Report Number: WUCS-2007-45

2007-08-01

Exploration of Dynamic Memory

Delvin Curvin Defoe

Since the advent of the Java programming language and the development of real-time garbage collection, Java has become an option for implementing real-time applications. The memory management choices provided by real-time garbage collection allow for real-time eJava developers to spend more of their time implementing real-time solutions. Unfortunately, the real-time community is not convinced that real-time garbage collection works in managing memory for Java applications deployed in a real-time context. Consequently, the Real-Time for Java Expert Group formulated the Real-Time Specification for Java (RTSJ) standards to make Java a real-time programming language. In lieu of garbage collection, the RTSJ... **Read complete abstract on page 2.**

Exploration of Dynamic Memory

Delvin Curvin Defoe

Complete Abstract:

Since the advent of the Java programming language and the development of real-time garbage collection, Java has become an option for implementing real-time applications. The memory management choices provided by real-time garbage collection allow for real-time eJava developers to spend more of their time implementing real-time solutions. Unfortunately, the real-time community is not convinced that real-time garbage collection works in managing memory for Java applications deployed in a real-time context. Consequently, the Real-Time for Java Expert Group formulated the Real-Time Specification for Java (RTSJ) standards to make Java a real-time programming language. In lieu of garbage collection, the RTSJ proposed a new memory model called scopes, and a new type of thread called NoHeapRealTimeThread (NHRT), which takes advantage of scopes. While scopes and NHRTs promise predictable allocation and deallocation behaviors, no asymptotic studies have been conducted to investigate the costs associated with these technologies. To understand the costs associated with using these technologies to manage memory, computations and analyses of time and space overheads associated with scopes and NHRTs are presented. These results provide a framework for comparing the RTSJ's memory management model with real-time garbage collection. Another facet of this research concerns the optimization of novel approaches to garbage collection on multiprocessor systems. Such approaches yield features that are suitable for real-time systems. Although multiprocessor, concurrent garbage collection is not the same as real-time garbage collection, advancements in multiprocessor concurrent garbage collection have demonstrated the feasibility of building low latency multiprocessor real-time garbage collectors. In the nineteen-sixties, only three garbage collection schemes were available, namely reference counting garbage collection, mark-sweep garbage collection, and copying garbage collection. These classical approaches gave new insight into the discipline of memory management and inspired researchers to develop new, more elaborate memory-management techniques. Those insights resulted in a plethora of automatic memory management algorithms and techniques, and a lack of uniformity in the language used to reason about garbage collection. To bring a sense of uniformity to the language used to reason about garbage collection technologies, a taxonomy for comparing garbage collection technologies is presented.

2007-45

Exploration of Dynamic Memory, Doctoral Dissertation

Authors: Delvin C. Defoe

Corresponding Author: dcd2@cse.wustl.edu

Abstract: Since the advent of the Java programming language and the development of real-time garbage collection, Java has become an option for implementing real-time applications. The memory management choices provided by real-time garbage collection allow for real-time Java developers to spend more of their time implementing real-time solutions.

Unfortunately, the real-time community is not convinced that real-time garbage collection works in managing memory for Java applications deployed in a real-time context.
Consequently, the Real-Time for Java Expert Group formulated the Real-Time Specification for Java (RTSJ) standards to make Java a real-time programming language. In lieu of garbage collection, the RTSJ proposed a new memory model called scopes, and a new type of thread called NoHeapRealTimeThread (NHRT), which takes advantage of scopes. While scopes and NHRTs promise predictable allocation and deallocation behaviors, no asymptotic studies have been conducted to investigate the costs associated with these technologies. To understand the costs associated with using these technologies to manage memory,

Type of Report: PhD Dissertation

WASHINGTON UNIVERSITY

Sever Institute
School of Engineering and Applied Science

Department of Computer Science and Engineering

EXPLORATION OF DYNAMIC MEMORY MANAGEMENT SYSTEMS

by

Delvin Curvin Defoe

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

August 2007

Saint Louis, Missouri

Acknowledgments

I would like to express my gratitude to my advisor Dr. Ron Cytron for his guidance, his support, his patience, his wisdom, his tutelage, and his supervision of this research, this dissertation, and my graduate education at Washington University. Dr. Cytron also supported my research financially during my last academic year, took us (the DOC Group) out to lunch on several occasions, and was just resourceful in many dimensions. Thanks much Ron for bearing with me for the last several years.

I would like to especially thank all five members of my dissertation committee, Dr. Ron Cytron (committee chair), Dr. Guy M. Genin, Dr. Christopher D. Gill, Dr. William D. Smart, and Dr. Aaron Stump, for providing me valuable feedback as I pursued this research.

Thanks go out to colleagues like Morgan Deters and Rob LeGrand for collaborations over the years and for their insightful discussions and valuable feedback; to Shakir James for reviewing drafts of this dissertation; to Reynold Bailey and the Bailey family for their support and friendship; to Justin Thiel, Richard Hough, and other members of the Distributed Object Computing Laboratory (DOC Group) at Washington University who assisted in one way or another with this dissertation and contributed meaningfully to my educational experience at Washington University.

Special thanks are in order for Dr. Randy Glean for exposing me to Washington University and the Chancellor's Graduate Fellowship Program. Dr. Glean took a deep interest in my educational career and in my advancement as a scholar and as an individual. Thanks much Dr Glean for the role you played in my development.

I need at this time to acknowledge Dean Robert Thach, Dean Sheri Notaro, Amy Gassel and the Chancellor's Graduate Fellowship Program at Washington University for creating a friendly, welcoming, relaxing environment conducive to excelling. I appreciate the financial support and the way the program accepted me with open arms and made me feel that I belonged. The open house socials, the games nights, the shows at the Fox, the bowling nights, and many other events and activities truly refreshed me and helped much with my stress relief.

To my mom Molly Williams, my granny Aldith Joseph, my dad Morrel Defoe, my sister Marilese Mellow, my brothers Derrickson, Cedrick, Ian, José, and Josiah, I say thank you. Thank you for believing in me and for giving me an opportunity to succeed. Your love and support is noted. Thanks to Annette, Valantine, Delson, and the rest of my family. I am just so thankful for your patience, love, and high expectations. Special thanks to Brother Bob and the rest of the Miligan family. Thanks for taking me in and treating me as your son. Thanks to Anita Phillips, Willie Mae Walker, Alvin T Walker, and the rest of the Walker family for welcoming me and for extending your love and support to me and my family. Thanks to my friends including but not limited to Hon. Vince Henderson, Neehall Philogene, Dr. Munaba Nasiiro, and Nigel Thomas.

Thanks to Dr. Thomas Jackson and my family at the Ferguson Heights Church of Christ. I truly appreciate the spiritual nourishment I receive at the heights. Thanks also to my family at the Massacre Church of Christ. You gave me a great start in my spiritual walk.

I am indebted to my wife Donna for her love, support, prayers, and understanding as I undertook this venture. Her prayers and support have been and continue to be my source of strength. Thanks to Jade for her smile and for becoming part of my life.

Last and by no means least, I thank my God for taking me to this and for taking me through this process. I am forever grateful.

Delvin Curvin Defoe

Washington University in Saint Louis

August 2007

To my beloved wife and family

Contents

List of Figures

ABSTRACT OF THE DISSERTATION

Exploration of Dynamic Memory Management Systems

by

Delvin Curvin Defoe

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2007

Ron K. Cytron, Chairperson

Since the advent of the Java programming language and the development of real-time garbage collection, Java has become an option for implementing real-time applications. The memory management choices provided by real-time garbage collection allow for real-time Java developers to spend more of their time implementing real-time solutions.

Unfortunately, the real-time community is not convinced that real-time garbage collection works in managing memory for Java applications deployed in a real-time context. Consequently, the Real-Time for Java Expert Group formulated the Real-Time Specification for Java (RTSJ) standards to make Java a real-time programming language. In *lieu* of garbage collection, the RTSJ proposed a new memory model called scopes, and a new type of thread called *NoHeapRealTimeThread* (NHRT), which takes advantage of scopes. While scopes and NHRTs promise predictable allocation and deallocation behaviors, no asymptotic studies have been conducted to investigate the costs associated with these technologies. To understand the costs associated with using these technologies to manage memory, computations and analyses of time and space overheads associated with scopes and NHRTs are presented. These results provide a framework for comparing the RTSJ's memory management model with real-time garbage collection.

Another facet of this research concerns the optimization of novel approaches to garbage collection on multiprocessor systems. Such approaches yield features that are suitable for real-time systems. Although multiprocessor, concurrent garbage collection is not the same as real-time garbage collection, advancements in multiprocessor concurrent garbage collection have demonstrated the feasibility of building low latency multiprocessor real-time garbage collectors.

In the nineteen-sixties, only three garbage collection schemes were available, namely *reference counting* garbage collection, *mark-sweep* garbage collection, and *copying garbage* collection. These classical approaches gave new insight into the discipline of memory management and inspired researchers to develop new, more elaborate memory-management techniques. Those insights resulted in a plethora of automatic memory management algorithms and techniques, and a lack of uniformity in the language used to reason about garbage collection. To bring a sense of uniformity to the language used to reason about garbage collection technologies, a taxonomy for comparing garbage collection technologies is presented.

Chapter 1

Introduction

Wirth [51], a Swiss computer scientist (born 1934), developed an imperative programming language suitable for structured programming in 1970. The resulting language, Pascal [51, 21], was named after Blaise Pascal [79], a French mathematician who was a pioneer in computer development history. According to the Pascal Standard ISO 7185 [38], Wirth designed Pascal to satisfy two original goals, namely:

a) "to make available a language suitable for teaching programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language;

b) to define a language whose implementation could be both reliable and efficient on available computers."

But Pascal went far beyond its original design goals to satisfy academic interests. Commercial use of the language often exceeded academic interests.

In 1972 another programming language evolved. The C [55] programming language, designed by Dennis Ritchie [77] at Bell Telephone Laboratory, is a general purpose, procedural, imperative programming language that was developed for the Unix [74] operating system. It has since gained popularity and has been used with many other operating systems. C has also become one of the primary languages for implementing system software.

Some of the design goals for C were that it could be compiled with a simple compiler, provide low-level access to memory, and require little run-time support. As such, C became a versatile language usable for both low-level and high-level implementations. A low-level language like C is easier to read and write than assembly language [78]. This explains its popularity with system software. However, when used for high-level implementations, C is not as easy to understand as other languages. C also suffers from its limited ability to perform automatic checks, *e.g.*, type checking and array bounds checking.

In an attempt to enhance C by addressing the above and other shortcomings, Bjarne Stroustrup [83, 84] of AT&T Bell Laboratories developed the C++ [83, 34, 77] programming language in 1983. C++ is a high-level language with low-level facilities. Some of the features C++ possesses beyond C include classes, virtual functions, operator overloading, single inheritance, multiple inheritance, templates, and exception handling. Some of these features make C++ an *object-oriented programming* [54] language.

