Adaptive Type-Partitioned Garbage Collection

by

Christopher Henry Rohrs

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

© Christopher Henry Rohrs, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author .................................................................

Department of Electrical Engineering and Computer Science

May 18, 2000

Certified by .............................................................

Barbara Liskov

Ford Professor of Engineering

Thesis Supervisor

Accepted by ............................................................

Arthur C. Smith

Chairman, Department Committee on Graduate Students
Adaptive Type-Partitioned Garbage Collection

by

Christopher Henry Rohrs

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

A novel type-based garbage collector is presented that partitions objects by type and collects only those objects most likely to be garbage. Unlike a classic generational collector, the partition is adaptively chosen at runtime to maximize collector efficiency and minimize intergenerational pointers. By using type information, the collector and compiler can eliminate up to 90% of all write barriers. An implementation for a commercial Java\textsuperscript{TM} Virtual Machine consistently copies less data than a standard collector, and up to 2.2 times less data than an age-based generational collector. However, the cost of tracking intergenerational pointers sometimes outweighs the reduction in copying. In the best case, the modified virtual machine runs 49% faster than the original; in the worst case, it runs 23% slower.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering
Acknowledgments

I wish to thank Compaq Computer Corporation for supporting me through the MIT VI-A internship program, and Bruce Foster for initially bringing me to Compaq. I am especially grateful to Bill McKeeman for mentorship during that time, and interesting me in adaptive garbage collection. I also thank my thesis supervisors, Bob Morgan and Prof. Barbara Liskov. In addition, I am indebted to many members of the JFEX team past and present, especially Chris Gillett, Richard Critz, Sanjay Ghemawat, Keith Kimball, and Rich Phillips.

Finally, I wish to thank my family for their unwavering support.
Contents

1 Introduction 8

2 A Garbage Collection Taxonomy 10
   2.1 Definitions ............................................. 10
   2.2 Basic Techniques ...................................... 10
      2.2.1 Reference Counting ................................. 11
      2.2.2 Mark-Sweep Collection .............................. 12
      2.2.3 Copying Collection ................................. 12
   2.3 Advanced Techniques ................................... 14
   2.4 Discussion ............................................. 15

3 Generational Collection 16
   3.1 Design Considerations ................................. 17
   3.2 Write Barriers ......................................... 17
   3.3 Rethinking Generational Assumptions .................. 19

4 Type-Based Collection 21
   4.1 Motivation ............................................. 21
   4.2 Basic Scheme ......................................... 22
   4.3 Type Partitioning ...................................... 24
      4.3.1 Theoretical Efficiency ............................. 24
      4.3.2 Algorithm I .................................... 25
      4.3.3 Algorithm II .................................... 27
      4.3.4 Optimal Algorithms ............................... 29
   4.4 Discussion ............................................. 32
      4.4.1 Adaptation .................................... 33
      4.4.2 Write Barrier Elimination ........................ 33
List of Figures

2-1  Reference counting pseudocode .............................................. 11
2-2  Cheney algorithm for depth-first copying from oldspace to newspace. ...... 13
3-1  Example Java constructors .......................................................... 19
4-1  Illustration of a minor collection .................................................. 25
4-2  Algorithm I .................................................................................. 26
4-3  Relationship between class graph, runtime class graph, and object graph ...... 28
4-4  Algorithm II: main loop ................................................................. 30
4-5  Algorithm II: helper functions ....................................................... 31
5-1  Algorithm for eliminating `putfield` write barriers ............................ 44
5-2  Storing a reference into an array ..................................................... 45
A-1  Runtime class graph of jess ............................................................ 61
A-2  Runtime class graph of jack ............................................................ 61
A-3  Runtime class graph of db ............................................................. 62
A-4  Runtime class graph of mtrt .......................................................... 62
A-5  Runtime class graph of javac .......................................................... 63
A-6  Runtime class graph of anagram ....................................................... 64
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Example mortality rates in <code>.209.db</code></td>
<td>22</td>
</tr>
<tr>
<td>6.1</td>
<td>Benchmark descriptions</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>MB copied during collection</td>
<td>49</td>
</tr>
<tr>
<td>6.3</td>
<td>Static percentage of <code>putfield</code> WBs eliminated at compile-time</td>
<td>50</td>
</tr>
<tr>
<td>6.4</td>
<td>Number of cards marked and requiring scanning</td>
<td>50</td>
</tr>
<tr>
<td>6.5</td>
<td>Collection time (seconds)</td>
<td>51</td>
</tr>
<tr>
<td>6.6</td>
<td>Overall benchmark time (seconds)</td>
<td>52</td>
</tr>
<tr>
<td>6.7</td>
<td>Cycles spent per GC routine for <code>_227_mtrt</code></td>
<td>52</td>
</tr>
<tr>
<td>6.8</td>
<td>The number of cycles, D-cache misses, and TLB misses for two critical routines of <code>.209.db, when using standard and type-based collection</code></td>
<td>54</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Traditional programming languages such as C and C++ require the programmer to explicitly free dynamically allocated memory. This is the cause of numerous programming errors. In one case, the programmer forgets to free unused memory, resulting in a memory leak. Even worse, the programmer may refer to objects that have been freed. Such dangling references can cause undefined behavior (e.g., crashes) later on.

Garbage collection (GC)—also known as automatic storage reclamation—eliminates such errors and simplifies program development by automatically reclaiming unneeded memory. When a program runs out of memory, the garbage collector traverses all pointers reachable from the call stack and registers. Objects encountered during this traversal are considered live. Objects not encountered are considered garbage and are reclaimed for future allocations. Reachable objects that are no longer used by the program may be retained. Many high-level languages use garbage collection, including Java\textsuperscript{TM}, LISP, Smalltalk, and ML. In addition, GC can be used with languages that are not designed to be garbage collected, such as C and C++, provided the programmer obeys certain rules [BW88].

Unfortunately, garbage collection comes at a cost. While it is hard to generalize for a large number of languages and programs, most garbage collectors consume from a few percent to 20 percent of overall program execution time [Jon96]. Wilson estimates that a well-designed garbage collector might consume no more than 10 percent of program execution time [Wil94]. Our own measurements of a production-quality Java virtual machine show that the garbage collector usually consumes under 5 percent of program execution, but occasionally consumes as much as 50 percent of program execution on standard performance benchmarks. (See Chapter 6 for details.) Another
problem is that collection pauses may disrupt many applications. However, this thesis is not concerned with reducing worst-case pause time; rather, the goal of this thesis is to reduce the overall cost of garbage collection.

The most popular and successful technique for improving garbage collection performance is age-based generational collection. These collectors are based on the weak generational hypothesis, the commonly-held assumption that young objects die faster than old objects. Generational collectors segregate objects by age and collect only the youngest objects. By concentrating on those objects most likely to be garbage, a generational collector may outperform standard techniques. While generational techniques have proven successful for functional languages such as LISP and ML, they may not be appropriate for Java. Java encourages an imperative style of programming and the frequent update of long-lived objects. With a generational collector, this may result in excess copying. In fact, our experience shows that generational techniques often degrade the performance of a Java Virtual Machine. (See Chapter 6 for details.)

However, age is not necessarily the only indicator of object mortality. In Java programs, we have found a strong correlation between an object’s type and its expected lifetime. For any given program, instances of some classes tend to be long-lived, while instances of others tend to be short lived. We expect similar behavior in other strongly-typed languages. Based on this observation, we propose a novel garbage collector that segregates objects by type—not age—and collects only the objects most likely to be garbage. A key feature of this collector is that the type partition is chosen at runtime to adapt to the program at hand. In addition, type information is used to eliminate many write barriers. We find that the type-based collector usually copies less data than a generational collector, but overhead from write barriers and intergenerational pointers are still significant.

This thesis is organized as follows. Chapter 2 reviews standard garbage collection techniques. Chapter 3 describes classic age-based generational techniques in greater detail. Chapter 4 presents the type-based collector and discusses some of the design options. Chapter 5 describes the implementation details for a Java Virtual Machine. Chapter 6 compares the performance of the type-based collector with a non-generational and a generational collector.
Chapter 2

A Garbage Collection Taxonomy

This section presents a brief taxonomy of collection techniques. Readers familiar with garbage collection basics may skip this section. Those seeking a more detailed presentation should consult references by Wilson [Wil94] or Jones and Lins [Jon96].

2.1 Definitions

The heap is storage where allocation and deallocation follow no specific order. An object is any structure stored contiguously on the heap. Every object has a header, a few words of memory adjacent to the object’s data. The header is invisible to the program and might be used by the runtime system to store the size of the object, the type of the object, whether the object is garbage, etc.

The root set of a program at some time consists of all objects that are pointed-to by some pointer on the stack frame or some pointer in a register. An object is defined to be reachable if it is either a member of the root set or pointed-to by a reachable object. Note that the set of reachable objects forms a directed graph. An object is garbage if it is no longer needed by a program. As is common practice, we ignore liveness and restrict this definition to refer only to those objects that are no longer reachable. A garbage collector reclaims garbage for the memory allocator.

2.2 Basic Techniques

There are three commonly-used GC techniques:

1. reference counting
2. mark-sweep collection

3. copying collection

These techniques form the foundation for almost all advanced garbage collectors. The last two techniques are considered  
tracing techniques since they trace the graph of reachable objects using depth-first or breadth-first search. In their simplest form, tracing collectors run when there is insufficient memory to satisfy allocation. In contrast, reference counting is interleaved with the program’s execution.

2.2.1 Reference Counting

Reference counting is the simplest of all automatic storage reclamation techniques. The reference count for each object is equal to the number of pointers pointing to it. If the reference count of an object is zero, the object is garbage. Reference counting schemes typically maintain the reference count of each object in its header. Pseudocode for maintaining reference counts is shown in Figure 2-1.

def decrement(x):
    #Decrements the reference count of x.
    #Reclaims x’s storage (and that of children) if garbage.
    1. Decrement the reference count of x by 1.
    2. If x’s reference count is zero
       a) Recursively call decrement on all of x’s children.
       b) Reclaim the storage of x.

def store(a,b):
    #Stores a pointer from object a to object b
    Let b’ be the object (if any) that a currently points to.
    1. Call decrement(b’)
    2. Store pointer from a to b.
    3. Increment the reference count of b.

Figure 2-1: Reference counting pseudocode. This code is not thread-safe.

There are two problems with reference counting. First, it cannot reclaim cycles of garbage. Hence a mark-sweep collector is often used in conjunction with reference counting. Secondly, the cost of reference counting is proportional to the number of pointer stores. This is likely to be significant.
A deferred reference counting scheme elides reference count adjustment when storing pointers from stack or register locations. As a consequence, an object cannot be reclaimed when its reference count drops to zero. Instead, the reference counts must be periodically reconciled by scanning the stack and incrementing the counts of any pointed-to objects. Now, objects with reference counts of zero may be reclaimed. The savings in pointer stores generally outweighs the cost of reconciliation.

