012, Purdue University, 2004.

[62] David Paul Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1998.

[63] Samuel Z. Guyer and Kathryn S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[64] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and Rene G. Rodriguez. Fiat: A framework for interprocedural analysis and transformations. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1993.

[65] Timothy Harris. Early storage reclamation in a tracing garbage collector. *ACM SIGPLAN Notices*, 1999.

[66] Timothy L. Harris. Dynamic adaptive pre-tenuring. In *International Symposium on Memory Management (ISMM)*, 2000.

[67] Barry Hayes. Using key object opportunism to collect old objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1991.

[68] Nevin Heintze. Analysis of large code bases: the compile-link-analyze model. `http://cm.bell-labs.com/cm/cs/who/nch/cla.ps`, 1999.

[69] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Programming Languages Design and Implementation (PLDI)*, 2001.

[70] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Programming Languages Design and Implementation (PLDI)*, 2001.

[71] Laurie Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, 1990.

[72] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. Error-free garbage collection traces: how to cheat and not get caught. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2002.

[73] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[74] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis (ISSTA)*, 2000.

[75] Martin Hirzel and Trishul M. Chilimbi. Bursty tracing: a framework for low-overhead temporal profiling. In *Feedback-Directed and Dynamic Optimizations (FDDO)*, 2001.

[76] Martin Hirzel and Amer Diwan. On the type accuracy of garbage collection. In *International Symposium on Memory Management (ISMM)*, 2000.

[77] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *Transactions on Programming Languages and Systems (TOPLAS)*, 2002.

[78] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[79] Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *European Conference for Object-Oriented Programming (ECOOP)*, 2004.

[80] Martin Hirzel, Amer Diwan, and Antony Hosking. On the usefulness of liveness for garbage collection and leak detection. In *European Conference for Object-Oriented Programming (ECOOP)*, 2001.

[81] Martin Hirzel, Harold N. Gabow, and Amer Diwan. Choosing a set of partitions to collect in a connectivity-based garbage collector. Technical Report CU-CS-958-03, University of Colorado at Boulder, 2003.

[82] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management (ISMM)*, 2002.

[83] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Programming Languages Design and Implementation (PLDI)*, 1992.

[84] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Programming Languages Design and Implementation (PLDI)*, 1994.

[85] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *International Workshop on Memory Management (IWMM)*, 1992.

[86] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Son Ltd., 1996.

[87] Andy C. King. Removing GC synchronization (extended version). `http://www.acm.org/src/subpages/AndyKing/overview.html`, 2003. Winner (Graduate Division) ACM Student Research Competition.

[88] James R. Larus and Satish Chandra. Using tracing and dynamic slicing to tune compilers. Technical Report 1174, University of Wisconsin, 1993.

[89] Chris Lattner and Vikram Adve. Automatic pool allocation for disjoint data structures. In *Workshop on Memory System Performance (MSP)*, 2002.

[90] Chris Lattner and Vikram Adve. Data structure analysis: an efficient context-sensitive heap analysis. Technical Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2003.

[91] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *International Conference on Compiler Construction (CC)*, 2003.

[92] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[93] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the precision of static reference analysis using profiling. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[94] Henry Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM (CACM)*, 26(6), 1983.

[95] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[96] Sun Microsystems. Java Native Interface specification. http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html, 1997.

[97] Nick Mitchell and Gary Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *European Conference for Object-Oriented Programming (ECOOP)*, 2003.

[98] J. Eliot B. Moss. Regions determined by kind and generation. Unpublished note, 1999.

[99] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*, 2001.

[100] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Programming Languages Design and Implementation (PLDI)*, 1992.

[101] Igor Pechtchanski and Vivek Sarkar. Dynamic optimistic interprocedural analysis: a framework and an application. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[102] Feng Qian and Laurie Hendren. An adaptive, region-based allocator for Java. In *International Symposium on Memory Management (ISMM)*, 2002.

[103] Feng Qian and Laurie Hendren. Towards dynamic interprocedural analysis in JVMs. In *Java Virtual Machine Research and Technology Symposium (JVM)*, 2004.

[104] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[105] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[106] Erik Ruf. Effective synchronization removal for Java. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[107] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Principles of Programming Languages (POPL)*, 1999.

[108] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[109] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium (SAS)*, 1997.

[110] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Principles of Programming Languages (POPL)*, 2002.

[111] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[112] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2001.

[113] Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. A framework for interprocedural analysis and optimization in the presence of dynamic class loading. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[114] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, 1996.

[115] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *International Symposium on Memory Management (ISMM)*, 2000.

[116] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: evaluation in a Java virtual machine. In *Workshop on Memory System Performance (MSP)*, 2002.

[117] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[118] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. On models for object lifetime distributions. In *International Symposium on Memory Management (ISMM)*, 2000.

[119] Darko Stefanović and J. Eliot B. Moss. Characterization of object behaviour in Standard ML of New Jersey. In *LISP and Functional Programming*, 1994.

[120] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[121] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Vallée-Rai Raja, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[122] Alexandru Sălcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Principles and Practice of Parallel Programming (PPOPP)*, 2001.

[123] David Tarditi and Amer Diwan. Measuring the cost of storage management. *Lisp and symbolic computation*, 1996.

[124] The Apache Tomcat Project. Apache Tomcat. `http://jakarta.apache.org/tomcat`.

[125] The Eclipse Project. Eclipse. `http://www.eclipse.org`.

[126] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[127] Mads Tofte. A brief introduction to regions. In *International Symposium on Memory Management (ISMM)*, 1998.

[128] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, 2001.

[129] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[130] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, 1984.

[131] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *European Conference for Object-Oriented Programming (ECOOP)*, 2000.

[132] Frédéric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Programming Languages Design and Implementation (PLDI)*, 2001.

[133] John Whaley and Monica Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis Symposium (SAS)*, 2002.

[134] Paul R. Wilson. Uniprocessor garbage collection techniques. Accepted for publication in *ACM Computing Surveys (CSUR)*.

[135] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: a survey and critical review. In *International Workshop on Memory Management (IWMM)*, 1995.

[136] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in a garbage-collected system. In *Programming Languages Design and Implementation (PLDI)*, 1991.

[137] Karen Zee and Martin Rinard. Write barrier removal by static analysis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[138] Craig B. Zilles. Benchmark Health considered harmful, 2001.