Although the programming languages highlighted above differ and were designed with different goals in mind, they show similarities in memory management. While C offers programmers the *malloc()* construct to allocate storage dynamically in the heap, both C++ and Pascal offer the *new()* construct. These constructs allocate a contiguous block of memory in the heap. In addition to using similar constructs to allocate heap storage, these languages offer programmers manual storage reclamation as the only option to reclaim heap storage. Every time C's *malloc()* construct allocate storage, the associated *free()* construct reclaims the storage. Similarly, C++'s *new()* construct is paired with *delete()* and Pascal's *new()* construct is coupled with *dispose()*. Otherwise, the storage persists for the duration of the application, whether or not it can be accessed. Another issue associated with manual storage reclamation is the extreme care that must be taken to ensure that objects are reclaimed only when they are no longer needed by the application. The term *object* is used loosely to refer to any dynamic storage that is allocated in the heap.

Improper reclamation of objects can lead to dangling pointers [2, 35]. Dangling pointers are references to objects that are already deleted. Such pointers surface when objects are reclaimed too early, *i.e.*, objects are deleted while there are still references to them. Programs that create dangling pointers may not necessarily experience problems with small inputs; however, with large or complex inputs such programs tend to crash. Crashes usually occur long after dangling pointers are created, thus making it difficult to trace such pointers. Figure 1.1 depicts code that contains dangling pointers.

```
Foo* p = new Foo();
Foo* q = p;
delete p;
p->DoSomething();   // p is now a dangling pointer!
p = NULL;           // p is no longer dangling
q->ProcessFoo();    // q is also a dangling pointer since q == p!
```

Figure 1.1: C++ source code that contains dangling pointers.

Improper reclamation of objects can also result in memory leaks [48], *i.e.*, a kind of memory consumption that occurs when a program fails to free up memory that it no longer needs. A direct consequence of such memory consumption is diminished system performance, which occurs because of a reduction in the amount of available memory. Such diminished performance can be manifested in the form of fragmentation, memory page faults, and cache misses. A program that leaks memory usually requests more and more memory of the allocator until it eventually crashes due to the allocator's inability to satisfy further requests for memory.

To avert dangling pointer and memory leak problems, library code and packages that offer automatic memory management facilities can be instrumented in the runtime system of the aforementioned programming languages. Alternatively, garbage collected languages can be used for application development.

The discussion above highlights some of the issues that have confronted researchers and programming language developers from as early as the nineteen-fifties. Those issues led to the evolution of automatic memory management. Automatic memory management, including garbage collection [3, 25, 56], became an option for programmers who used the programming language LISP [61] to write their applications. LISP used garbage collection to reclaim lists automatically. Garbage collection (GC) refers to a system's method of automatically reclaiming storage that an application no longer uses. Since its inception, GC has become ubiquitous. Many functional, logical, and object-oriented programming languages use GC techniques to automatically manage their heaps [53]. Some examples include Scheme [33], Dylan [80], ML [75], Haskell [50], Miranda [88], Prolog [91], Smalltalk [76], Eiffel [63], Oberon [73], and Java [5]. Modula-3 [20] offers its programmers the choice of either manual storage reclamation of the heap or the use of GC.

Another memory management option offered by programming languages like the Standard ML Core Language [45] is denoted region-based memory management [87]. Region-based memory management is a memory management scheme that serves as a compromise between the two extremes described above, namely manual memory management and automatic memory management. The notion of region-based memory management is that objects are allocated and deallocated without the presence of a garbage collector. At runtime, all objects are put in *regions* [87], which are pushed onto a stack and popped off the stack when they are no longer needed. The stack of regions, which grows and shrinks, sits in memory and allows memory to be managed using a stack discipline. This memory management scheme is used by the Real-Time Specification for Java (RTSJ) to transform Java into a programming language suitable for real-time application development. In particular, the RTSJ allows Java programmers to instantiate regions of memory, denoted *scoped-memory* areas [15], where dynamic memory allocation occurs. Objects allocated in a scoped-memory area are not collected individually; rather, the entire scoped-memory area is collected at once when no application threads execute in that region. While subsequent sections elaborate more on the use of the RTSJ scoped-memory areas, it is important to note that scoped-memory areas are designed to follow a stack discipline. A scoped-memory area is always collected before its parent scope. This stack discipline restricts the referencing relation among objects. More specifically, it prohibits objects in scoped-memory areas deeper in the stack from referencing objects in scoped-memory areas closer to the top of the stack. The necessity for this restriction will become obvious in subsequent sections.

The rest of this chapter is organized as follows. Section 1.1 supplies details on the RTSJ's region-based memory management scheme; Section 1.2 provides intuition on multi-threaded multiprocessor garbage collection; Section 1.3 discusses the state of the art in classifying garbage collectors; Section 1.4 lists the contributions offered by this dissertation to the memory management community, and Section 1.5 gives a road map for the rest of this dissertation.

1.1 The RTSJ Scoped-memory Areas

Real-time systems require bounded-time memory-management performance. Many real-time applications are written in languages that do not offer garbage-collection capabilities by default; as such, these applications do not suffer from the overhead associated with garbage collection. Since the advent of Java in 1995 [5, 41], however, programmers have been exploring ways to use Java for real-time programming. Java has become attractive because of its automated memory-management capabilities and the ease with which it can be used to write programs.

Java has become ubiquitous in many programming environments but not in real-time and embedded environments. To advance the use of Java for real-time programming, the Real-Time for Java Expert Group (RTJEG) issued the Real-Time Specification for Java [15] (RTSJ) standard. The RTSJ provides extensions to Java that support real-time programming [29] without changing the basic structure of the language. It adds new class libraries and Java Virtual Machine (JVM) extensions to the language, so compilers not optimized for those extensions are not affected by their addition. Although the RTSJ revises many areas of the Java programming language, this dissertation focuses on storage management.

In addition to the heap, where dynamic storage allocation occurs, the RTSJ specifies other memory areas for dynamic storage allocation. Those memory areas are called *immortal memory* and *scoped-memory* areas [15]. Objects allocated in those memory areas are never subjected to garbage collection although the garbage collector may scan immortal memory. Objects in immortal memory are not collected until execution of the program completes. They are never collected earlier, whether or not there are references to them. Objects in a scoped-memory area, on the other hand, are collected *en masse* when every thread that executes in that area exits it. Scoped-memory areas are best used with *NoHeapRealtimeThreads* (NHRTs) [15]. NHRTs are real-time threads that can immediately preempt any garbage collection logic triggered from within the `run()` methods of threads (all other threads have lower priority than NHRTs). They may be allocated in immortal memory, however, they work best with scoped-memory areas.

The use of scoped-memory areas is not without additional cost or burden, as illustrated in this dissertation. Much work has been done with scoped-memory areas and NHRTs; however, to the best of our knowledge, there is no record in the literature of system-independent cost analysis for scoped-memory regions and NHRTs. This dissertation provides a framework for comparing programming with the RTSJ scoped-memory to programming with other memory models, *e.g.*, the heap. While there are several approaches that can be used to do this comparison, we utilize a model that employs asymptotic analysis.

1.2 Incremental and Concurrent Collection

During the past few decades, garbage collection technology has experienced tremendous growth. The state-of-the-art collector is no longer a simple *stop-the-world* collector that executes in a single thread when all *mutators* are suspended. Instead, incremental collectors like Metronome [8, 9, 10] and PERC [68] have evolved. Incremental collectors are collectors that interleave their execution with mutators, thus sharing the central processing unit(s) and other system resources. We use “mutator” interchangeably with “thread” to refer to an application thread, *i.e.*, a thread that does work on behalf of the application. It is possible for a mutator to do work on behalf of the collector, but, in general, the job of a mutator is to perform work on behalf of the application. From the collector’s perspective, a mutator is a nuisance, a thread that mutates (modifies) the heap and increases the work of the collector.

Beyond incrementality, garbage collection technology has experienced growth in the area of concurrency. Concurrent collectors [81, 4, 30, 32, 71] are collectors that execute in parallel with mutators. Concurrent collectors require a multiprocessor environment. Typically, they suspend mutators at the same time at the beginning or end of a collection to compute a unified view of the heap. With such a view they are able to correctly collect objects that become garbage. Although concurrent collection is a significant improvement over the previous collection techniques, it is not without drawbacks. One limitation is scalability, which becomes an issue as the number of mutators increases. Mutators can only be suspended at “safe points” [7, 8, 9, 10, 57]; thus, the duration of suspensions grows linearly with the number of mutators.

An added contribution to garbage collection technology is *on-the-fly* collection. On-the-fly collection is collection in which the collector does not suspend all mutators at the same time, as concurrent collection does. Instead, the collector interacts with mutators on an individual basis. There is never a time when more than one mutator is suspended. While this approach presents an inconsistent view of the heap, the information gathered by the collector during pairs of successive collections is sufficient for the collector to accurately collect garbage objects.

Bacon [7] and Levanoni and Petrank [57] presented on-the-fly collectors. We are particularly interested in the collector presented by Levanoni and Petrank (LPC) [57] because it is a high performance reference counting garbage collector which offers short pause times and low synchronization. LPC denotes Levanoni and Petrank's collector, an on-the-fly reference counting garbage collector for Java. Details on LPC are presented in Chapter 2.

Although LPC is a high performance collector, one of its shortcomings is overhead. LPC suspends all mutators 4 times during a collection to perform transactions with them. The duration of a transaction is not fixed, but varies by mutator and collection. After each transaction the collector resumes the suspended mutator so it can proceed with its computation. The process of suspending a mutator, transacting with it, and resuming it is called a *handshake*. Each mutator experiences 4 handshakes during a collection. Subjecting mutators to that many handshakes can slow down the application and have other adverse effects outlined in Chapter 2. We address these issues by redesigning the collector to minimize the number of handshakes to reduce overhead.

1.3 Classifying Garbage Collectors

The literature on garbage collection is extensive [52, 93, 94]. What is noteworthy, however, is that most of the contributions offered by the garbage collection community involve the presentation of a new collector, the evaluation of an existing collector, the comparison of at least two extant collectors, or the summarization of existing collectors. Very little contribution has been made toward *taxonomizing* garbage collection technology. We address this limitation by developing a garbage collection taxonomy called *GC-Tax*. GC-Tax identifies and formalizes a list of relevant garbage collection features that address specific issues concerning the creation of enabling environments for application execution. We utilize GC-Tax to classify a cross-section of extant garbage collection techniques to gain insight into how different technologies compare. The idea is that GC-Tax will enable application developers to select the most appropriate collectors for their applications. In order to take advantage of GC-Tax, developers need to determine the most important characteristics of their applications. Using that information, they can select the most suitable collectors to manage memory on behalf of their applications.