Reference counting is generally considered inferior to tracing techniques. However, reference counting is simple to implement and has been used in numerous language implementations, including Modula-2+, Perl, and Python [vR00]. It is also frequently used ad hoc by programmers of languages such as C++ that do not support automatic storage reclamation.

2.2.2 Mark-Sweep Collection

Mark-sweep collectors operate in two stages. First, they mark reachable objects. This usually involves setting a bit in each reachable object through depth-first search from the root set. Secondly, they “sweep” through the heap and check the mark bit of each object. Those whose mark bits are not set are added to a doubly-linked free list. Subsequently, objects are allocated by searching the free list.

The mark stage is usually implemented with a stack to perform the depth-first search. In this case, the collector must be able to handle the rare case of stack overflow. The usual treatment is to clear the stack, scan the heap for marked objects with pointers to unmarked objects, place them on the stack, and resume scanning. Alternatively, marking can be done with no extra space by using pointer reversal techniques. Generally, pointer reversal is considerably more expensive than using a stack.

A few optimizations can be done to the sweep stage. First, mark bits can be stored in large bit vectors instead of in each object. This enables the collector to “sweep” large areas of garbage in a few instructions. Also, the sweep stage is often done “lazily” by interleaving it with allocation.

2.2.3 Copying Collection

Copying collectors partition the heap into two contiguous areas, fromspace and tospace. During program execution, all live objects are found in fromspace. New objects are allocated at the end of fromspace. When fromspace is full, reachable objects are copied from fromspace to tospace. The roles of oldspace and newspace are then reversed.

The Cheney algorithm is a popular way of performing this copying [Che70]. It requires no
external stack; instead, tospace is used as a queue for breadth-first search. Two pointers \( S \) and \( F \) are maintained in tospace. Objects at \( S \) are scanned, and their children are copied to \( F \). To prevent objects from being copied more than once, a forwarding pointer is left in oldspace after an object is copied. Pseudocode is shown in Figure 2-2.

**copy_all():**

Precondition: no objects in fromspace contain forwarding pointers. Postcondition: all reachable objects in fromspace are in tospace and the graph of reachable objects is isomorphic to that before the call.

\[
S = \text{start of tospace}
\]

\[
F = S
\]

for each \( r \) in the root set:

\[
\text{r=copy(}r\text{)}
\]

while \( S < F \):

\[
\text{Let n be the size of the object at } S
\]

for each child pointer \( p \) of the object:

\[
\text{p=copy(}p\text{)}
\]

\[
S = S + n
\]

**copy(p):**

Precondition: \( p \) points to the start of an object in fromspace. Postcondition: a unique copy of the object exists in tospace and \( p \) contains a forwarding pointer to this object.

if \( p \) contains a forwarding pointer \( p' \) to newspace:

Return \( p' \)

else

\[
\text{Let n be the size of object at } p.
\]

Copy object to \( F \).

\[
F = F + n
\]

Return \( F - n \)

Figure 2-2: Cheney algorithm for depth-first copying from oldspace to newspace.

The cost of copying large objects can be prohibitively expensive. Accordingly, many collectors treat large objects specially [Ung84]. More generally, some copying collectors partition the heap and collect these partitions independently. Such collectors, including age-based generational collectors, will be discussed in Section 2.3.
2.3 Advanced Techniques

Collection pauses may disrupt programs with strong or weak real-time constraints, including interactive applications and programs that interact with a network. Incremental collectors perform a small amount of work at a time, making the pauses in a garbage collected program less noticeable. Real-time collectors are incremental collectors that provide hard guarantees on the amount of time taken for any collection. Concurrent collectors run at the same time as the program (often called the mutator). Concurrent collectors are almost always straightforward extensions of incremental collectors. Most incremental algorithms require significant synchronization overhead between the collector and the program. Classic incremental collectors are based on mark-sweep algorithms [DLM+78], but other algorithms can be used with sufficient engineering.

Parallel collectors use multiple threads to speed up the garbage collector itself. Parallel and concurrent collection are often confused in the literature, but they are orthogonal; a parallel collector may or may not be concurrent. Parallel collectors have not been widely studied in the literature. Both copying and mark-sweep collectors can be parallelized [ETY97, IT93].

Conservative collectors are designed for use with uncooperative languages such as C and C++. Such collectors scan the stack and registers conservatively, treating any value that could be a pointer as a pointer. Both copying and mark-sweep collectors can be conservative [Bar88, BW88]. Conservative copying collection is discussed in detail in Section 3.3 since our experimental collector is based upon it.

Partitioned garbage collectors divide the heap into separate regions and collect each region separately. This includes a wide range of collectors. The proposed type-based collector is a partitioned collector. Other examples:

- Peter Bishop proposed dividing a large address space into areas, and collecting each area independently of the others [Bis77]. Bishop’s system maintains inter-area links and automatically moves objects between areas. By collecting small areas, the garbage collector avoids excessive paging.

- Generational collectors segregate objects by age and usually collect the youngest objects. They are of significant interest and are discussed in detail in the next chapter.

- More recently, partitioned garbage collection has been studied in the context of object databases and persistent heaps, since tracing the entire heap usually requires too many disk
accesses. In this domain, it is generally acknowledged that object age is not a good indicator of mortality, so other partition selection schemes are used. Cook, Wolf, and Zorn examine the effects of selection policy on garbage collection performance [CWZ94].

2.4 Discussion

Numerous studies have been done on the efficiency of copying and mark-sweep collectors. In theory, copying collectors are more efficient, since their collection time is proportional to the volume of reachable objects. On the other hand, a mark-sweep collection takes time proportional to the volume of the heap. However, a copy collector is not necessarily faster, since implementation details can have a significant impact on performance.

Collectors frequently combine the techniques described above. Most collectors usually pick one basic algorithm (e.g., copying collection) and a number of advanced techniques (e.g., conservative generational collection). This thesis presents a novel technique for improving collection efficiency that could be used with most basic algorithms, or in combination with other advanced techniques. However, we will consider type-based techniques primarily with copying, non-incremental collection.
Chapter 3

Generational Collection

It has long been observed that most objects are short-lived, while a few objects tend to live for a long time. Generational collectors segregate objects by age and collect only the youngest objects. By doing so, they avoid repeatedly copying these long-lived objects. Section 3.3 gives a formal justification for generational collection.

In the simplest case, the heap is partitioned into newspace and oldspace. All objects are allocated into newspace. When newspace is full, a minor collection occurs, and live objects in newspace are copied to oldspace. When a minor collection does not free enough memory, a major collection occurs, and both newspace and oldspace are collected and copied into oldspace. Minor collections are much more common than major collections.

Generational collectors have three advantages, at least if the weak generational hypothesis holds [Bak93]:

- The amortized cost of a generational collector is less than that of a non-generational collector, since fewer objects are copied.

- Because minor collections collect a smaller region, they are presumed to have better cache locality.

- Since most collections are minor, pause times are less noticeable. Hence generational collectors are often used as a cheap replacement for incremental collectors. (See Section 2.3)
3.1 Design Considerations

Generational collectors can be implemented in a wide variety of ways, but copying techniques are most commonly used. In the above example, oldspace might be divided into fromspace and tospace, providing a place to copy survivors of major collections. On the other hand, major collections could be implemented with a mark-compact algorithm, preventing the need for such complications.

Regardless of the underlying collection technique, the following considerations must be made:

- **How many generations?** Most generational collectors have two generations, but commercial collectors have been built with up to seven generations [CWB86].

- **Are generation sizes fixed?** Generally a larger newspace results in less overall copying, since objects are given more time to die. On the other hand, a larger newspace could result in larger pause times, since minor collections are less frequent. Again, this thesis is not directly concerned with pause times.

- **When are objects promoted?** In the above example, minor collections copy all survivors of newspace to oldspace. To prevent the most recently allocated objects from being “prematurely tenured”, some collectors only copy objects into oldspace after they survive $n$ collections. The disadvantage is that additional information must be stored in each object. Also, it generally requires dividing newspace into two spaces to provide a place for survivors of a minor collection.

There has been some research on adaptive generational collectors, mainly to reduce pause times. Ungar and Jackson dynamically adjust the tenuring policy based on object demographics [UJ92]. Barret and Zorn use a “dynamic threatening boundary” to reclaim tenured garbage [BZ93].

Numerous authors have suggested special treatment of large objects. Ungar and Jackson, for example, recommend segregating large strings and bitmaps [UJ92]. Their collector benefits from this scheme because large objects are not copied repeatedly; the authors do not find that large objects have different lifetimes than smaller objects.

3.2 Write Barriers

During a minor collection, all pointers into newspace must be considered part of the root set, or reachable objects may be reclaimed. Since scanning all of oldspace for pointers to newspace is
generally too expensive, write barriers are used to record the location of every intergenerational pointer. Write barriers can be implemented with software or hardware, and a variety of data structures can be used for recording pointers.

In the case of a software write barrier, the compiler emits instructions at every pointer store to record the location of store. (In an interpreted language, the interpreter does this work.) The details of the recording are discussed below. In the case of a hardware write-barrier, virtual memory is used to write-protect oldspace. When a pointer is written in oldspace, the fault handler records the pointer store. Hardware-based write barriers have the advantage that they can be used without compiler cooperation, but they are generally slower because of the excessive cost of handling faults [Zor90]. Both software and hardware-based write barriers may check whether a trapped pointer is indeed intergenerational before storing it.

The most popular ways of recording intergenerational pointers are remembered sets and card marking. A remembered set is an array of all objects in oldspace with pointers to newsspace. The write barrier simply adds the address of the stored-into object to the end of this array. In a card marking scheme, the heap is divided into fixed-sized “cards”. Each card has a corresponding entry in a bitmap. If some object on a card has a pointer from oldspace to newsspace, the bit for that card is set. The write barrier simply right shifts the address of the stored-into object and marks the corresponding card. Remembered sets are more precise than cards, but cards are usually simpler and avoid duplicates. Almost all hardware-based write barriers use card marking, with each card corresponding to one virtual memory page.

For a software write-barrier, the compiler need not emit write barriers for a pointer store if the target object is known to be older than the source object. For example, no write barrier is needed for the LISP expression (cons A B), since it cannot create intergenerational pointers. This eliminates the need for write barriers at most pointer stores. However, the situation is complicated in languages that do not separate object allocation and initialization. In Java, for example, arguments to a constructor are not necessarily older than the object being constructed. Figure 3-1 gives an example. As a result, Java programs may incur a much higher write barrier overhead than other programs.

Write barriers can be a serious performance problem in a generational collector. There is overhead in both recording pointers during the program execution and scanning these pointers at collection time. If the reduction in copying does not outweigh the write barrier overhead, performance suffers.