1.4 Contributions

The contributions that this dissertation offers to the memory management community begin in Chapter 3. Before the Real-Time Specification for Java scoped-memory model can be compared with other memory models, in the RTSJ's attempt to present Java as a programming language suitable for real-time, quantitative analysis needs to be done on the RTSJ scoped-memory model. We provide a method for determining asymptotic bounds for the RTSJ scoped memory model.

This dissertation also contributes to the aforementioned community by presenting a high performance, multiprocessor, reference counting garbage collector that offers high throughput, negligible pause times, and low synchronization in its write barrier. This collector is based on previous work by Levanoni and Petrank [57, 58].

Further, in an effort to unify garbage collection technology this dissertation provides a taxonomy of garbage collectors. The goal is that this taxonomy does not only classify garbage collection technology, but that it also offers developers a tool to determine which garbage collectors are best suited for their applications.

1.5 Road Map

The rest of this dissertation is organized as follows. Chapter 2 provides essential background on the evolution of garbage collection techniques and highlights the RTSJ's influence on memory management. Chapter 3 provides a model to determine asymptotic bounds for the RTSJ scoped-memory model. Chapter 4 presents an improved on-the-fly reference counting garbage collector for Java. Chapter 5 summarizes the contributions made in the field of garbage collection, motivates the need for a taxonomy that classifies garbage collectors, and presents a taxonomy of garbage collectors. Chapter 6 summarizes the contributions of this dissertation to the memory management community and lists ideas for future consideration.

Chapter 2

Background

We provide background on the evolution of garbage collection in this chapter by visiting its history and highlighting the contributions of a few pioneers in the garbage collection community. We then discuss a handful of modern approaches to emphasize the growth and impact of the garbage collection community on memory management systems. Finally, we highlight the influence of the Real-Time Specification for Java on memory management. These discussions motivate the subject of this dissertation.

2.1 Classical Garbage Collection Techniques

At a high level, this dissertation concerns exploration of automatic memory management systems. The study of automatic memory management systems is not new, but dates back to the nineteen-sixties when pioneers like McCarthy [61], Collins [26], and Minsky [64] designed garbage collectors to address the automatic erasure of lists in LISP. McCarthy designed the *mark-sweep* or *mark-scan* collector; Collins designed the *reference counting* collector, and Minsky designed the *copying* collector. We give a brief overview of these collectors without necessarily listing their advantages or disadvantages since we are more concerned with giving intuition into their functionality.

2.1.1 Mark-Sweep Garbage Collection

In the late nineteen-fifties, when LISP was the prevailing programming language, programmers were required to handle erasure of lists explicitly using a built-in operator called *eralis* [62]. A method for automatic erasure of lists was needed, so the mark-sweep garbage collector was developed in 1960. That effort was pioneered by McCarthy who published it in April of that year [61].

Under the mark-sweep garbage collection scheme, objects or *cells* are not reclaimed as soon as they become garbage, but remain dead until the storage pool is exhausted. If a new cell is requested and the storage pool is exhausted, the mutator's computation is suspended until all dead cells in the heap are swept and returned to the pool of free cells. The garbage collection routine reclaims garbage by traversing (tracing) the graph of all live objects and returning to the pool of free cells all cells that are not live. This forward trace begins from the *root set* and marks every object reachable as live. Every other object is dead and is returned to the pool of free cells. If the collection routine is successful in reclaiming sufficient storage, the mutator's request is satisfied and computation resumes. Otherwise, an error condition is reported by the collector.

McCarthy's mark-sweep collector is a typical example of a tracing collector since it traverses or traces live data in the heap. Another example of tracing collectors is the copying collector of Fenichel and Yochelson [36] described in Section 2.1.3.

2.1.2 Reference Counting Garbage Collection

The classical reference counting garbage collector was originally developed for LISP by George Collins [26, 53]. Collin's collector is a simple, direct method for reclaiming garbage. It uses the count of references to heap objects from the root set and from all live objects. Each object keeps count of the number of references to it in a special field called its *reference count*. The reference count of each object is equal to the number of references that point to it from the root set and from all live objects. Objects are collected when they have a reference count of zero.

When a new object *Obj*, for example, is allocated from the pool of free memory, it has an initial reference count of zero. However, before it is returned to the application, its reference count is set to one. Each time thereafter a pointer references *Obj*, its reference count is incremented. Each time a pointer to *Obj* is deleted, its reference count is decremented. If that causes *Obj*'s reference count to drop to zero, no object or field points to it; as such, *Obj* cannot be reached by the mutator. *Obj* is thus garbage and should be returned to the pool of free storage. Before *Obj* is returned to that pool, however, the reference count of every object to which it points must be decremented. This decrement is necessary because an object's *liveness* is determined only by reference count contributions from live objects. Reclaiming one object can potentially lead to reclaiming multiple objects, a process known as *recursive freeing* [53].

2.1.3 Copying Garbage Collection

In 1963 Minsky published the first copying garbage collector for LISP 1.5 [64]. Jones and Lins [53] give a brief overview of Minsky's collector. The copying collector presented here however, was devised by Fenichel and Yochelson in 1969 [36]. Fenichel and Yochelson's collector divides the heap equally into *semi-spaces*; at any time one is called the *FromSpace* and the other is called the *ToSpace*. A collection is initiated when there is not enough free storage in ToSpace for the mutator to allocate an object. The mutator is thus suspended.

A collection begins with the collector flipping the role of the semi-spaces; FromSpace becomes ToSpace and ToSpace becomes FromSpace. The collector then traverses the live data in FromSpace, starting with references from the roots, and copies each object to ToSpace when it is first visited. After all the live objects in FromSpace have been copied to ToSpace, the data in ToSpace is a compacted replica of the data from FromSpace. FromSpace becomes garbage and is recycled. The mutator resumes execution if enough free storage is recovered in ToSpace to satisfy the allocation request that initiated the collection. Subsequent allocation requests are satisfied in ToSpace.

2.2 Modern Garbage Collection techniques

The classical garbage collection strategies serve as the basis for the design of modern garbage collection techniques. A plethora of modern garbage collection techniques exists in the literature; however, we give a brief overview of the generational garbage collection technique [44] and Levanoni and Petrank's on-the-fly referencing counting garbage collection scheme [57].

2.2.1 Generational Garbage Collection

Generational garbage collection is based on the *weak generational hypothesis* "most objects die young" [89, 44]. This hypothesis led to the generational strategy, which separates objects by age into a minimum of two regions of the heap called *generations* [59]. Newly allocated objects are placed in one generation of the heap and are promoted to other generations if they survive collection of their allocated generation. Since the allocated region contains newly created objects, it is generally referred to as the *new generation* or the *youngest generation*. Objects in the new generation are expected to have short lifetimes; thus, the new generation is collected relatively frequently.

Objects that survive collection of the new generation are promoted to other regions of the heap referred to as *older generations*. Older generations are not collected as regularly as the new generation due to the *strong generational hypothesis* [47], which suggests that the longer an object lives the less likely it is to die. Note: the youngest generation can be collected independently of the older generations but not *vice versa* since most inter-generational pointers are from younger generations to older generations. This mandates that younger generations be collected when an older generation is collected.

The techniques used to collect the different generations may vary. Thus, copying, mark-sweep, reference counting, or a combination of these techniques can be used to collect the various generations. Collection of the youngest generation is generally called a minor collection while collection of the older generations is usually referred to as a major collection. The number of generations also varies, but the use of two generations is popular in the literature. Jones's garbage collection book [53] provides more details on generational garbage collectors.

2.2.2 On-the-Fly Garbage Collection

The on-the-fly collector (see Section 1.2) described in this section was designed by Levanoni and Petrank [57] and for convenience, it is referred to here as LPC. LPC is an on-the-fly reference counting garbage collector for Java[1] with low synchronization in its write barrier. The ideal target for LPC is a multi-thread multiprocessor system that runs N mutators on $N + 1$ processors. Each mutator executes on a dedicated processor. The extra processor is reserved for the collector. The collector executes a series of collections in *cycles*. Each collection consists of four lightweight synchronization points called *soft handshakes* or *handshakes* for short. A handshake is a synchronization point between the collector and all mutators where the collector transacts with each mutator on an individual basis. A transaction entails the suspension of a mutator by the collector, the retrieval of the mutator's buffered information, and the resumption of the mutator. There is never an occasion when the collector suspends more than one mutators at the same time. This is an overhead reduction improvement over previous collectors that suspend all mutators at the same time.

For the correctness of LPC, there are two instances in a mutator's life when it cannot be suspended by the collector: when it is executing in the write barrier (updating a reference field in an object) and when it is instantiating a new object. We refer to the code segments that perform these functionalities as *collector-proof code* or *CP-code*. CP-code is code the

[1]They designed their collector for Java but their approach is suitable for any language where pointers can be distinguished.

collector is not allowed to preempt. The collector can preempt a mutator only when it is not executing CP-code.

LPC requires four handshakes per collection denoted HS_m (handshake m) where $1 \leq m \leq 4$. LPC also requires a sequentially consistent memory model since reads and writes in the write barrier must be executed in the order they appear. See Figure 2.1 for the LPC write barrier. In Chapter 4 we present a similar collector; however, our collector minimizes the number of handshakes and reduces the effect of each handshake on mutator execution.

```
Procedure **Update**(*s:* **Slot**, *new:* **Object**)
begin
1.  Object $old$ := **read**($s$)
2.  if $\neg Dirty(s)$ then
3.     $Buf_i[CurrPos_i] := \langle s, old \rangle$
4.     $CurrPos_i := CurrPos_i + 1$
5.     $Dirty(s) :=$ **true**
6.  **write**($s$, $new$)
7.  if $Snoop_i$ then
8.     $Locals_i := Locals_i \cup \{new\}$
end
```

Figure 2.1: The Levanoni and Petrank write barrier [57] where i is the mutator index.

When using the write barrier in Figure 2.1, each pointer s in an object can reference any object. Further, each pointer s is associated with a *dirty* flag that is set when s is first written to during a collection. Each mutator T_i is equipped with a local buffer in which it stores $\langle s, old \rangle$ tuples for the collector's use. In the $\langle s, old \rangle$ tuple s denotes a pointer that receives an assignment and old represents the address of the last object that s referenced during the previous collection. T_i buffers $\langle s, old \rangle$ only if it is the first mutator to assign to s during a collection. Each mutator T_i is also equipped with a *snoop* flag $Snoop_i$ that indicates whether T_i is involved in the computation of a *sliding view* [57, 58]. A sliding view is an inconsistent or inexact view of the heap computed during a collection by the collector when it transacts with the mutators on an individual basis. During the computation of a sliding view, objects pointed to are stored in a mutator's local state to ensure that they are not prematurely collected at the end of the collection. Further discussion on sliding view computation and on the operation of the collector is provided in Chapter 4.

2.3 Real-Time Garbage Collection Techniques

Baker [12] began his seminal paper on real-time garbage collection with the following definition:

> A real-time list processing system is one in which the time required by the elementary list operations . . . is bounded by a (small) constant.

A list processing system, according to Baker, is a system that collects garbage. Hence, a real-time garbage collector is a collector that performs garbage collection work in bounded time. Many real-time garbage collectors exist [4, 13, 17, 22, 97] in the literature; however, here we give an overview of Metronome [9] and PERC [68], for the sake of discussion.

2.3.1 Metronome

Metronome [8, 9, 10] is an incremental, but non-parallel garbage collector that targets a uniprocessor, embedded environment. Since Metronome is not parallel, it must be interleaved with the mutator(s), instead of running on a separate processor. The interleaving in Metronome is controlled explicitly.

Many *hybrid* collectors exist in the literature and Metronome is one of them. As long as usable storage is available Metronome executes as a non-copying, incremental mark-sweep collector, but when storage becomes scarce, it defragments the heap with a limited, incremental copying collector. Metronome can thus be characterized as an incremental, mark-sweep collector that utilizes a limited, incremental copying collector to defragment the heap.

Metronome uses a "snapshot-at-the-beginning" [97] algorithm that allocates objects marked. Metronome also uses segregated free lists so that memory is divided into fixed sized pages (*e.g.*, 16 KB). Each page is further divided into blocks of a particular size class, usually a power of 2. Objects are allocated from the smallest size class that is able to satisfy their allocation requests of say s bytes. The next size block is $s(1+\rho)$ where ρ is most likely the fraction 1/8, which yields a worst case fragmentation of 12.5%.

If a page becomes fragmented because of garbage collection, its objects are moved to a mostly full page using the incremental copying collector. Relocation of objects is achieved by using a forwarding pointer located in the header of each object. A read barrier is used to maintain the ToSpace invariant: mutators always see objects in ToSpace. While the above is true for the incremental copying collector, collection is dominated by the incremental mark-sweep collector, which is similar to Yuasa's [97] "snapshot-at-the-beginning" algorithm.

One additional feature of Metronome that should be noted is that Metronome breaks large arrays into fixed size pieces, called *arraylets*, to bound the work of scanning or copying an array. This feature was also added to limit external fragmentation caused by large objects. Further, Metronome exhibits the following features when it performs at its best.

1. Mutator interval = 6ms
2. Collector interval = 6ms
3. Pause time = 6ms
4. Minimum mutator utilization of the CPU = 50 %

2.3.2 PERC

PERC [68] is an incremental garbage collector that divides its work into thousands of small *uninterruptible* increments of work. Depending on the choice of CPU, in practice the maximum time required to execute an increment of work can be approximated to about 100 microseconds. Mutators with priorities higher than the collector may preempt the collector. However, when the collector resumes following preemption, it continues to execute where it left off.

Garbage collection consists of dividing memory into a number of equal-sized regions and selecting one region to serve as FromSpace and another to serve as ToSpace. These regions are defragmented using an incremental copying collector. The other regions are subsequently reclaimed using an incremental mark-sweep collector that does not relocate live objects.

During the incremental copying stage of garbage collection, access to an object, *obj*, is to the single valid copy of *obj*, whether it is located in FromSpace or ToSpace. Should there exist an invalid copy of *obj* in ToSpace or FromSpace, it will contain a pointer to the valid copy. Each object waiting to be relocated has a forwarding pointer to the memory that is reserved for the eventual copy. At the conclusion of the incremental copying stage of garbage collection the free space in FromSpace is coalesced and ToSpace is compacted. This is accomplished by incrementally reserving space in ToSpace and subsequently relocating live objects from FromSpace to ToSpace. Incremental copying guarantees 50% memory utilization.

When the copying stage of garbage collection completes, the mark-sweep stage begins. If a mutator preempts the collector during the mark phase of incremental mark-sweep collection, it may rearrange the relationship between objects before relinquishing to the collector. To remedy that situation, the mutator uses a write barrier to mark the referenced object every time a pointer is overwritten. One of the downsides of the incremental mark-sweep collector is that its memory utilization is inconsistent–it varies from high to low.

2.4 The RTSJ's Influence on Memory Management

One of the most prominent features of the RTSJ is its new memory management model based on *scoped memory areas* (or *scopes* for short) [72]. This new memory model assures programmers of timely reclamation of memory and predictable performance. This comes at the cost of an unfamiliar programming model—a restrictive model that relies on the use of scopes. These new scopes were designed to meet two very important requirements [72]: to provide predictable allocation and deallocation performance, and to ensure that real-time threads do not block when memory is reclaimed by the Java Virtual Machine (JVM).

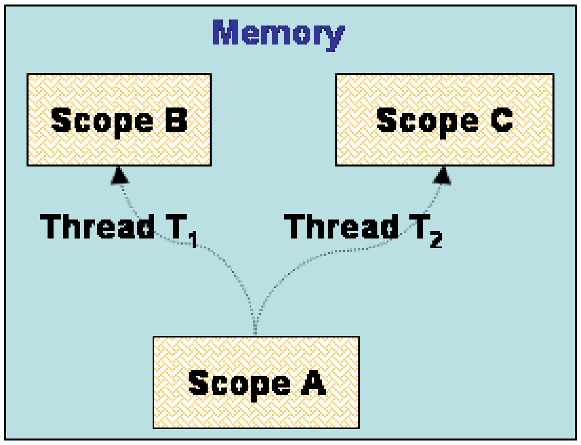

Figure 2.2: Scoped-memory single-parent rule. A is the parent of both B and C.

To meet these requirements, the RTSJ ensures that objects in a scope are not deallocated individually. Instead, the entire scope is collected *en masse* when all threads within the scope exit it. A scope is a pool of memory from which objects are allocated. Each scope can be entered by multiple threads. These threads can allocate objects in the memory pool and communicate with each other by shared variables. A new scope can also be instantiated by a thread executing within its current scope. This is known as the nesting of scopes. Such nesting, however, is controlled by the order of threads entering the scopes—see Figure 2.2. A scope can become the parent of multiple scopes but no scope is allowed to have multiple parents. This restriction is called the *single-parent rule*. To take advantage of scopes, the RTSJ defined a new type of thread called *NoHeapRealtimeThread* (NHRT). NHRTs cannot allocate objects in the garbage-collected heap and they cannot reference objects in the heap. These constraints were added to prevent NHRTs from experiencing unbounded delay due to the locking of heap objects during garbage collection [29]. NHRTs have the highest priority among all threads and can preempt even the garbage collector.

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes*

Figure 2.3: References between storage areas [15]. * If an object is in the same scope or the outer scope.

Figure 2.3 details which objects in certain memory areas are allowed to reference objects in other memory areas. These constraints do not apply to objects only, but also to threads so that real-time threads do not block when the JVM reclaims objects.

Chapter 3

Asymptotic Analysis of the RTSJ scoped memory areas

The **Real-time Specification for Java**^{TM} (RTSJ) covers many issues related to real-time programming. However, in this dissertation we are only concerned with storage, threads, and their relationships to memory management. In particular, we are concerned with computing bounds for the costs associated with using the RTSJ scoped-memory areas, and the RTSJ's **NoHeapRealtimeThread** (NHRT).

3.1 Chapter road map

The rest of this chapter is organized as follows. Section 3.2 provides background and motivation for our work. Sections 3.3 and 3.6 describe our approaches for performing scoped-memory analysis. Sections 3.4, 3.5, 3.7, and 3.8 present scoped-memory analysis for selected abstract data types.

3.2 Background and Motivation

Chapters 1 and 2 gave an overview of the RTSJ scoped-memory areas and NHRTs. Those chapters also motivated the research on scoped-memory areas presented in this chapter. Additionally, they noted the limitations of garbage collection and described the improvements offered by the RTSJ scoped-memory areas and NHRTs. This section recaps some of these highlights.

Garbage collection occurs at unpredictable times with unbounded latency. Consequently, the time required to allocate a new object in the heap is unbounded. Researchers have proposed real-time garbage collection [65, 23] as a way to bound object allocation; however, it is still questionable [66] whether a collector and allocator can always provide storage in bounded time [29]. Consider the case, for example, where an object needs to be allocated and the heap is exhausted, except for a few dead objects. To reclaim storage to satisfy the allocation, a garbage collector needs to run. An exact collector, like mark-sweep, would require a marking phase to discover all live objects. Such a phase is unbounded. Other collectors can limit the extent of a marking phase, but at the cost of potentially skipping over garbage objects. The result is that the cost of allocating a new object cannot be reasonably bounded if a garbage collector needs to run to free storage to satisfy the allocation request.

To overcome these shortcomings, the Real-Time for Java Expert Group (RTJEG) proposed the Real-Time Specification for Java [15] (RTSJ) standard, which provides extensions to Java in support of real-time programming [29]. These extensions include scoped-memory areas and NHRTs. Recall from Section 2.4 that scoped-memory areas were designed to meet two very important requirements [72], namely:

1. to provide predictable allocation and deallocation performance, and

2. to ensure that real-time threads do not block when memory is reclaimed by the virtual machine.

These requirements are met by ensuring that objects in a scoped-memory area are not collected individually. Instead, the entire scoped-memory area is collected *en masse* when no threads execute in it.

3.3 Scoped-memory Analysis

While there are several open problems associated with the RTSJ scoped-memory areas, this dissertation focuses on computing bounds for the RTSJ scoped-memory model when NHRTs are used. These bounds give an idea of how expensive it is to execute applications in a scoped-memory environment. They also facilitate comparison of execution in scoped-memory environments with execution in other memory environments (*e.g.*, the heap).

3.3.1 Empirical analysis of scoped memory

One approach to performing the comparisons described above employs empirical methods. These methods measure the execution time for an application with a given input size in a scoped-memory environment. They also measure the execution time for an equivalent application with the same input size in a different memory environment. The execution times are then compared to determine which memory environment is more appropriate for the application. Although this approach produces results that are important for certain problems, one observation is that the results are both implementation-dependent and system-dependent. Should the same applications be implemented using different programming paradigms (*e.g.*, imperative programming, procedural programming, object-oriented programming, or functional programming), the results may differ. Should the same applications be executed on systems with different resource allocations (*e.g.*, a faster processor, faster memory, more memory, multiple processors, etc.), the results could also differ.

3.3.2 Asymptotic analysis of scoped memory

To address the limitations of the empirical methods, asymptotic methods are used to compute the cost of using the RTSJ scoped-memory model with NHRTs. This approach gives bounds that are both implementation-independent and system-independent. To the best of our knowledge, there is no record in the literature of system-independent cost analysis for scoped-memory regions and NHRTs that preceded our paper [28].

We present a model for computing asymptotic bounds for the RTSJ scoped-memory regions and NHRTs that follows the steps listed below.

1. Select an abstract data type (ADT) that can hold an arbitrary number of elements.
2. Define the fundamental operations of the ADT and provide an interface.
3. Propose at least one implementation for each operation.
4. Adopt methods from Cormen *et al.* [27] to compute the worst-case running time for each operation.
5. Perform step 4 for an intermixed sequence of n operations.
6. Repeat from step 1.