18
public class A {
    Object f;

    public A(Object x) {
        f = x;
    }

    public A() {
        this(new Object());
    }
}

public class B extends A {
    public B() {
        super(new Object());
    }
}

Figure 3-1: Example Java constructors. Argument \( x \) to the first constructor of class \( A \) is not necessarily older than the object being constructed, because it could have been instantiated in the second constructor of \( A \), or in the subclass \( B \).

3.3 Rethinking Generational Assumptions

Generational collectors make two assumptions:

1. Most pointers are from young to old objects.

2. Young objects have higher mortality rates than older objects. (weak generational hypothesis)

If the first assumption is not met, the cost of tracking intergenerational pointers will be prohibitively high. In addition, young objects may be needlessly copied because of pointers from dead objects in oldspace. (This is referred to as nepotism.) If the second assumption is not met, minor collections will not be more efficient than major collections, so generational collection will be no more efficient than standard collection.

The first assumption has been proven correct for a wide variety of functional languages such as ML and LISP. However, it is not immediately obvious that it holds in imperative languages such as Java and C++, where programmers frequently create long-lived data (such as an array) and modify it during the course of the program. In fact, Stefanović found that neither old-to-young nor young-to-old pointers dominate in Smalltalk and Java programs [Ste99]. However, he found that most pointer stores occurred in the youngest objects. Hence few pointers required recording.
The second assumption is generally correct, at least for functional languages [Ste93]. However, numerous papers have erroneously claimed to prove this by instead showing that most objects die young. These statements are not equivalent. The first statement concerns the survivor function $s(t)$, the fraction of objects still live $t$ time units after they were allocated. The survivor function is a monotonically decreasing, non-negative function for which $s(0) = 1$. (Note that the probability distribution function (PDF) of object lifetimes is given by the negative time derivative $-s'(t)$ [Cox84].) The second statement concerns the mortality function $m(t)$, the fraction of objects still live at time $t$ that die in the next instant. The mortality function is related to the survivor function by $m(t) = -\frac{s'(t)}{s(t)}$. The mortality function is generally not smooth nor decreasing.

Baker gives an example where most objects die young but mortality is independent of age [Bak93]. Imagine that object lifetimes for some program are described by an exponential distribution, so $s(t) = e^{-t/\tau}$. It follows from the above definition that the mortality rate is $m(t) = \frac{1}{\tau}$. In other words, an object's chance of dying in the next instant is independent of its age. It follows that a generational collector will always copy the same number of objects as a standard collector.

While the above example is purely hypothetical, it suggests the possibility of other collection schemes. Clinger and Hansen present a “non-predictive” collector for the above example [CH97]. This collector considers only the oldest objects. Survivors are “renewed” by placing them with the youngest objects. Stefanović, McKinley, and Moss describe alternative age based schemes [Ste99, SMM99]. In particular, they find that an oldest-first collector with a sliding collection window can outperform a classic youngest-first collector. It is based on the observation that the older half of a region of recently allocated data is more likely to be garbage than the younger half, since the survivor function is monotonically decreasing. This collector seems to perform well because it often finds a “sweet spot” where objects have the least probability of surviving.

More generally, any garbage collector that concentrates its efforts on those objects most likely to be garbage will outperform a standard collector, at least if the cost of tracking intergenerational pointers is low. In the next section, we present a collector that uses type—not age—to predict an object's mortality.

---

1Let $T$ be a random variable equal to the time after allocation that some object dies. Informally, $m(t) = Pr(t + \Delta \geq T | T \geq t)$ for an infinitely small value of $\Delta$. By the definition of conditional probability, it follows that $m(t) = \frac{Pr(t + \Delta \geq t)}{Pr(T \geq t)}$. Note that the top value of this fraction is equal to the PDF of $T$ evaluated at $t$, i.e., $-s'(t)$, while the bottom value is equal to $s(t)$. 
Chapter 4

Type-Based Collection

This chapter presents a novel collector that uses type to predict an object’s mortality. The proposed collector segregates objects by type and collects only those objects most likely to be garbage. To our knowledge, this is the first description of type-based collection, although the idea had been hinted at by Dieckmann and Hölze [DH99].

Section 4.1 provides motivation for our collector by demonstrating a correlation between Java classes and mortality. Section 4.2 gives the basic collection algorithm and describes some of its design parameters. Section 4.3 discusses in detail how to choose the collection region. Finally, Section 4.4 compares type-based collection with generational collection.

4.1 Motivation

In typed languages, age is not necessarily the only indicator of mortality. We found a strong correlation between the type of an object and its probability of surviving the next collection. Table 4.1 shows the approximate average mortalities of selected classes in _209.db, a database written in Java from the SPEC JVM98 benchmark suite [Cor]. (See Section 6.1.) These mortalities were estimated by counting the number of bytes of each type immediately before and after a collection.

In this example, instances of Vector$1 and Entry[] have high mortality rates. The remaining types, including String and char[] have low mortality rates. This is readily explained. Vector$1, for example, is an inner class used to iterate through the elements of a Vector. Instances of this class are only live for the duration of the iteration. On the other hand, strings (an instance of String coupled with an instance of char[]) are mainly used in this benchmark as fields of database records. Instances of these classes are live for the duration of the benchmark.
Hence, it makes sense to avoid repeatedly copying strings.

Ideally we would measure the distribution of object lifetimes for each type, but we do not have the appropriate infrastructure. Dieckmann and Hölze provide this data for certain categories of types (e.g., all arrays of references) for the JVM98 benchmark suite [DH99]. Their results also show that objects of different types have substantially different mortalities and lifetimes.

### 4.2 Basic Scheme

Based on the above observation, we propose segregating objects by type and concentrating collection efforts on instances of those types most likely to be garbage. In the simplest case, the heap is partitioned into two spaces: *hotspace* and *coldspace*. These spaces are analogous to newspace and oldspace in an age-based generational scheme. Initially, all objects are allocated into hotspace. At the first collection, the collector measures the number of bytes by type before and after collection. If instances of some type occupy $b$ bytes of the heap before collection and $a$ bytes after collection, the average survival rate of the type is $a/b$. The average mortality rate is $1 - a/b = (b - a)/b$.

Now, all types are partitioned into two sets, *cold* and *hot*. Hot objects (i.e., instances of a hot class) have high mortalities; cold objects have low mortalities. The next section discusses the details of partitioning.

Subsequently, all hot objects are allocated in hotspace, while cold objects are allocated directly to coldspace. When memory is exhausted, a minor collection occurs. A minor collection in this scheme considers only objects in hotspace, much as a minor collection in a generational scheme considers only objects in newspace. When a minor collection cannot free enough memory, a major collection occurs. A major collection considers all objects.

<table>
<thead>
<tr>
<th>Type</th>
<th>Mortality Rate (%)</th>
<th>Volume as % of Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.Vector$1</td>
<td>100</td>
<td>43.3</td>
</tr>
<tr>
<td>spec.benchmark..209.db.Entry[]</td>
<td>99.4</td>
<td>26.4</td>
</tr>
<tr>
<td>char[]</td>
<td>21.9</td>
<td>12.3</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>21.9</td>
<td>5.2</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>20.9</td>
<td>9.2</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>20.4</td>
<td>1.5</td>
</tr>
<tr>
<td>Total</td>
<td>76.6</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table 4.1: Example mortality rates in .209.db. Fully-qualified class names (i.e., including the class' package) are shown.
Because type-based collection is similar in structure to age-based generational collection, the considerations listed in Section 3.1 still apply. The most important question is where to place the survivors of a collection. In a purely type-based collector (i.e., one devoid of any age-based benefits), the collector always copies hot objects to hotspace and cold objects to coldspace. (In the terminology of Section 3.1, the number of times an object is copied before promotion, \( n \), is infinite.) This can be difficult to implement, as the collector must maintain two "NEXT" pointers. As a compromise, we propose copying survivors of minor collections to hotspace, and survivors of major collections—hot or cold—to coldspace. However, it may make sense to copy all survivors to coldspace for the following reasons:

- If there is a correlation between age and mortality—as well as type and mortality—this will result in less copying for the usual generational reasons. In other words, it may benefit a collector to consider both object type and age.

- If a cold object references a hot object \( O \), moving \( O \) (and all its children) to coldspace may reduce the number of intergenerational pointers.

Another key question is when to recompute the type partition. For programs whose behavior does not vary much with time, the partition need only be computed once. For more complicated programs, the collector may need a mechanism to detect when the partition is inefficient and recompute it as necessary. In this case, the collector could consider the time-averaged mortality of each type. By doing so, the collector prevents unusual or inaccurate samples from causing a bad partition. We have not investigated such mechanisms.

Finally, object mortality could be measured in other ways. First, instead of considering every object, the collector might choose to randomly sample the heap. (This is discussed more in Section 5.2.3.) Secondly, one might gather mortality statistics during a trial run of the benchmark, compute the type partition offline, and feed this partition to the collector during subsequent runs. This feedback model admits more time-consuming partitioning algorithms that examine how mortalities change with time. Finally, it may even be possible to estimate mortalities of each type at compile-time, though this is likely to be difficult.
4.3 Type Partitioning

At the core of the type-based collector is a means of partitioning types into hot and cold sets so that the mortality rate of hot types is significantly higher than the overall mortality rate. This section describes the issues involved in the partitioning process. First, we establish a theoretical estimate of the efficiency of a type-based collector. Efficiency is measured by the “mark/cons” ratio, the number of bytes copied during garbage collection divided by the number of bytes allocated. (This name is historical; in early LISP systems, new objects were created by the cons procedure.) The lower the mark/cons ratio, the more efficient the collector. Using this result, we give a partitioning algorithm that optimizes the mark/cons ratio in the absence of intergenerational pointers. Next we provide a more refined algorithm that accounts for intergenerational pointers. We conclude with a discussion of optimal partitioning algorithms.

4.3.1 Theoretical Efficiency

Assume for simplicity that survivors of a collection (major or minor) are always placed in coldspace. Assume we have partitioned all classes into sets $C$ and $H$, for “cold” and “hot”, respectively. Let $n$ be the size of the heap. Let $x$ be the fraction of allocated object in $H$. Note that $1 - x$ is the fraction of allocated objects in $C$. Let $f$ be the fraction of instances of $H$ surviving a minor collection. Let $f'$ be the fraction of instances of $C \cup H$ surviving a major collection. Assume that $f$ and $f'$ are constant. (This means there are no age-based benefits.) For simplicity, also assume the absence of intergenerational pointers. In Section 4.3.3, we will consider their effects.

Now consider a series of minor collections preceding a major collection. Before the first minor collection, $n$ bytes of objects can be allocated on the heap. Of these objects, $xn$ bytes will be hot. Now a minor collection occurs. Of the hot objects, $fxn$ bytes will survive and $(1 - f)xn$ bytes will be freed. (All cold objects survive since they are uncollected.) This is illustrated in Figure 4-1.