We define an abstract data type or ADT as a set of legal data values and a number of primitive operations that can be performed on these values [18]. Such a data type is abstract in the sense that the focus is not on its implementation since implementation is subject to change. The actual implementation is not defined and does not affect the use of the ADT. Instead, an ADT is represented by an interface that hides the underlying implementation. Users of an ADT are very concerned about its interface. The reader is referred to Weiss's book on data structures [92] for more information on ADTs.

For the ADTs we consider, the input size n for each operation is characterized by the number of elements in the data set immediately before the operation is run. Should there be a need to use a different characterization for the input size of an operation, one will be provided. The pseudocode provided for selected operations follows conventions from Cormen *et al.* [27].

We utilize this model to solve the problem at hand. This study is vital to both the real-time community and the software engineering community. It empowers these communities by presenting to them a means of deciding which memory model is most appropriate for their applications. Moreover, it allows us to reason more completely about different memory models. We use this model with the *stack* ADT and the *queue* ADT in Section 3.4 and Section 3.5 respectively.

3.4 Stack analysis

We present a scoped-memory implementation of the stack abstract data type and analyze its running time. The stack is an ADT that operates on the *Last-In-First-Out* (LIFO) principle. The idea is that the last element pushed on a stack is the first element popped off the stack. A common use of a stack is found at a buffet restaurant. When a family goes to the serving line, plates are usually stacked one on top of another. Before the second plate atop the stack can be retrieved, the topmost plate must first be retrieved. In Computer Science, the notion of a stack is also used in expression evaluation, syntax parsing, and in solving search problems. The end of the stack where an item is added to or retrieved from the stack is called the *top* of the stack. The stack is also associated with a size component that keeps count of the number of items or elements on the stack. The fundamental operations of the stack ADT are as follows:

1. IS-EMPTY(S) - an operation that returns the binary value **TRUE** if stack S is empty, and **FALSE** otherwise.

2. PUSH(S, x) - an operation that puts element x onto the top of stack S.

3. POP(S) - an operation that removes the element at the top of stack S and returns it. If the stack is empty the special pointer value $NULL$[1] is returned.

The POP operation does not take an element as a parameter because the element popped off the stack is always the topmost element of the stack. The top of a stack is the end where its fundamental operations are performed. Whereas the PUSH operation increases the stack size by one element, the POP operation decreases the stack size by one element. The empty stack has a size of zero elements. We now provide analysis for the stack ADT.

We are most concerned about implementing and analyzing stacks in scoped memory. However, we first present an implementation with analysis in heap memory for the sake of comparison.

3.4.1 Typical implementation of stack

Several data structures, including the array and the singly linked list, can be used to implement a stack in the heap. We discuss these implementations in the subsections below.

Array implementation

The array data structure is sometimes used to implement a stack in the heap; however, the resulting stack is a bounded stack, which does not conform to the definition of the stack ADT for the following reasons. A size n must be specified for the stack at creation time since an array by definition has a maximum size. If an element is to be pushed on a stack with n elements, the PUSH operation will fail. This conflicts with the definition of stack. To overcome these constraints, the array must be allowed to grow in size. The resulting data structure would no longer be an array, but an arraylist. For these reasons we do not consider an array data structure to be suitable for implementing a stack, or any of the other ADTs we consider in our studies.

[1]$NULL$ is used to signify that a pointer has no target.

Singly linked list implementation

A data structure commonly used to implement the stack ADT in the heap is the singly linked list. With such implementation the PUSH and POP operations in Figure 3.2 and Figure 3.3, respectively, are done at the *front* of the list. The IS-EMPTY operation in Figure 3.1 is also performed at the front of the list. Consequently, the front of the list represents the top of the stack it implements. Pseudocode and computation of the worst-case running times for the fundamental operations of the stack ADT are presented. Let $T(n)$ denote the worst-case running time for a problem of size n. Recall the definition of n given in Section 3.3.2.

The IS-EMPTY operation—singly linked list implementation

```
IS-EMTY($S$)                               line   cost    times
1 return $S[top] = NULL$                     1    $c_1$     1
```

Figure 3.1: Procedure to determine whether the stack is empty—linked list implementation.

The line numbers of IS-EMPTY, the cost of executing each line, and the number of times each line executes are given above. The worst-case running time for IS-EMPTY $T(n)$ is thus given as:

$$\begin{aligned} T(n) &= c_1 * 1 \\ &= c_1 \\ &= O(1) \end{aligned}$$

The PUSH operation—singly linked list implementation

The matrix in Figure 3.2 gives the line numbers of the PUSH operation, the cost of executing each line, and the number of times each line executes. This matrix provides enough information to compute the worst-case running time for the PUSH operation. Notice that

```
PUSH($S, x$)                          line  cost   times
1  $x[Next] \leftarrow S[top]$          1    $c_1$    1
2  $S[top] \leftarrow x$                2    $c_2$    1
```

Figure 3.2: Procedure to push an element on the stack—linked list implementation.

like the IS-EMPTY operation, the PUSH operation is also a constant time operation.

$$\begin{aligned} T(n) &= c_1 * 1 + c_2 * 1 \\ &= c_1 + c_2 \\ &= O(1) \end{aligned}$$

The POP operation—singly linked list implementation

```
POP($S$)                                      line  cost   times
1  $x \leftarrow S[top]$                        1    $c_1$    1
2  **if** !IS-EMPTY(S)                          2    $c_2$    1
3     **then** $S[top] \leftarrow S[top[Next]]$  3    $c_3$    1
4  **return** $x$                               4    $c_4$    1
```

Figure 3.3: Procedure to pop the topmost element off the stack—linked list implementation.

The POP operation assigns local variable x the value $S[top]$ then checks for emptiness. If the stack is not empty then its top is set to the next element (from the top) in the stack. Either the special value $NULL$ (if the stack was empty) or the previous topmost element of the stack is then returned. From the matrix in Figure 3.3 the worst-case running time for the POP operation reduces to $T(n) = max(T_1(n), T_2(n))$ where

$$\begin{aligned} T_1(n) &= c_1 * 1 + c_2 * 1 + c_4 * 1 \\ &= c_1 + c_2 + c_4 \\ &= O(1) \\ T_2(n) &= c_1 * 1 + c_2 * 1 + c_3 * 1 + c_4 * 1 \\ &= c_1 + c_2 + c_3 + c_4 \\ &= O(1) \end{aligned}$$

Although $T_1(n) \leq T_2(n)$, $T_1(n)$ and $T_2(n)$ are both $O(1)$ running times; thus, $T(n)$ reduces to $T_2(n) = O(1)$. Notice that a constant time value is used as the cost for checking emptiness (line 2). This is because the worst-case running time for IS-EMPTY is $O(1)$. This and the previous analyses lead to the conclusion that for a singly linked list implementation of the stack ADT, each operation executes in constant time.

3.4.2 Scoped-memory implementation of stack

For a scoped-memory implementation of a stack we make the following assumptions:

1. Each application A that manages a stack S is fully compliant with the RTSJ.
2. A has a single thread T_a, which is an instance of the RTSJ's NHRT. The RTSJ allows multiple threads to share data structures as long as all threads enter scopes in the same order. This simplifying assumption of a single thread is made only for easy exposition and analysis.
3. A executes on an RTSJ-compliant JVM.
4. T_a can legally access S and the elements managed by S.
5. Before an element x is pushed on stack S, a new scope s must first be instantiated to store x, and T_a must enter s.

Assumption 5 is relevant for the purpose of complexity analysis. Although we do not suggest one scope per-element in practice, here we are concerned about worst-case analysis. Thus, it is essential that we consider the worst possible scenario for stack operations that utilize the RTSJ scoped-memory model.

The IS-EMPTY operation—scoped-memory implementation

IS-EMPTY(S)	*line*	*time cost*	*frequency*
1 **return** $TOS = S$	1	c_1	1

Figure 3.4: Procedure to determine whether the stack is empty—scoped-memory implementation.

We assume there is a TOS field in the current scope that points to the top-of-stack element, which is either in the current scope or is accessible from the current scope. If TOS points to the stack object S (a sentinel used for indicating the empty stack), then the application thread T_a is executing in the scope containing S. Thus, S contains no elements, so the stack is empty. If c_1 is the time required to execute line 1 of IS-EMPTY, then the worst-case running time for IS-EMPTY is $T(n) = O(1)$.

The PUSH operation—scoped-memory implementation

PUSH(S, x)	*line*	*time cost*	*frequency*
1 $sm \leftarrow new\ ScopedMemory(m)$	1	c_1	1
2 $enter(sm, T_a)$	2	c_2	1
3 $TOS \leftarrow x$	3	c_3	1

Figure 3.5: Procedure to push an element on the stack—scoped-memory implementation. $m \geq |x| + |TOS|$.

The PUSH operation depicted in Figure 3.5 is equivalent to the following sequence of basic operations performed by the application thread T_a. From the current scope T_a instantiates a new scope sm of size m bytes. T_a enters sm then sets the TOS field in sm to point to element x. Assuming each line i in PUSH requires c_i time for execution, the worst case execution time for PUSH is

$$\begin{aligned} T(n) &= c_1 * 1 + c_2 * 1 + c_3 * 1 \\ &= c_1 + c_2 + c_3 \\ &= O(1) \end{aligned}$$

The correctness of this result is based on the fact that each line is executed once per invocation. Because a scope has limited lifetime dictated by the reference count of threads that execute in it, T_a is not allowed to exit sm. To ensure that T_a keeps sm alive T_a does not return from the $enter()$ method of line 2 of Figure 3.5. Should T_a return from the $enter()$ method, the thread reference-count of sm would drop to zero, sm would be collected, and the PUSH operation would fail.

The POP operation—scoped-memory implementation

```
POP($S$)
1 **if** IS-EMPTY(S)
2    **then** $x \leftarrow NULL$
3    **else** $x \leftarrow TOS$
4 **return** $x$
```

line	*time cost*	*frequency*
1	c_1	1
2	c_2	1
3	c_3	1
4	c_4	1

Figure 3.6: Procedure to pop the topmost element off the stack—scoped-memory implementation.

The POP operation is one of the simplest operations for a scoped-memory implementation of stack. POP simply returns the TOS element if one exists, $NULL$ otherwise. Assuming each line i of the POP operation (Figure 3.6) requires c_i time to execute, the worst-case execution time for the POP operation is given as $T(n) = max(T_1(n), T_2(n))$

where

$$\begin{aligned}
T_1(n) &= c_1 * 1 + c_2 * 1 + c_4 * 1 \\
&= c_1 + c_2 + c_4 \\
&= O(1) \\
T_2(n) &= c_1 * 1 + c_3 * 1 + c_4 * 1 \\
&= c_1 + c_3 + c_4 \\
&= O(1)
\end{aligned}$$

Since $T_1(n)$ and $T_2(n)$ are both $O(1)$ worst-case running times, it follows that $T(n) = O(1)$. After popping the stack, T_a must return from the *enter*() method of line 2 of Figure 3.5. We assume for all practical purposes that returning from the *enter*() method takes constant time so the worst-case execution time of the POP operation remains $O(1)$. The new top-of-stack becomes the TOS element of the parent scope[2].

3.4.3 Cumulative analysis for stack

Here we consider an intermixed sequence of n PUSH and POP operations on a stack instance. We analyze this sequence of operations for a singly linked list implementation and a scoped-memory implementation of stack. We let n denote the total number of operations and let m denote the number of PUSH operations. The number of POP operations is thus given by $n - m$ where $n - m \le m \le n$. The worst-case running time for the singly linked

[2]The parent of the scope that contained the popped element becomes the new current scope for T_a.

list implementation of the intermixed sequence of operations is computed as

$$\begin{aligned}
T(n) &= T_{push}(m) + T_{pop}(n-m) \\
&= m * c_1 + (n-m) * c_2 \\
&= mc_1 + nc_2 - mc_2 \\
&= nc_2 + m(c_1 - c_2)n \\
&= O(n)
\end{aligned}$$

For a scoped-memory implementation the running time for PUSH is $O(1)$ and the running time for POP is also $O(1)$. Thus, the running time for the intermixed sequence of operations in the context of a scoped-memory implementation is given as $T(n) = O(n)$.

3.4.4 Discussion

The singly linked list implementation presented above has a $T(n) = O(1)$ worst-case execution time for each stack operation. The scoped-memory implementation also has $T(n) = O(1)$ as its worst-case execution time for each operation. The problem of running an intermixed sequence of n PUSH and POP operations has a running time of $T(n) = O(n)$ in each context, as expected. Given a particular program that uses a stack, the programmer can thus choose among stack implementations. However, the following are some concerns to bear in mind.

Although a singly linked list implementation works well in the heap, pointer manipulation can affect the proportionality constants of the running time for each operation. Garbage collection can also interfere with the running times of stack operations if the application executes in a heap that is subject to garbage collection.

A scoped-memory implementation, while ideal for real-time environments, comes at the cost of learning a new, more restrictive programming model. Real-time programmers, however, can benefit from the timing guarantees offered by the RTSJ. Recall that garbage collection cannot interrupt a NHRT since NHRTs possess higher priorities than the collector.

3.5 Queue analysis

Our scoped-memory implementation of the queue abstract data type uses an approach similar to Okasaki's [69] functional-language implementation in that a queue is simulated as a pair of stacks. The *queue* ADT operates on the *First-In-First-Out* (FIFO) principle, *i.e.*, the first element added to the queue is the first element removed from the queue. This is equivalent to the requirement that whenever an element is added to the queue, all elements that were already in the queue must first be removed before the new element can be removed. A common application of queue is seen at most banks or financial institutions. Whenever customers go to a banking location to receive service, they join the back of the queue before they can be served. The customer at the front of the queue is the first to receive service. The queue is also used in Computer Science for scheduling and buffering problems. The fundamental operations of the queue ADT are the following.

1. ISQ-EMPTY(Q) - an operation that returns the binary value **TRUE** if queue Q is empty, and **FALSE** otherwise.
2. ENQUEUE(Q, x) - an operation that adds element x to the *rear* of queue Q, and
3. DEQUEUE(Q) - an operation that removes the element at the *front* of queue Q and returns it. If the queue is empty, $NULL$ is returned.

One theoretical characteristic of a queue worth noting is that it does not have a specific size or capacity. Regardless of the number of elements already in a queue, a new element can always be added to it. An empty queue cannot be dequeued because there is no element to remove from it. The rear of a queue is the end where an element is inserted into the queue or where the ENQUEUE operation is performed. The front, on the other hand, is the end where an element is removed from the queue or where the DEQUEUE operation is performed. We now provide analysis for the queue ADT. First, we present an implementation and analysis of a queue in heap memory.

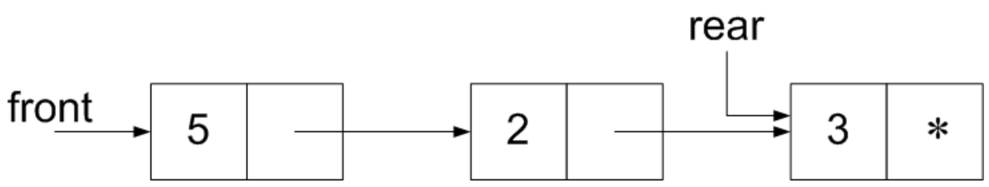

Figure 3.7: Linked list representation of a queue.

3.5.1 Typical implementation of queue

One typical implementation of the queue ADT in the heap uses a singly linked list data structure with two special pointers, $front$ and $rear$. See Figure 3.7 for a depiction of the singly linked list representation of a queue. The ISQ-EMPTY operation checks whether $front$ points to $NULL$. The ENQUEUE operation adds a new element to the rear end of the linked list and updates the $rear$ pointer. The DEQUEUE operation updates the $front$ pointer and returns the element that was at the front of the linked list.

The ISQ-EMPTY operation—singly linked list implementation

ISQ-EMPTY(Q)	*line*	*time cost*	*frequency*
1 **return** $Q[front] = NULL$	1	c_1	1

Figure 3.8: Procedure to determine whether the queue is empty—singly linked list implementation.

Assume the running time of line 1 of the ISQ-EMPTY operation of Figure 3.8 is c_1. Line 1 is executed only once per invocation of ISQ-EMPTY. Thus, the worst-case running time of ISQ-EMPTY is given as

$$\begin{aligned} T(n) &= c_1 * 1 \\ &= c_1 \\ &= O(1) \end{aligned}$$

The ENQUEUE operation—singly linked list implementation

```
ENQUEUE($Q, x$)
1  $x[Next] \leftarrow NULL$
2  if ISQ-EMPTY(Q)
3      then $Q[front] \leftarrow x$
4          $Q[rear] \leftarrow x$
5      else $Q[rear[Next]] \leftarrow x$
6          $Q[rear] \leftarrow x$
```

line	*time cost*	*frequency*
1	c_1	1
2	c_2	1
3	c_3	1
4	c_4	1
5	c_5	1
6	c_6	1

Figure 3.9: Procedure to add an element to the rear of the queue—singly linked list implementation.

As depicted in Figure 3.7 and Figure 3.9, an element is added to the rear of the queue. The matrix in Figure 3.9 gives the running time and frequency of executing each line of the ENQUEUE operation. The worst-case running time for ENQUEUE is thus given as $T(n) = max(T_1(n), T_2(n))$ where

$$\begin{aligned} T_1(n) &= c_1 * 1 + c_2 * 1 + c_3 * 1 + c_4 * 1 \\ &= c_1 + c_2 + c_3 + c_4 \\ &= O(1) \\ T_2(n) &= c_1 * 1 + c_2 * 1 + c_5 * 1 + c_6 * 1 \\ &= c_1 + c_2 + c_5 + c_6 \\ &= O(1) \end{aligned}$$

But $T_1(n)$ and $T_2(n)$ are both $O(1)$ running times. Thus, the worst-case running time for ENQUEUE is $T(n) = O(1)$.

The DEQUEUE operation—singly linked list implementation

```
DEQUEUE($Q$)                                          line  time cost  frequency
1  $x \leftarrow Q[front]$                             1      $c_1$        1
2  if !ISQ-EMPTY(Q)                                    2      $c_2$        1
3      then $Q[front] \leftarrow Q[front[Next]]$       3      $c_3$        1
4  return $x$                                          4      $c_4$        1
```

Figure 3.10: Procedure to remove an element from the front of the queue—singly linked list implementation.

The DEQUEUE operation in Figure 3.10 removes the element at the front of the queue and returns it. The front of the queue is adjusted to point to the element immediately following the dequeued element. The worst-case running time for the DEQUEUE operation is thus computed as

$$\begin{aligned} T(n) &= c_1 * 1 + c_2 * 1 + c_3 * 1 + c_4 * 1 \\ &= c_1 + c_2 + c_3 + c_4 \\ &= O(1) \end{aligned}$$

ENQUEUE and ISQ-EMPTY each runs in $O(1)$ time. Thus, the worst-case running time for each operation of the queue ADT implemented using singly linked list is $T(n) = O(1)$.

3.5.2 Scoped-memory implementation of queue

Consider execution of an application A that manages a queue instance in an RTSJ scoped-memory environment. Efficient execution of A depends on proper management of memory, which is a limited resource. Assume A uses a stack of scoped-memory instances to manage the queue. Assume also, for the purpose of worst-case analysis, that a queue element resides in its own scope when enqueued. A service stack with its own NHRT T_1 is used to facilitate the ENQUEUE operation. Figure 3.11 shows a representation of a queue instance. If T_0 is the application thread, then T_0 is a NHRT. Detailed analysis of the fundamental queue operations follows.

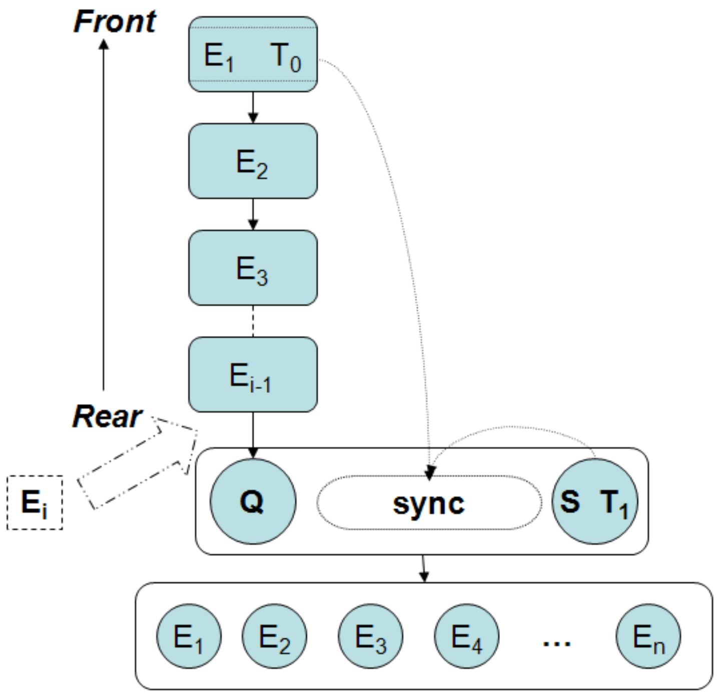

Figure 3.11: Representation of a queue instance in an RTSJ scoped-memory environment. Rounded rectangles represent scoped-memory instances and ovals represent element instances. T_0 is the application thread and T_1 services the stack. The arrows pointing downward represent legal scope references. The *sync* field is a synchronization point for T_0 and T_1. E_1 denotes element i.

The ISQ-EMPTY operation—scoped-memory implementation

```
ISQ-EMPTY($Q$)                          line   time cost   frequency
1 **return** $front = Q$                1      $c_1$       1
```

Figure 3.12: Procedure to determine whether the queue is empty—scoped-memory implementation.