Let $T(n)$ be the volume of objects (in bytes, for concreteness) allocated on a heap of size $n$ before the major collection. $T(n)$ can be described by the following recurrence:

$$T(n) = n + T((1 - f)xn)$$

$$T(0) = 0$$

Let $G(n)$ be the volume of objects copied during these minor collections. During the first minor
collection, the \( fxn \) bytes of hot objects that survive must be copied. Thus \( G(n) \) can be described by the following recurrence:

\[
G(n) = fxn + G((1 - f)xn)
\]

\[
G(0) = 0
\]

It can be shown by induction that

\[
T(n) = \frac{n}{1 - x + fx}
\]

\[
G(n) = \frac{xnf}{1 - x + fx}
\]

Now following the minor collections, one major occurs. \( f'n \) bytes must be copied during this collection. So given \( \{C, H\} \), the mark/cons ratio for the sequence of minor collections followed by one major collection is

\[
M_{C,H} = \frac{G(n) + f'n}{T(n)} = xf + f'(1 - x + xf)
\]

### 4.3.2 Algorithm I

Algorithm I partitions types into \( \{C, H\} \) so that the mark/cons ratio \( M_{C,H} \) is maximized, assuming the absence of intergenerational pointers. The basic idea is that the mortality rate of the collected region should be maximized, but the collected region should be large enough so that (expensive) major collections are infrequent.
Let $H$ be a set of hot classes and $U$ be the set of all classes. For any class $u$, let $b(u)$ and $a(u)$ be the volume of objects of type $u$ immediately before and after a major collection. Let $x(Y)$ denote $\sum_{y \in Y} x(y)$. $b(U)$ and $a(U)$ are the total number of bytes before and after the major collection. Pseudocode for Algorithm I is shown in Figure 4-2. Note that Algorithm I runs in linear time with respect to the number of classes.

\[
\text{for each class } i:
\begin{align*}
\quad &\text{if } [a(U) + b(U)]a(i) - a(U)b(i) < 0: \\
\quad &\quad H = H \cup \{i\} \# \text{Place } i \text{ in hotspace} \\
\quad &\text{else:} \\
\quad &\quad C = C \cup \{i\} \# \text{Place } i \text{ in coldspace}
\end{align*}
\]

Figure 4-2: Algorithm I. $H$ is the set of hot types and $C$ is the set of cold types.

The justification of Algorithm I follows. The values for $x$, $f$, and $f'$ described in the previous section can be approximated by

\[
x = \frac{b(H)}{b(U)}
\]
\[
f = \frac{a(H)}{b(H)}
\]
\[
f' = \frac{a(U)}{b(U)}
\]

Hence the mark/cons ratio $M_{C,H}$ is

\[
M_{C,H} = \frac{b(H) a(H)}{b(U) b(H)} + \frac{a(U)}{b(U)} \left(1 - \frac{b(H)}{b(U)} + \frac{b(H) a(H)}{b(U) b(H)}\right)
\]
\[
= k_1 (k_2 a(H) - k_3 b(H) + k_4)
\]
\[
= k_1 k_4 + k_1 \sum_{h \in H} (k_2 a(h) - k_3 b(h))
\]

where

\[
k_1 = \frac{1}{b(U)^2}
\]
\[ k_2 = a(U) + b(U) \]
\[ k_3 = a(U) \]
\[ k_4 = a(U)b(U) \]

It follows that \( M_{C,H} \) is minimized when

\[ H = \{ h | k_2 a(h) - k_3 b(h) < 0 \} \]

Remember, however, that Algorithm I assumes no intergenerational pointers. If the partition were to result in many pointers between coldspace and hotspace, there would be a high overhead from scanning intergenerational pointers.

### 4.3.3 Algorithm II

Algorithm II considers intergenerational pointers when partitioning types. It has two conflicting goals: maximize the mark/cons ratio while also minimizing the number of intergenerational pointers. Algorithm II uses a heuristic to group related classes into pseudocomponents. These pseudocomponents are then partitioned into cold and hot sets using Algorithm I.

First, some terminology is in order. Let a runtime class graph \( G \) be a weighted directed graph so that

1. Each vertex of \( G \) represents a class loaded by the program
2. If \( G \) contains an edge with weight \( w \) from \( A \) to \( B \), the program contains \( w \) references from instances of \( A \) to instances of \( B \).

The runtime class graph changes during the program’s execution. In particular, the fact that there is no edge between \( A \) and \( B \) does not imply that there could not be in the future. Figure 4-3 shows an example runtime class graph and its relationship to the so-called class graph and object graph. Appendix A shows example runtime class graphs for several benchmarks.

We say that a class \( A \) is a pseudopredecessor of a class \( B \) if the intergenerational pointers from placing \( A \) in coldspace and \( B \) in hotspace would be prohibitively expensive. In this case, the edge from \( A \) to \( B \) in the runtime class graph is said to be significant. This is intentionally vague and might be defined several possible ways:
Figure 4-3: Relationship between class graph, runtime class graph, and object graph. The class graph is the only graph that shows inheritance relationships.
• There are more than \( k \) references between instances of \( A \) and \( B \) for some empirically determined \( k \), i.e., \( w > k \). This avoids intergenerational pointers. (Recall that \( w \) is the number pointers between \( A \) and \( B \).)

• The expected number of references between dead instances of \( A \) and (live or dead) instances of \( B \) exceeds \( k \), i.e., \( \frac{b(A) - a(A)}{b(A)} \times w > k \). This avoids wasted copying because of intergenerational pointers. (Recall that \( \frac{b(A) - a(A)}{b(A)} \) approximates the probability that an instance of \( A \) is dead.)

We define the pseudocomponents of a program to be the smallest partition \( \{S_1, S_2, \ldots \} \) of the set of all classes so that no \( S_i \) contains a pseudopredecessor of \( S_j \), for \( i \neq j \). That is, the pseudocomponents are the components of the runtime class graph with all insignificant edges removed.

Algorithm II takes a snapshot of the runtime class graph (or its transpose) as input. This can be computed with relatively little overhead during the first major collection by recording the types of all pointers encountered. For example, when the collector traverses a pointer from a class \( A \) to a class \( B \), it increments the edge from \( A \) to \( B \) in the runtime class graph. Then, the algorithm finds the pseudopredecessors of the program using modified depth-first search on the transposed runtime class graph. (Alternatively, one could remove the insignificant edges and then perform standard depth-first search.) Each pseudocomponent is treated exactly like a normal class in Algorithm I; it is added to hotspace if doing so will increase the overall mark/cons ratio. This is shown in Figures 4-4 and 4-5.

4.3.4 Optimal Algorithms

Clearly the choice of partitioning algorithm is crucial. Unfortunately, it is impossible to estimate the cost of intergenerational pointers from the runtime class graph alone (or the static class graph, for that matter). Consider, for example, the program illustrated in Figure 4-3. Here, an instance of type \( A \) points to two distinct instances of type \( C \), while the instances of \( C \) point to a single instance of type \( D \). If all classes but \( D \) were placed in hotspace, it would appear from the runtime class graph that each minor collection would have to copy two objects because of intergenerational pointers. However, the object graph reveals that a minor collection would only have to copy one object. The fundamental problem is that one cannot extract the object graph from the runtime class graph.

We can simplify the problem somewhat by assuming a fixed penalty \( p[A, B] \) of putting class \( A \) in coldspace and class \( B \) in hotspace—regardless of whether other classes are in hotspace or
algorithm2(G):

# Precondition: G is a runtime class graph (V,E), where
# - V is the set of all classes loaded by the program
# - E is a set of weighted edges. The presence of an edge (v1, v2, w)
# means that w references were found between instances of class v1 and
# class v2. Of course v1 and v2 must be elements of V.
# Postcondition: returns a the set of cold classes C and hot classes H as
# a pair of sets (C,H), so that C+H=V. C and H are chosen in a way that
# maximizes the expected efficiency of the collector and minimizes the
# number of intergenerational pointers.

(V,E)=G
C={}
H={}

def unmark(v):
    for each v in V:
        if v is unmarked:
            v=stack.pop()
            for each child v2 of v1:
                w=weight of edge between v1 and v2
                if v2 is unmarked and significant(v1,v2,w):
                    stack.push(v2)
                    S=S+{v2}
                    mark v2

    #2. Looking at entire pseudocomponent, decide whether to place
    # in depth coldspace or hotspace.
    if profitable(S):
        H=H+S
    else:
        C=C+S

    return (C,H)
profitable(S):
  #Precondition: S is a set of classes
  #Postcondition: returns true if all members of S should be placed in
  # hotspace, or false if they should be placed in coldspace, assuming
  # NO intergenerational pointers.
  before=0
  after=0
  #Calculate byte counts of all classes in S. b(C) and a(C) are the
  #number of bytes of C before and after the last collection, as
  #defined in Section 4.3.1.
  for all classes C in S:
    before=before+b(C)
    after=after+a(C)
  #Now compute the profitability, as in Algorithm I (Section 4.3.2)
  profit=[a(U)+b(U)]after-a(U)before
  return profit < 0

significant(v1,v2,w):
  #Precondition: v1 and v2 are classes, and w is the number of
  # intergenational pointers between them.
  #Postcondition: returns true iff placing v1 in hotspace and v2 in
  # coldspace will cause an unacceptable overhead from
  # intergenerational pointers. There are many possible
  # heuristics; the simplest one is shown.
  return w>k

Figure 4-5: Algorithm II: helper functions
coldspace. In other words, this assumes the cost of intergenerational pointers from $A$ to $B$ is dominated by copying dead instances of $B$ (i.e.: nepotism from $A$ to $B$), and not the copying of objects pointed-to by $B$. We will not specify exactly how this penalty is determined. In addition, each class $A$ has an associated value $s[A]$ equal to the savings in copying—positive or negative—by putting that class in hotspace, as determined in Section 4.3.2. Hence the goal is of the partitioning algorithm is to find the partition $\{C, H\}$ that maximizes

$$f(C, H) = \sum_{h \in H} s[h] - \sum_{c, h \in E} p[c, h]$$

We do not know of a polynomial time algorithm to optimize this function. We do not even know of a polynomial time algorithm to verify a partition is optimal. The partitioning problem has similarities to other graph problems, but we have not proved or disproved that it is NP-hard. However, we suspect that Algorithm II is good enough for most programs, if tuned properly.

4.4 Discussion

A type-based collector will only perform well if it can find a type partition meeting the following criteria:

1. Instances of hot classes have significantly shorter lifetimes than instances of cold classes

2. There are few pointers between cold and hot objects, and very few pointers between dead cold and dead hot objects

If the first condition is not met, the type-based collector will not be more efficient than a standard collector. If the second condition is not met, the type-based collector will incur a high overhead from scanning intergenerational pointers and copying dead objects. Preliminary evidence for several benchmarks suggests that such partitions do exist, but it remains to see whether a broad range of programs meet these criteria.

If these conditions are met, a type-based collector has all the potential advantages of a generational collector: increased copying efficiency, better cache locality, and shorter average pause times. Again, this thesis is concerned only with those factors that affect the overall cost of garbage collection: efficiency and (to a lesser extent) cache locality.

In addition, the type-based collector has several advantages over generational schemes. First, an age-based collector copies each long-lived object at least once, when it is promoted from newspace
to oldspace. On the other hand, a type-based collector may never copy such objects; they are placed directly in oldspace. More importantly, the type-based collector may better adapt to programs and eliminate more write barriers than an age-based collector. These two points are discussed below.

4.4.1 Adaptation

We were initially drawn to alternative garbage collection techniques because we observed that different programs placed very different demands on the collector. For example, some programs had many long-lived objects, while others had few such objects. This led us to believe that any high-performance collector must be adaptive; no program should exhibit its worst-case performance. There has been little work in this area. (See Section 3.1.) Unfortunately, we found it difficult to fashion adaptive age-based collectors. It seems that statistics about object lifetimes are difficult to measure at runtime.

In contrast, it is easy for a type-based collector to adapt to the program. Statistics about object types are readily and cheaply measured. While it is possible to make a type-based collector that is not adaptable—one might decide a priori to place all arrays in coldspace, for example—there seems no good reason to do so. Ultimately we feel the ability to adapt gives type-based collection an advantage over generational techniques.

4.4.2 Write Barrier Elimination

Recall from Section 3.2 that a generational collector cannot statically eliminate most write barriers in languages such as Java that separate object allocation and initialization. This does not necessarily mean more cards will be marked, as each write barrier may check whether a pointer is indeed intergenerational. However, it does mean that every pointer store will require some overhead.

In contrast, a type-based collector can disable many write barriers using type information; a store from type \( A \) to \( B \) only needs a write barrier if \( A \) is cold and \( B \) is hot. In the example from Section 4.1, the collector can disable all write barriers for stores from \( \text{String} \) to \( \text{char[]} \). This increases overall performance in two ways; the mutator is faster since fewer pointer stores have write barriers, and the garbage collector is faster since fewer cards are spuriously marked.

The details of write barrier elimination are highly dependent on the programming language and its implementation. For example, if the type partition changes after it is initially computed, certain write barriers might require re-enabling. Also, the number of write barriers that can be eliminated will depend on the amount of type information available at compile time, and on the language's
treatment of subtyping. Section 5.4 explains in detail how our collector for a Java Virtual Machine eliminates (and "un-eliminates") write barriers.
Chapter 5

Implementation

We have implemented the type-based collector for JFEX, a high-performance Java Virtual Machine (JVM) in development at Compaq for Alpha workstations running Tru64 UNIX. JFEX features a Just-in-Time Compiler (JIT) that dynamically compiles frequently used methods to Alpha machine code. Other methods are interpreted to avoid the overhead of compilation. Compilation is done lazily, i.e., when a method is called.

Implementing a collector for a large and (necessarily) complicated virtual machine is potentially time-consuming. Alternatively, we could have simulated the collector using program traces [DH99], but it is our experience that implementation details can dramatically affect overall performance. Likewise, implementing the collector for a research virtual machine might artificially mitigate the cost of garbage collection if the compiled code were slower. Fortunately, we were able to base our implementation on the original JFEX collector. This saved us from rewriting much tedious code such as root scanning.

This chapter describes the details of that implementation. Section 5.1 describes the original JFEX collector. Section 5.2 describes several changes we made to enable type-based collection. Finally, Sections 5.3 and 5.4 describe two optimizations that proved critical for good performance: (1) triggering major collections and (2) eliminating write barriers.

5.1 Original Collector

JFEX originally used Bartlett’s mostly-copying collector [Bar88], a conservative extension of the basic Cheney algorithm. (See Chapter 2). Bartlett developed this collector for use in uncooperative environments such as C and C++. Hence it requires no knowledge of the layout of registers or the
In this scheme, the heap is divided into contiguous blocks of memory called pages. (Here, "page" has nothing to do with virtual memory pages.) Pages in tospace are labelled with an odd identifier; pages in fromspace are labelled with an even identifier. Hence fromspace and tospace are no longer contiguous. Objects may be moved to tospace in one of two ways: by physically copying the object to a tospace page in the usual fashion, or by relabeling the page on which the object resides. Allocation is now a two-step process; first, find a page with free space. (This is generally the page used for the last allocation.) Then, return a pointer to the beginning of free memory in that page. Large objects may span multiple pages. We shall call the location used for the next allocation the allocation point.

Garbage collection now works as follows: first, the registers and stack are conservatively scanned for potential pointers to the heap. For each possible pointer, the page on which the pointed-to object resides is promoted to tospace by relabeling its space identifier, and the page is queued. Note that no register or stack locations are modified. Now, each page on the queue is scanned in the same way that the Cheney algorithm scanned tospace. All pointed-to objects in fromspace are copied to the location NEXT in tospace. The only major difference is how the NEXT pointer is computed; whereas the Cheney algorithm simply increments NEXT, Bartlett’s algorithm calculates NEXT with the two-step allocation process described above. It follows that when a new page is allocated during collection, it must be added to the queue.

Conservative collection was required in Java 1.0 since native methods could contain pointers to the heap. In Java 1.1 and 1.2, native methods must register heap references so a copying collector can be used [Sun97]. Nevertheless, JFEX uses Bartlett’s collector for several reasons:

- In our experience, conservative stack scanning is much faster than precise stack unwinding and results in negligible excess retention.

- Large objects can be collected by simply promoting the pages on which they reside. This dramatically reduces copying cost.

- There is little need for cooperation between the compiler and the garbage collector, so development is greatly simplified.

JFEX also supports generational collection. When generational collection is enabled, pages with odd space identifiers are considered oldspace; pages with even space identifiers are considered
newspace. Minor collections only consider pages with even identifiers. Pages are normally allocated into newspace, but they are allocated into oldspace during collection. Promoted pages are always given odd identifiers. The collector uses compiler-emitted write barriers and a card marking scheme; these are discussed in detail below. All survivors of a minor collection are promoted to coldspace.

5.2 Basic Changes

To convert the existing collector from a generational to a type-based scheme, we had to make the following changes:

- Augment the collector with the set of cold and hot types
- Change allocation routines to place objects in the appropriate space
- Add code to the collector to periodically sample the heap and recompute the type partition

This section discusses these modifications. Note that the basic collection algorithm required no changes.

5.2.1 Representing Hot and Cold Sets

The type-based collector maintains a list of all hot and cold classes in order to allocate objects to the correct space. The choice of data structure is somewhat tricky because programmers can dynamically load classes via the static method `Class.forName`. This method allows programmers to specify a class loader, often to download classes off of the internet. As a corollary, a string name such as `'java.lang.Object'` is not sufficient to uniquely identify a class; a class loader must be specified as well. To prevent confusion, the JVM assigns a unique CID to each class when it is loaded. From a CID, one can look up a `class.t` data structure containing the name of the class, name and type of all fields, etc.

Our implementation represents the partition of classes with a bit vector called the `hotlist`. If bit $i$ of the hotlist is set, the class with CID $i$ is hot; otherwise it is cold. It is possible for a hot class and a cold class to have the same name, but they will have distinct CIDs.

In addition each `class.t` is augmented with an extra bit that is set if and only if the class is hot. These bits contain the same information as the bit vector described above, but they provide an efficient way to eliminate certain write barriers with runtime checks. (See Section 5.4.) Actually,
each hot "bit" is implemented with an entire 32 bit longword, since that is the smallest unit of memory that all Alphas can address in a single instruction. (Newer Alphas can read a single unaligned byte in one instruction, but this requires expensive software emulation on earlier models.)

5.2.2 Allocation and Promotion

When the VM starts, the hotlist is empty, so all objects are initially allocated to coldspace. Note that all objects could have been allocated to hotspace as well, since the first collection is always a major collection. Once the hotlist is computed, an object must be placed in hotspace or coldspace depending on its type. A simple way of implementing this is to look up each class in the hotlist when an instance is allocated. This adds a few extra instructions to every allocation. Because the current garbage collector normally only takes about 10 machine instructions to allocate an object, we suspect that this overhead would be noticeable.

However, our implementation places objects in the appropriate space with no overhead. When JFEX loads a class $C$, it generates a custom allocator for that class. This allocator is simply a few machine instructions that branches to a more general purpose allocator. The custom allocator is always called to instantiate an object and allows certain kinds of optimizations. We augmented the JVM to record the type and location of each generated allocator. When the hotlist is recomputed, each allocator is modified to subsequently allocate its objects to hotspace or coldspace appropriately.

As an optimization, our allocator avoids interleaving hot and cold pages in the heap. Instead it tries to place hot pages at one end of the heap and cold pages at the other end. This has numerous benefits. First, marked cards tend to be clustered together, so identifying intergenerational pointers is faster during minor collections. Secondly, allocating hot pages is faster, since the collector need not iterate past many cold pages to find a free page. Finally, this seems to increase cache locality, since long-lived objects are placed closer together.

Our collector uses the promotion policy suggested in Section 4.2. (Minor collections copy to hotspace; major collections copy to coldspace.) We chose this policy to mitigate the effects of age-based collection on our results. ("Pure" type-based collection, which only allows cold objects in coldspace, proved too difficult to implement.) Our policy is implemented by using a special allocation routine during garbage collection. During major collections, new objects are allocated to hotspace. During minor collections, new objects are allocated to coldspace. As a result, we had to modify the collector so it does not unmark all cards following a minor collection. After the collector has scanned a card, it unmarks the card if and only if it found no intergenerational pointers.
5.2.3 Recomputing the Hotlist

Our implementation normally only recomputes the hotlist during the first collection. During normal major collections—i.e., when a minor collection does not free enough memory—the hotlist is not recomputed. We chose this policy because most of our benchmarks vary little with time.

As discussed in Section 4.2, the cost of mortality estimation can be made arbitrarily small by randomly sampling the heap instead of counting the exact volume of each type before and after collection. Sampling is particularly easy to implement using Bartlett’s heap layout; the garbage collector chooses a random subset of pages and only counts the objects on those pages. However, we found that random sampling is unnecessary; because the collector only recomputes the hotlist once during the program’s execution, examining the entire heap is rarely expensive. On the other hand, experimental collectors that used random sampling occasionally performed poorly because of inaccurate mortality estimates.

Once object mortalities have been estimated, the collector uses Algorithm II to partition types. For this reason, the first collection must also construct a runtime class graph as it traverses pointers. When running Algorithm II, references from a class $A$ to class $B$ are considered “significant” if there are more than 10 dead pointers. (See Section 4.3.3.) This value was discovered empirically.

Earlier versions of our collector used Algorithm I, but it proved unsatisfactory; this is discussed in Section 6.2. After types have been partitioned, the runtime class graph is discarded.