The current scope contains a *front* field that points to the front of the queue. An empty queue is a queue with no elements. Emptiness, in Figure 3.12, is illustrated by the *front* field of the current scope pointing to the queue object itself. Assuming that the running time of the only line of ISQ-EMPTY is c_1, the worst-case running time of

ISQ-EMPTY is given as

$$\begin{aligned} T(n) &= c_1 * 1 \\ &= c_1 \\ &= O(1) \end{aligned}$$

The DEQUEUE operation—scoped-memory implementation

```
DEQUEUE($Q$)
1  if ISQ-EMPTY(Q)
2      then $x \leftarrow NULL$
3      else $x \leftarrow front$
4  return $x$
```

line	*time cost*	*frequency*
1	c_1	1
2	c_2	1
3	c_3	1
4	c_4	1

Figure 3.13: Procedure to remove an element from the front of the queue—scoped-memory implementation.

The DEQUEUE operation removes the element at the front of the queue and returns it if one exists. Otherwise, it returns $NULL$. A close examination of the DEQUEUE operation in Figure 3.13 reveals that it is very similar to the POP operation in Figure 3.6. Hence, the worst-case running time for DEQUEUE is $T(n) = O(1)$ time.

The ENQUEUE operation—scoped-memory implementation

```
ENQUEUE($Q, x$)
1  while !ISQ-EMPTY(Q)
2      do $sync \leftarrow$ DEQUEUE(Q)
3          PUSH(S, $sync$)
4  $S_c \leftarrow new\ ScopedMemory(m)$
5  $enter(S_c, T_0)$
6  $front \leftarrow x$
7  while !IS-EMPTY(S)
8      do $sync \leftarrow$ POP(S)
9          PUSH-Q(Q, $sync$)
```

time cost	*frequency*
c_1	$n+1$
c_2	n
c_3	n
c_4	1
c_5	1
c_6	1
c_7	$n+1$
c_8	n
c_9	n

Figure 3.14: Procedure to add an element to the rear of the queue—scoped memory implementation. Each c_i is a constant and $n = |Q| + |S|$. Initially stack S is empty.

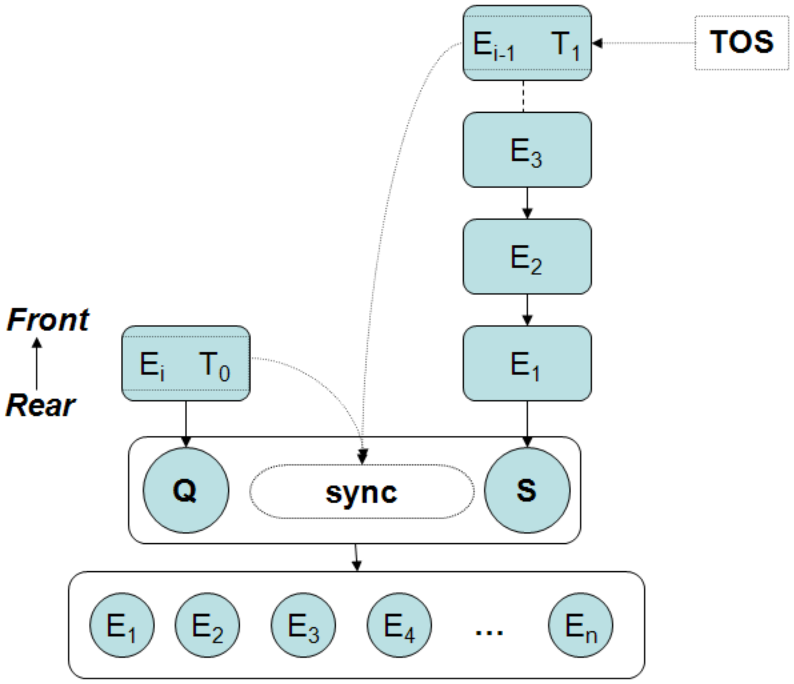

Figure 3.15: Storing queue elements on stack to facilitate enqueue of E_i.

The ENQUEUE operation depicted in Figure 3.14 is a complex operation because of the referencing constraints imposed by the RTSJ: objects in an ancestor scope cannot reference objects in a descendant scope because the descendant scope is reclaimed before the ancestor scope. Consequently, the elements in a queue must first be stored somewhere before a new element can be enqueued. After the element is enqueued, all the stored elements are put back on the queue in the correct order. A stack is an ideal structure to store the queue elements because it preserves the order of the elements for the queue; see Figure 3.15. As illustrated in figures 3.11, 3.15, and 3.16 two threads are needed to facilitate the ENQUEUE operation: one for the queue and one to service the stack. The thread that services the queue is the application thread and is referred to as T_0 in Figure 3.15. T_1 is the service thread for the stack. These two threads are synchronized by a parameter *sync*, which they use to share data between them—see Figure 3.14.

The PUSH-Q method in Figure 3.17 is a private method that puts a stored element back on the queue in the way that the PUSH operation works for a stack. The worst-case

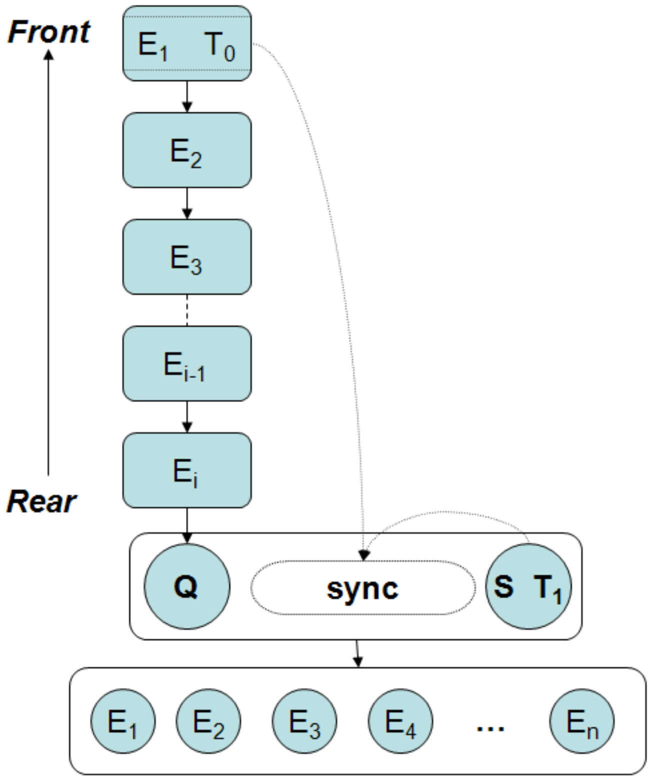

Figure 3.16: E_i is enqueued.

running time for this method is $T(n) = O(1)$ time. This is the same running time for the PUSH operation in Figure 3.5.

PUSH-Q(S, x)	*line*	*time cost*	*frequency*
1 $scope \leftarrow new\ ScopedMemory(m)$	1	c_1	1
2 $enter(scope, T_a)$	2	c_2	1
3 $front \leftarrow x$	3	c_3	1

Figure 3.17: Private helper method that puts an element at the front of the queue in the same manner that an element is pushed onto a stack—scoped-memory implementation. $m \geq |x| + |front|$.

Given the matrix in Figure 3.14 the worst-case running time for ENQUEUE is computed as follows.

$$\begin{aligned}
T(n) &= (n+1)c_1 + nc_2 + nc_3 + c_4 + c_5 + c_6 + \\
&\quad (n+1)c_7 + nc_8 + nc_9 \\
&= (c_1 + c_2 + c_3 + c_7 + c_8 + c_9)n + c_1 + c_4 + \\
&\quad c_5 + c_6 + c_7 \\
&= c_b * n + c_a \\
&= O(n)
\end{aligned}$$

Thus, the worst-case running time for the ENQUEUE operation is $T(n) = O(n)$.

3.5.3 Cumulative analysis for queue

We compute the theoretical running time for an intermixed sequence of n ENQUEUE and DEQUEUE operations on a queue instance by analyzing the worst-case running time of the sequence. Since we suggested two implementation contexts for the queue ADT we compute the running time for each implementation. Suppose that starting with an empty queue n denotes the number of operations in the sequence and m denotes the number of ENQUEUE operations. Then the number of DEQUEUE operations is given as $n-m$ where $n-m \leq m \leq n$. The worst-case running time for the heap implementation is thus given as:

$$\begin{aligned}
T(n) &= T_{enq}(m) + T_{deq}(n-m) \\
&= m * c_1 + (n-m) * c_2 \\
&= nc_2 + m(c_1 - c_2) \\
&= O(n)
\end{aligned}$$

This is identical to the linked-list analysis of an intermixed sequence of PUSH and POP operations on a stack because the insertion operation and the deletion operation each executes in constant time.

The scoped-memory implementation for the ENQUEUE operation is complex. Thus, the running time for the sequence of operations in that context is also complex and more costly than the heap implementation. We compute the worst-case running time for the scoped-memory implementation of the intermixed sequence of operations as follows:

$$\begin{aligned} T(n) &= T_{enq}(m, \vec{s}) + T_{deq}(n-m) \\ &= T_{enq}(m, \vec{s}) + (n-m) * c_2 \end{aligned}$$

$\vec{s} = \langle s_1, s_2, \ldots, s_m \rangle$ is included as input to the computation of the running time for the ENQUEUE operation because the running time of each invocation of the ENQUEUE operation depends on the number of elements in the queue; s_i denotes the number of elements on the queue before the ith operation. Given fixed values for n and m, the worst-case running time for the sequence of operations occurs when no DEQUEUE operations precede an ENQUEUE operation. In this case the values in $\vec{s}$ are monotonically increasing from 0 to $m-1$; so for the computation of $T(n)$ given below, $s_i = i-1$. c_a and c_b are derived from the ENQUEUE analysis above and c_d is the time for the constant DEQUEUE operation.

$$\begin{aligned}
T(n) &= T_{enq}(m, \vec{s}) + T_{deq}(n-m) \\
&= \sum_{i=1}^{m}(c_a + s_i c_b) + T_{deq}(n-m) \\
&= \sum_{i=1}^{m}(c_a + (i-1)c_b) + T_{deq}(n-m) \\
&= mc_a + \left(\sum_{i=1}^{m} c_b i\right) - mc_b + T_{deq}(n-m) \\
&= mc_a - mc_b + c_b\left(\sum_{i=1}^{m} i\right) + T_{deq}(n-m) \\
&= m(c_a - c_b) + c_b\frac{m(m+1)}{2} + T_{deq}(n-m) \\
&= m(c_a - c_b) + \frac{m^2 c_b}{2} + \frac{mc_b}{2} + T_{deq}(n-m) \\
&= \frac{c_b}{2}m^2 + \frac{2c_a - c_b}{2}m + (n-m)c_d \\
&= \frac{c_b}{2}m^2 + \frac{2c_a - c_b - 2c_d}{2}m + c_d n
\end{aligned}$$

Since $m \leq n$ it follows that $T(n) = O(n^2)$. Thus, for a scoped-memory queue implementation the worst-case running time for an intermixed sequence of n ENQUEUE and DEQUEUE operations is $T(n) = O(n^2)$.