There is one additional complication. Java allows a programmer to force garbage collection by calling the method `System.gc()` [GJS96]. The garbage collector is free to force a major or minor collection, or do nothing at all. We conjecture that `System.gc()` often suggests that the usage of types is likely to change. For example, the SPEC harness calls `System.gc()` between benchmarks. For this reason, we would like the collector to force a major collection and resample the heap. However, it is grossly inaccurate to sample the heap during a major collection that follows a minor collection; since cold objects have had extra time to die, their mortalities will be artificially high. As a result, the collector will chose a bad partition. Hence `System.gc()` forces a major collection and ensures that the following collection is major as well. The hotlist is only recomputed after the second collection.
5.3 Forcing Major Collections

An early version of our implementation only forced major collections when a minor collection was unable to reclaim enough space to satisfy an allocation. As a result, the collector would sometimes fall into a series of minor collections that were far less efficient than a major collection. Often, a large number of objects in coldspace would die, but the collector would continue to traverse intergenerational pointers from these objects, resulting in much unneeded copying. See Section 6.2 for details.

To prevent behavior, our collector monitors the retention rates of major and minor collections. Whenever the retention rate of a minor collection exceeds that of the last major collection, a major collection is forced. Retention rates are easily estimated by counting the number of allocated pages before and after collection. (Our collector maintains these numbers for reporting statistics to the user.)

5.4 Eliminating Write Barriers

As discussed in Section 4.4.2, one of the advantages of a type-based garbage collector is that many write barriers can be eliminated by using type information. Our implementation aggressively eliminates write barriers using compile-time analysis and runtime checks.

There are two Java bytecodes that can cause intergenerational pointers:

aastore Stores a reference into an array of references

putfield Stores data into the field of an object. Fields of primitive type do not need special consideration.

When creating a pointer from an object $A$ to an object $B$, $A$ is considered the source object and $B$ is considered the target object.

In addition, the JVM implements (in C) two standard Java methods that can cause intergenerational pointers:

System.arraycopy Copies the contents of one array to another.

Object.clone Given an object, creates a new object of the same type with the same state

First, this section discusses JFEX’s original card marking scheme. Then it describes exactly how our implementation eliminates each kind of write barrier.
5.4.1 Original Card Marking

The original collector uses a card-marking scheme with software write barriers to record intergenerational pointers. Storing a pointer into an object requires three steps, each requiring a single instruction:

1. Right shift the source object's address to get the address of its card-marking bit
2. Store the pointer into the source object
3. Mark the corresponding card bit

When the JVM interprets a `putfield` bytecode, these instructions are executed by a call to `gc_store_ptr`. When the JVM compiles this bytecode, the three instructions are inserted “inline” into the generated code. On the other hand, `aastore` bytecodes always result in a call to `comp_array_store`, which includes these three instructions. Likewise, these three instructions are found in `JVM_Clone` and `JVM_ArrayCopy`.

Our implementation uses a card shift of 7. In other words, each card is $2^7 = 128$ bytes. Each mark “bit” is actually a byte, since that is the smallest unit of memory that can be modified in one instructions on newer Alphas (EV5.6 and later). The `stb` (“store byte”) instruction is used to mark and unmark cards. On older processors, the `stb` instruction must be patched with the `stq_u` (“store 64 bit unaligned quadword”) instruction; this loses some precision in the card-marking but avoids the expensive emulation of `stb` by the operating system.

The order of these instructions is significant. Consider what would happen if a card were marked after the pointer was stored. Thread $T_1$ first marks a card $C$ corresponding to the source object. Then another thread $T_2$ triggers garbage collection, suspending thread $T_1$. During the collection process, card $C$ is unmarked since it contains no intergenerational pointers. When garbage collection finishes, thread $T_1$ continues and stores the pointer. However, since no corresponding card is now marked, the target object may be prematurely collected in the future.

One subtlety: if an object on card $i$ contains an intergenerational pointer, either card $i$ or $i-1$ may be marked. The reason is as follows: when emitting the write barrier for a `putfield` instruction, the offset of the field of the source object relative to its start is known at compile time. To save a single instruction, the compiler emits a “store with offset” instruction (e.g., `stb a1, 32(a0)`) to mark the card instead of first storing the address of the field (e.g., contents of a0 plus 32) into a register. Let the card shift be $K$. In the above example, the compiler would actually mark the card
byte \((32 \gg K) + (a0 \gg K)\) instead of \((32 + a0) \gg K\). (Here the notation \(a \gg b\) means the logical right shift of \(a\) by \(b\).) Since \((a \gg K) + (b \gg K) = (a+b) \gg K\) or \(((a+b) \gg K) - 1\), card \(i\) or \(i - 1\) may be marked.

5.4.2 putfield

Ideally the compiler would only emit write barrier code for those \texttt{putfield} instructions from cold to hot objects. Unfortunately, most code is compiled before the hot and cold types have been discovered. An obvious option is delaying compilation until the hotlist has been computed—the JVM can always interpret bytecodes—but this could degrade performance considerably. Furthermore, it does not give the collector the flexibility to change the type partition later if needed.

Instead, the JIT initially emits a write barrier for each compiled \texttt{putfield} instruction. At this time, the JIT records the address of the emitted \texttt{stb} instruction and the type of the source and target object. Whenever the hotlist is changed subsequently, the write barrier can be turned off by replacing the \texttt{stb} instruction with a NOOP. The write barrier can be turned on later by replacing the NOOP with the original \texttt{stb}.

Exactly which write barriers can be eliminated is constrained by the details of the Java Virtual Machine instruction set and the analysis provided by our compiler. When a \texttt{putfield} instruction is compiled, the signature of the source object’s field is known. This provides some information about the target object. For example, if a field’s signature is \texttt{java.util.Vector}, the target object is an instance \texttt{java.util.Vector}—or one of its subclasses. In special cases, the compiler can actually determine the runtime time of the target object:

- If the target object was created by a \texttt{new}, \texttt{newarray}, \texttt{anewarray}, or \texttt{multinewarray} bytecode, the runtime type of the target object is known by looking at that bytecode.
- If the target object was created by an \texttt{aconst.null} bytecode, the runtime type of the target object is null.

In addition, if the target object is a copy of a method argument, the possible runtime types of the target object is sometimes reduced. This includes argument 0 of an instance method, i.e., \texttt{this}.

Figure 5-1 presents the algorithm for disabling spurious write barriers. For each write barrier \(W\) recorded by the compiler, the algorithm checks the type of \(W\)’s target object \(T\). If the runtime type of \(T\) is known, \(W\) is enabled iff \(T\) is hot. Otherwise, \(T\) is enabled iff it is the superclass of a hot class. However, there is an additional subtlety here; the compiler only knows the \texttt{name} of the
target object's type. Hence the algorithm first computes the names of all hot classes. Now, if any hot class has the same name as $T$ (or one of its subclasses), $W$ is disabled.

5.4.3 aastore

It might appear that many of the aastore write barriers could be eliminated at compile-time using techniques similar to those described above. Unfortunately, the compiler does not know the type of the target object. This is because the aastore instruction—unlike putfield—contains no references into the Virtual Machine constant pool. Although it would be possible to occasionally identify the type of the target object with a bit of compiler analysis, many write barriers would probably remain enabled.

Instead we eliminate spurious aastore write barriers with runtime checks. Before storing a pointer into an array, we first check the hot bit of the class of the target object. Figure 5-2 shows this pseudocode; in practice, this is implemented in assembly code for efficiency. This may seem like undue overhead. However, the code must load the class into memory anyway to ensure that the target object is compatible with the source array. (In Java, an array of type $T[]$ may actually contain an array of type $T'[]$, where $T'$ is a subtype of $T$.) Hence only two additional instructions—a load and a conditional branch—are needed to test whether the class is hot.

In truth, our implementation uses some compile-time techniques in addition to runtime tests. Our compiler currently records the locations of all calls to arraystore and the runtime type of the target object—if it is known. The type of the source object is not readily known. When the hotlist is (re)computed, wb.enable.some may replace some calls to arraystore with calls to arraystore.raw. arraystore.raw is exactly like arraystore except that the hot bit in not tested and no cards are ever marked. In the future, we plan on removing this trickery from our code, since it only saves a marginal number of cycles and complicates the compiler.

5.4.4 JVMArrayCopy

JVMArrayCopy copies a subsequence of an array $A$ into an array $B$. For simplicity of discussion, however, we shall assume that JVMArrayCopy copies the entire array. If $A$ contains pointers to hotspace and $B$ is in coldspace, JVMArrayCopy must mark the corresponding card of $A$. The simplest implementation of JVMArrayCopy would simply mark all card bits corresponding to $B$. This is needlessly expensive.

Instead, we use runtime techniques to reduce the number of cards marked. First we load the
disable(wb):
  #Postcondition: ensures that no cards will be marked because of
  # this write barrier
  Replace the instruction at wb with a NOP

enable(wb):
  #Postcondition: ensures that appropriate cards will be marked
  # when this write barrier is executed
  Replace the instruction at wb with its original STB
  instruction

wb_enable_some():
  #Postcondition: all write barriers that can cause an
  # intergenerational pointer are enabled. Some of those
  # write barriers that cannot cause intergenerational
  # are replaced with NOPs.

  #1. Find the names of all hot class and store in the set
  #    s. Find the names of the superclasses of all hot
  #    classes and store in the set ss
  s={}  
  ss={}
  for each class i in the hot list:
    s=s U {name(i)}
    ss=ss U {name(i)}
    for each superclass j of class i
      ss=ss U {name(j)}

  #2. Turn each write barrier on or off
  for each wb in the list of recorded write barriers:
    if the runtime type T of the target object is known:
      if T is in s:
        enable(wb)
      else:
        disable(wb)
    else:
      let T be the compile-time type of the target object.
      if T is in ss:
        enable(wb)
      else:
        disable(wb)

Figure 5-1: Algorithm for eliminating putfield write barriers
array_store(a, i, x):
    #Effects: a[i]=x
    Compute offset of index i into a. If out of bounds, throw an exception.
    Load class of a into register c.
    Load class of x into register c'.
    If c' is not compatible with c, raise an exception.
    If hot bit of c' is set:
        Calculate address of card for a[i] with a right shift.
        a[i]=x
        Mark the card bit corresponding to a[i]
    else
        a[i]=x

Figure 5-2: Storing a reference into an array

class pointers of A and B. If both contain primitive types, no write barriers are needed. (If exactly
one contains a primitive type, a runtime exception is thrown.) Otherwise, there are two cases to consider:

1. The type of B is a subclass of the type of A. In this cases, a type check is needed for every
element x copied from A to B. Since the class pointer of x is already loaded for the type
check, the collector can quickly check its hot bit. A card is marked only if x’s hot bit is set
and B is in coldspace.

2. All elements of A can automatically be stored in B. Hence no per-element checks will be
needed. There are now three sub-cases to consider:

   (a) If B is hotspace, no write barriers are needed.

   (b) If both A and B are in coldspace, then each card i of B is marked if card i of A is
       marked.

   (c) If A is in hotspace and B is in coldspace, then each card of B is marked. Although
       more cards may be marked than necessary, this case seems rare in practice.

The design rationale here is that a class pointer should never be loaded unless the original version
of JVM.ArrayCopy (i.e., without support for write barriers) would have needed it anyway. Hence
per-element checks are not used for case (2).

In the above description, testing whether “X is in hotspace” is done by checking the space of
the page on which X resides. It is not sufficient to simply check the hot bit of X’s class pointer,
since major collections promote hot objects to coldspace. Assume, for example, that arrays $A$ and $B$ are instances of hot classes but were promoted to coldspace by a major collection. Subsequently, pointers are stored from $A$ to objects residing in hotspace. The appropriate cards corresponding to $A$ are marked. If the above scheme only checked the hot bits of $A$ and $B$'s classes, $A$'s pointers would be copied into $B$, but no cards would be marked—an egregious error.

5.4.5 JVM.Clone

Given an object $x$ as input, JVM.Clone instantiates a new object of the same type and state. Again, we use a runtime check to prevent cards from being unnecessarily marked. After cloning $x$ in the usual fashion, JVM.Clone checks the hot bit of $x$'s class. (The class data structure of $x$ is already loaded anyway.) If this bit is set, the newly allocated clone was placed in hotspace, so no intergenerational pointers were created. If the bit is not set, the newly allocated clone is in coldspace, so intergenerational pointers may need recording. In this case, JVM.Clone simply marks each card $i$ of the clone if card $i$ of $x$ is marked. (This is similar to case 2b of JVM.ArrayCopy.) As usual, it is important that the object cloning precedes the card marking in case another thread triggers garbage collection.

The correctness of this algorithm depends on the fact that all survivors of major collections are copied to coldspace. Consider what might happen if this were not the case. Let $C$ be a hot class. JVM.Clone is called on an instance $x$ of $C$. After the hot bit of $C$ is checked, another thread calls System.gc(), forcing a major collection and the recomputation of the hotlist. Now, $C$ is cold, so the clone $x'$ is allocated to coldspace and no cards are marked. If the children of $x'$ were copied to hotspace instead of coldspace, intergenerational pointers would be unrecorded.

5.5 Discussion

Converting our original collector from a generational scheme to a type-based scheme was mostly straightforward. However, our first implementations performed terribly. To achieve good performance, we found it crucial to (1) force major collections when minor collections become inefficient and (2) eliminate as many write barriers as possible. The majority of our development effort was spent eliminating write barriers; incorrectly eliminated write barriers result in unusual fatal errors long after garbage collection has completed.
Chapter 6

Performance Results

This section evaluates the performance of our implementation on six benchmarks. We consider four measures of performance: the number of bytes copied, the number of cards marked, the amount of time spent garbage collecting, and the total benchmark time.

6.1 Method

We tested the garbage collector on six benchmarks from the SPEC JVM98 suite, a standard benchmark suite for Java [Cor]. Five of these are from the current suite; one (.211.anagram) is from an earlier (i.e., pre-release) version of the JVM98 suite. These benchmarks were chosen because they allocate large amounts of memory or are garbage-collection intensive. Table 6.1 shows the amount of data allocated by each benchmarks (in megabytes) and the approximate percentage of the benchmark time spent in the original JFEX garbage collector. The latter value was estimated by running the VM with the -verbose:gc switch to measure the amount of time spent in the garbage collector and dividing this value by the overall benchmark time. Note that garbage collection consumes under 10% of most benchmarks, but consumes a significant portion of .213.javac and .211.anagram. Also, the reported volume of allocation differs from that reported by SPEC; this could be because objects are smaller in JFEX than in the SPEC reference implementation.

The benchmarks were not run in the official SPEC manner. Instead, each benchmark was run from the command line three times in a row with the parameters -M3 -m3 -g -d3000 -a -s100. Only one virtual machine is invoked for all three runs. Thus if the VM compiles bytecodes for the first run, it need not compile the bytecodes for latter runs. Likewise, if the type-based garbage collector samples the heap during the first run, it can use this information during subsequent runs.
All tests were done on a single processor 500 MHz EV6 running Compaq Tru64 UNIX 4.0. The system was lightly loaded (i.e., only one user) for all tests, but daemon processes were not stopped. A fixed heap size of 48MB was used. The choice of heap size is critical; larger heaps require less frequent garbage collection and thus decrease the overall cost of garbage collection. Generally SPEC numbers are reported with much larger heaps\(^1\) though the benchmarks can run in a 20MB heap. We chose 48MB as a compromise between the two extremes. The test machine had considerably more physical memory so paging is not an issue.

A note is needed about the .213_javac benchmark. This program frequently forces garbage collection with calls to System.gc. Unfortunately, each call to System.gc() causes our implementation to force two major collections and recompute the hotlist. (See Section 5.2.3.) Although it would be legal for the implementation to ignore these calls, this would cause problems in other situations. For example, the standard SPEC benchmark harness calls System.gc between benchmarks. This garbage collection time is not recorded in final results, so the garbage collector should run at this point.

### 6.2 Collection Efficiency

If the type-based collector does not copy less data than a standard collector, it is unlikely to perform as well. Hence we modified the collector to count every byte copied during collection. Table 6.2 shows the number of megabytes copied by the standard, generational, and type-based collector.

---

1Official results are categorized by the hardware's memory size: 0-48MB, 48-256MB, or over 256MB. No results were submitted for the first category in 1999. Most results are run with a heap of 256MB or many gigabytes.
during the usual three runs of each benchmark. This is a good measure of the collector's efficiency.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Standard</th>
<th>Generational</th>
<th>Type-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>.228.jack</td>
<td>19.8</td>
<td>44.8</td>
<td>16.3</td>
</tr>
<tr>
<td>.209.db</td>
<td>123.3</td>
<td>22.3</td>
<td>10.0</td>
</tr>
<tr>
<td>.202.jess</td>
<td>43.6</td>
<td>7.6</td>
<td>32.9</td>
</tr>
<tr>
<td>.227.mtrt</td>
<td>85.6</td>
<td>43.6</td>
<td>30.7</td>
</tr>
<tr>
<td>.213.javac</td>
<td>632.8</td>
<td>272.5</td>
<td>408.1</td>
</tr>
<tr>
<td>.211.anagram</td>
<td>633.9</td>
<td>30.9</td>
<td>21.2</td>
</tr>
</tbody>
</table>

Table 6.2: MB copied during collection

In every case, the type-based collector copied less data than the standard collector. In comparison, the age-based collector copied more data in .228.jack than the standard collector, though we suspect this could be minimized by by monitoring the retention rate of minor collections, as discussed in Section 5.3. Furthermore, in all benchmarks but .202.jess, the type-based collector copied less data than the generational collector.

An early version of our collector used Algorithm I (Section 4.3.2). On .211.anagram, this collector copied around 1600 MB—twice as much as the non-generational version. The collector was placing strings in coldspace and character arrays in hotspace, since strings have slightly lower than average mortality rates than character arrays. However, because each string points to a character array, numerous character arrays were copied because of intergenerational pointers from dead strings. This confirms our intuition that a type-based collector must avoid intergenerational pointers.

### 6.3 Card Marking and Scanning

Section 5.4 claimed that many `put field` write barriers could be eliminated at compile-time using type information. Table 6.3 shows the number of write barriers that are actually eliminated. Overall, most write barriers can be eliminated with compile-time analysis. Remember, however, that this is a static count; if, for example, all remaining write barriers were in a tight loop, numerous cards could still be marked.

Table 6.4 shows (1) the number of cards marked and (2) the number of cards requiring scanning during three benchmark runs using the age and type-based collectors. (Obviously card-marking is not needed for the standard collector.) The first quantity approximates the runtime overhead of write
Table 6.3: Static percentage of `put_field` WBs eliminated at compile-time. Recall that `.213.javac` recomputes the partition repeatedly. For this benchmark, only the number of write barriers eliminated by the first partition are reported; later partitions are similar.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>WB eliminated</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.228.jack</code></td>
<td>86.1</td>
</tr>
<tr>
<td><code>.202.jess</code></td>
<td>56.2</td>
</tr>
<tr>
<td><code>.209.db</code></td>
<td>90.9</td>
</tr>
<tr>
<td><code>.227.mtrt</code></td>
<td>61.0</td>
</tr>
<tr>
<td><code>.213.javac</code></td>
<td>60.6</td>
</tr>
<tr>
<td><code>.211.anagram</code></td>
<td>58.9</td>
</tr>
</tbody>
</table>

Barriers; if we unrealistically assume that no two write barriers mark the same card, the number of cards marked equals the number of write barriers executed. The second quantity indicates the GC-time overhead of write barriers. Note that the number of cards requiring scanning is generally much less— and never more— than the number of cards marked.

Table 6.4: Number of cards marked and requiring scanning.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Marked</th>
<th>Scanned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generational</td>
<td>Type-based</td>
</tr>
<tr>
<td><code>.228.jack</code></td>
<td>407126</td>
<td>11790</td>
</tr>
<tr>
<td><code>.202.jess</code></td>
<td>552152</td>
<td>1272995</td>
</tr>
<tr>
<td><code>.209.db</code></td>
<td>150240</td>
<td>8</td>
</tr>
<tr>
<td><code>.227.mtrt</code></td>
<td>179806</td>
<td>121044</td>
</tr>
<tr>
<td><code>.213.javac</code></td>
<td>455493</td>
<td>785127</td>
</tr>
<tr>
<td><code>.211.anagram</code></td>
<td>571543</td>
<td>414492</td>
</tr>
</tbody>
</table>

Results are mixed. For `.202.jess` and `.213.javac`, the type-based collector dramatically increased the number of cards marked and requiring scanning. For `.228.jack`, the type-based collector decreased the number of cards marked, but substantially increased the number of cards requiring scanning. For `.211.anagram` and `.227.mtrt`, the type-based collector decreased both quantities slightly. Finally, for `.209.db`, the type-based collector reduced both quantities to almost zero.

### 6.4 Collection Time

Table 6.5 shows the time (in seconds) spent in the garbage collector during the usual three runs of each benchmark for the standard and type-based collectors. Again, these values were reported with
the -verbosegc option. The actual time spent garbage collecting is probably lower than reported here, since there is some overhead in running the collector with the -verbosegc option.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Standard</th>
<th>Generational</th>
<th>Type-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>.228.jack</td>
<td>0.310</td>
<td>1.023</td>
<td>0.680</td>
</tr>
<tr>
<td>.209.db</td>
<td>1.528</td>
<td>0.851</td>
<td>0.618</td>
</tr>
<tr>
<td>.202.jess</td>
<td>0.775</td>
<td>0.651</td>
<td>2.085</td>
</tr>
<tr>
<td>.227.mtrt</td>
<td>1.925</td>
<td>1.365</td>
<td>1.840</td>
</tr>
<tr>
<td>.213.javac</td>
<td>15.158</td>
<td>18.828</td>
<td>19.268</td>
</tr>
<tr>
<td>.211.anagram</td>
<td>14.236</td>
<td>2.175</td>
<td>1.794</td>
</tr>
</tbody>
</table>

Table 6.5: Collection time (seconds)

The type-based collector is faster than both of the other collectors on those benchmarks where it copies significantly less data, .209.db and .211.anagram. On .228.jack, the type-based is faster than the generational collector, but slower than the standard collector. On .227.mtrt, the type-based collector is faster than the standard collector but slower than the generational collector. On .213.javac and .202.jess, the type-based collector is slower than both of the other collectors.

6.5 Benchmark Time

Table 6.6 shows the overall time spent in the benchmark for the standard and type-based collector. Both the best and worst of the three runs is shown. (This is how JVM98 results are reported. Remember, though, that these tests were not done according to the SPEC regulations.)

Compared to the standard collector, the .213.javac benchmark runs significantly slower with the type-based collector. The .228.jack, .202.jess, and .227.mtrt benchmark run slightly slower with the type-based collector. On the other hand the .211.anagram and .209.db benchmarks runs significantly faster. In fact, the .209.db benchmark runs significantly faster with the type-based collector than with the age-based collector.

6.6 Discussion

6.6.1 Type Partitioning

In general, the efficiency of the type partition appears to determine the overall performance of the collector. On benchmarks where the type-based collector performed best (.211.anagram and
Benchmark Standard Generational Type-based
---
_.228_jack_ 8.173 8.065 8.381
_.209_db  27.756 26.996 21.587
  30.53 30.853 25.237
_.202_jess  7.139 7.201 7.931
  8.419 9.548 10.011
_.227_mtrt  6.806 6.552 6.712
  7.403 7.962 8.17
_.213_javac  16.863 19.583 20.738
  23.416 23.222 24.383
_.211_anagram  8.376 4.206 4.557
  10.086 5.421 5.674

Table 6.6: Overall benchmark time (seconds)

_.209_db_), it copied an order of magnitude less data than the standard collector. On the other hand, when the type-based collector did not copy substantially less data (_202_jess and _228_jack), its overall performance was substantially worse than the standard collector.

A good partition is needed partly because copying does not always dominate the cost of garbage collection. Consider, _227_mtrt_, a benchmark where the type-based collector copies half the data of the standard collector, but is only slightly faster than the non-generational collector. When we profiled the standard collector with DCPI, we found that copying objects (done by the routines _move_ and _memcpy_) only consumed 17% of the overall runtime. Although the type-based collector reduced the cost of copying by 90%, the benefits are somewhat ameliorated by a modest 4% increase in runtime costs from sampling and card scanning routines. This is summarized in Table 6.7.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Standard</th>
<th>Type-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>13171</td>
<td>4390</td>
</tr>
<tr>
<td>memcpy+.OtsMove</td>
<td>8060</td>
<td>1959</td>
</tr>
<tr>
<td>memset+.OtsFill</td>
<td>29965</td>
<td>24627</td>
</tr>
<tr>
<td>measure_volumes</td>
<td>NA</td>
<td>1354</td>
</tr>
<tr>
<td>card_scan_marked</td>
<td>NA</td>
<td>1811</td>
</tr>
<tr>
<td>Sum (runtime+libc)</td>
<td>73023</td>
<td>62332</td>
</tr>
</tbody>
</table>

Table 6.7: Cycles spent per GC routine for _227_mtrt_. An “NA” in a column means that a routine is not used by that particular garbage collector. There is some nondeterminism in the profiling, so comparisons across columns are less important than comparisons within a column. The cost of compiler-emitted write barriers is not included here.
The poor performance of the .213_javac benchmark can be explained by several factors. First, each of its numerous calls to System.gc() causes the collector to sample the heap and recompute the hotlist. (See Section 6.2.) This overhead could probably be eliminated by changing the implementation of System.gc() to check whether re-partitioning is appropriate. A more fundamental problem is that the object graph of .213_javac does not seem amenable to a good partition. Those objects with the highest mortalities—String, char[], and StringBuffer—are pointed to by thousands of objects with higher mortalities. As a result, if the collector chooses an efficient partition (Algorithm I), it will incur a high write-barrier overhead. On the other hand, if the collector avoids intergenerational pointers (Algorithm II), it cannot possibly outperform the standard collector. The problem is exhibited to a lesser extent by other benchmarks, particularly .202_jess. Runtime class graphs for all benchmarks are given in the Appendix.

6.6.2 Write Barriers

Marked cards in hotspace do not incur much overhead at collection time since the collector never scans their contents. Only cards in coldspace are significant, since all pointers on these cards must be scanned. This overhead may erode the benefits from the improvement in collector efficiency. On half the benchmarks—.228_jack, .202_jess, and .213_javac—the number of coldspace cards requiring scanning increased by roughly an order of magnitude relative to the generational collector.

Most of these cards are marked because of pointer stores into newly allocated cold objects. On these benchmarks the collector places a fairly large number of new objects in coldspace. Any pointers written into these objects—e.g., as a result of executing constructors—must be recorded. More generally, any collector that records pointers in newly allocated objects may have this problem. Stefanović found that an oldest-first collector (see Section 3.3) incurred a considerably larger write-barrier overhead than a standard generational collector. He concludes:

... pointer-maintenance costs in a generation collector are very low, confirming past research. The analysis of pointer stores reveals that the explanation lies in the location of stored pointers (in recently allocated data) and not in the temporal direction of pointers (younger-to-older or older-to-younger).

This suggests that aggressive write barrier elimination is a necessity in a type-based collector.
6.6.3 Cache Benefits

The type-based collector sped up .209.db a surprising amount. Although the type-based collector only decreased the garbage collection time of each benchmark by an average of $(1.528 - 0.618)/3 = 0.303$ seconds, the best and worst benchmark times were decreased by 6.17 and 5.29 seconds, respectively. Profiling .209.db with the Compaq (formerly Digital) Continuous Profiling Infrastructure (DCPI) [A+97] shows that the type-based collector reduces the number of cycles spent executing compiled code from 806529 to 709189—a 12% reduction.

It appears that the type-based collector improves the performance of .209.db because it places strings closer together than the standard collector. (Strings are "cold", and cold objects are allocated in a different location than hot objects.) Because the database spends most of its time sorting arrays of strings, the improved locality results in a significant speed-up. A careful examination with DCPI shows a significant decrease in dynamic stalls in two critical routines, Database.shell.sort and String.compareTo(String). (The latter is the comparison routine for the sort.) Exactly what is the cause of these stalls? Our profiling information suggests a decrease in D-cache misses and a significant decrease in TLB misses. This is shown in Table 6.8. We would also expect a decrease in S-cache and B-cache misses, but we have not tested this.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cycles</th>
<th>D-cache Misses</th>
<th>D-TLB Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard</td>
<td>Type</td>
<td>Standard</td>
</tr>
<tr>
<td>Database.shell.sort</td>
<td>301587</td>
<td>261528</td>
<td>111799</td>
</tr>
<tr>
<td>String.compareTo(String)</td>
<td>178696</td>
<td>118122</td>
<td>45553</td>
</tr>
</tbody>
</table>

Table 6.8: The number of cycles, D-cache misses, and TLB misses for two critical routines of .209.db, when using standard and type-based collection.
Chapter 7

Conclusions

7.1 Types or Age?

It is not clear that generational collection is appropriate for Java. Generational collectors make two assumptions: (1) young objects have higher mortality rates than old objects and (2) there are few pointers from old objects to young. The Java programming language encourages an imperative style of programming, so Java programs may not meet these conditions. For this reason, we think it is important to search for alternative collection techniques.

This thesis has demonstrated a strong correlation between the type of an object and its probability of surviving the next collection. Based on this observation, we proposed a novel type-based collector that partitions objects by type and collects only those objects most likely to be garbage. The partition is chosen at runtime according to mortality estimates. In this way, the type-based collector adapts to the program at hand. In addition, the collector uses type analysis to eliminate many write barriers at compile time.

Performance results from an implementation for a Java Virtual Machine are mixed. The collector always copies less data than a standard collector, and generally copies less data than an age-based generational collector. This demonstrates the potential for good performance. In terms of wall clock time, however, the type-based collector only occasionally outperforms a standard collector, and rarely beats the generational collector. It appears that the overhead of write barriers is still substantial in the type-based collector.
7.2 Future Work

We suspect that our collector’s performance could be improved by tuning the type-partitioning algorithm. There is likely a trade-off between avoiding intergenerational pointers and finding an efficient partition. Future partitioning algorithms may also try to improve cache and virtual memory locality. It remains to be seen whether appropriate type partitions can benefit the hit rate of other benchmarks as much as _209.db. Towards this end, the collector may have to partition types into more than two spaces. Finally, the partitioning algorithm may use data collected from a previous run and consider time-varying mortality rates.

Eliminating more write barriers would likely improve our collector’s performance. Two changes to the compiler would make this possible. First, our compiler could do more analysis to determine the runtime types of objects. Secondly, our compiler could support object inlining. When an object B is used exclusively by an object A, B’s data may be “inlined” into A, eliminating the need for B and reducing the number of pointer stores. (This has other benefits, including increased cache locality and reduced overhead from object stores.)

Performance would also be improved if Java supported parameterization by type (parametric polymorphism). This would allow the collector to partition types more finely. For example, instead of treating all instances of Vector the same, the collector could consider Vector[char] and Vector[float] separately.

Future work might also evaluate the appropriateness of type-based collection to a wider variety of Java programs. Specifically, the following questions remain to be answered for broader range of programs:

- Is there a correlation between object type and mortality? If all types have similar lifetime distributions, the type-based collector cannot improve performance.

- Are there numerous pointers between cold and hot types? If so, the type-based collector may not be able to find an efficient partition without incurring a high overhead from write barriers.

The Java programming language has revived interest in garbage collection. As memory becomes slower relative to microprocessors, garbage collection will only become more expensive. State-of-the-art generational collection is an imperfect solution to this problem. Although the efficacy of type-based collection on a wide range of programs remains to be seen, our implementation demonstrates strong potential.
Bibliography


Appendix A

Example Runtime Class Graphs

Tables A-1, A-3, A-4, and A-1 show runtime class graphs of several benchmarks. These may be useful in developing future type-partitioning algorithms. Each vertex is labelled with the CID of that class. The area of each vertex is proportional to the volume of objects allocated of that class. The color indicates the survival rate of instances of that class. Unless otherwise noted, white means a survival rate of 0-25%, grey means 25-85%, and black means 85-100%.
Figure A-1: Runtime class graph of .202.jess. The diamond represents a class with too many outgoing edges to show. Grey objects have a survival rate of 3-85%.

Figure A-2: Runtime class graph of .228.jack
Figure A-3: Runtime class graph of _209.db

Figure A-4: Runtime class graph of _227.mtrt
Figure A-5: Runtime class graph of _213_javac
Figure A-6: Runtime class graph of _211_anagram