3.5.4 Discussion

Two possible implementations for the queue ADT were presented: a singly-linked-list implementation and an RTSJ scoped-memory implementation. The singly-linked-list implementation gives $T(n) = O(1)$ worst-case execution time for each queue operation. The scoped-memory implementation gives $T(n) = O(1)$ worst-case execution time for the ISQ-EMPTY and DEQUEUE operations. The ENQUEUE operation requires but $T(n) = O(n)$ time. The referencing constraints imposed by the RTSJ's scoping rules are the reasons for the linear worst-case execution time. Scopes are instantiated in a stack-like fashion. Thus, to enqueue an element the scope stack must first be popped and the element in each scope must be stored on a stack or another data structure. A new scope to enqueue the element must then be instantiated from the base of the scope stack and be placed on the queue. The elements stored away for the ENQUEUE operation must then be restored on the queue in a LIFO manner.

In addition to performing analysis for each operation, we performed analysis for a sequence of n consecutive queue operations on a queue instance. The singly-linked-list implementation gives a worst-case running time of $O(n)$ and the scoped-memory implementation gives a worst-case running time of $O(n^2)$, *i.e.*, an order of magnitude worse than the running time for the singly-linked-list implementation. This is expensive for an environment that governs its own memory and gives NHRTs higher priorities than any garbage collector.

3.5.5 Improved scoped-memory implementation of queue

We presented thus far an implementation for queue in an RTSJ scoped-memory environment with a worst-case running time of $O(n^2)$ for n consecutive ENQUEUE operations. Here, we present a modified queue implementation that has better worst-case time performance on a sequence of queue operations, see Figure 3.18.

As with the previous implementation, we use a service stack with its own NHRT T_1 to manage the queue. We also limit each scope to holding at most one queue element. The ISQ-EMPTY and DEQUEUE operations remain the same as those presented above. Whereas before we copied the entire queue over to the service stack for each ENQUEUE operation, now we do so only for the ith ENQUEUE operation when i is a power of 2. After the queue elements are stored on the service stack, but before they are placed back in the queue in their previous order, we create not one but i new scopes at the rear of the queue. The new element is enqueued in the deepest, new scope—the one closest to the front of the queue. The other scopes remain empty until they are filled by subsequent ENQUEUE operations.

```
ENQUEUE($Q, x$)
 1  $n_{\text{enq}} \leftarrow n_{\text{enq}} + 1$
 2  if $n_{\text{enq}}$ is some power of 2
 3     then while !ISQ-EMPTY(Q)
 4              do $sync \leftarrow$ DEQUEUE(Q)
 5                 PUSH(S, $sync$)
 6          for $i = 1$ to $n_{\text{enq}}$
 7              do $S_c \leftarrow new\ ScopedMemory(m)$
 8                 $enter(S_c, T_0)$
 9                 $front \leftarrow getOuterScope()$
10          $thread[next] \leftarrow front$
11          $front \leftarrow x$
12          while !IS-EMPTY(S)
13              do $sync \leftarrow$ POP(S)
14                 PUSH-Q(Q, $sync$)
15     else $temp \leftarrow thread[next][front]$
16          $thread[next][front] \leftarrow x$
17          $thread[next] \leftarrow temp$
```

Figure 3.18: Procedure to add an element to the rear of the queue—scoped memory implementation.

Suppose we start with an empty queue and perform 15 consecutive ENQUEUE operations. The queue now has 15 elements, each in its own scope. Suppose another ENQUEUE operation is to be performed. First, the elements already in the queue are moved over to the service stack. Then, not one but 16 nested scopes, each capable of holding one queue element, are instantiated. The element being enqueued is placed in the most deeply nested scope, *i.e.*, the one closest to the front of the queue. The 15 elements on the service stack are then placed back in the queue in their correct order. The next 15 ENQUEUE operations will fill the empty scopes without having to use the service stack or to instantiate new scoped. A field n_{enq} in the synchronized shared memory in Figure 3.18 (in the scope containing both the queue and service stack) will keep track of the number of times ENQUEUE has been called.

3.5.6 Cumulative analysis for queue revisited

The worst-case running time for a single call of the ENQUEUE operation is $O(n)$, where n is the number of elements already on the queue. The worst-case running time for n consecutive ENQUEUE calls, starting with an empty queue, might reasonably be expected to be $O(n^2)$.

line	*time cost*	*freq. when* $n_{\text{enq}} = 2^x$	*freq. otherwise*
1	c_1	1	1
2	c_2	1	1
3	c_3	$n+1$	0
4	c_4	n	0
5	c_5	n	0
6	c_6	$n_{\text{enq}}+1$	0
7	c_7	n_{enq}	0
8	c_8	n_{enq}	0
9	c_9	n_{enq}	0
10	c_{10}	1	0
11	c_{11}	1	0
12	c_{12}	$n+1$	0
13	c_{13}	n	0
14	c_{14}	n	0
15	c_{15}	0	1
16	c_{16}	0	1
17	c_{17}	0	1

Figure 3.19: Statistics for procedure in Figure 3.18. Each c_i is a constant and $n = |Q| + |S|$. Initially stack S is empty and so $n = |Q|$.

Fortunately, that turns out not to be the case. Consider beginning with an empty queue and performing a series of n ENQUEUE operations with no intervening DEQUEUE operations. During the ith ENQUEUE call, $n = i - 1$ (since n is the number of elements already on the queue) and $n_{\text{enq}} = i$ (after the shared-memory field n_{enq} is incremented as the first step of the ENQUEUE algorithm). It can be seen from Figure 3.19 that the ith ENQUEUE call takes $c_a + c_b i$ time if $i = 2^x$ for some integer x, where

$$\begin{aligned} c_a &= c_1 + c_2 - c_4 - c_5 + c_6 + c_{10} + c_{11} - c_{13} - c_{14} \\ c_b &= c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{12} + c_{13} + c_{14} \end{aligned}$$

and c_c time otherwise, where

$$c_c = c_1 + c_2 + c_{15} + c_{16} + c_{17}$$

Assuming $n = 2^x$ for some integer x (which is a worst case, since the last ENQUEUE will be a linear-time and not a constant-time operation), the total running time for all n

ENQUEUEs is given as

$$\begin{aligned}
T(n) &= \sum_{j=0}^{x}(c_a + c_b 2^x) + (2^x - x - 1)c_c \\
&= (x+1)c_a + \left(\sum_{j=0}^{x} 2^x\right) c_b + (2^x - x - 1)c_c \\
&= (x+1)c_a + (2^{x+1} - 1)c_b + (2^x - x - 1)c_c \\
&= (2c_b + c_c)2^x + (c_a - c_c)x + c_a - c_b - c_c \\
&= (2c_b + c_c)n + (c_a - c_c)\log_2 n + c_a - c_b - c_c \\
&= O(n)
\end{aligned}$$

Thus, the improved ENQUEUE operation has a worst-case running time of $T(n) = O(n)$ on the sequence of operations. However, because it overallocates when resizing, it relies at some point on having twice the number of scopes allocated as are actually in use. Interestingly, a space-time trade-off of this nature is also endemic to real-time collectors [9]. Still, the memory required is bounded and proportional to the maximum number of elements in the queue at any given time.

3.6 Functional programming parallel

In Section 3.3.2 we introduced a model to compute asymptotic bounds for scoped-memory areas and NHRTs. We used the model to analyze the runtime behavior of the stack and queue abstract data types. In this section, we articulate a new and interesting relationship between RTSJ programs and *functional* programs. The result of our findings offers RTSJ developers some relief in migrating extant functional implementations of popular data structures and analyses of their runtime behaviors to the RTSJ.

We say an RTSJ program P is *scope-safe* if no execution of P can issue an `IllegalAssignmentError` exception. Such exceptions are issued if the program fails to follow the scope-access rules discussed in Section 2.4.

Theorem 3.6.1 *Static determination of the scope-safety of an RTSJ program is undecidable.*

Proof: *By reduction from the halting problem:* Given an encoding of a Turing machine T and its input w, we construct an RTSJ program P as follows:

- P simulates T on w by interpreting T in standard Java: no RTSJ features are used.
- If T should halt on w, then P instantiates two scoped-memory areas, A and B, where A is the parent of B. P next issues a reference from A to B.

Clearly, P generates an `IllegalAssignmentError` if and only if T halts on w. Thus, deciding (statically) that P halts also decides that T halts on input w, which contradicts the undecidability of the halting problem. ■

Theorem 3.6.1 implies that a compiler cannot generally detect programs that would execute without error in Java but fail due to scope errors in the RTSJ. Extant responses to this problem can be summarized as follows:

- A program can be written in a subset of the RTSJ that provably avoids scope errors [67], or annotations can be attached to RTSJ programs so that a compiler can reason about scope-safety [16].

  While this approach can be successful, an application must essentially be rewritten to conform with restrictions or to supply annotations. Moreover, a developer must understand the application at a depth sufficient to modify the application correctly.

  Java's extensive libraries offer significant functionality for developers, but they are inherently unsuitable for use in the RTSJ's scoped-memory areas. Rewriting the libraries for the RTSJ is a daunting task, with no real guarantee of correctness or efficiency.

- Scopes can be avoided by using ordinary Java with a real-time garbage collector [9].

> While this approach avoids having to rewrite an application, certain program properties must be asserted or analyzed [60] to configure the automatic garbage collector so that it sufficiently paces the application's storage needs. Some time efficiency will be lost, as a predictable share of the CPU must be given to the garbage collector. Some space efficiency is also lost, as the heap must be sufficiently over-provisioned to mitigate the collector's share of the CPU.
>
> Even at its best, this approach has its skeptics, and there are (hard real-time) applications for which developers believe they must avoid garbage collection.

As an alternative to modifying Java programs to be RTSJ-safe, we consider an apparently different programming paradigm and show that programs written in that paradigm can be easily moved to the RTSJ and enjoy scope-safety.

Functional programming languages have emerged as an alternative to the more prevalent style of programming languages (including Java and the RTSJ) in which state, and mutation of state, dominate the design and construction of programs. Lisp [61] is perhaps the earliest example of a practical functional programming language still in use today, and Backus's Turing lecture [6] inspired generations of research on functional programming languages.

The property of a pure Lisp program most relevant to our work concerns its mathematical transparency: Lisp expressions can be manipulated mathematically, because the symbols of a symbolic expression cannot change value unexpectedly. Languages like pure Lisp achieve this property by allowing names to be associated with expressions at most once. This "single assignment" rule allows mathematical substitution of a program's names but also implies the following property: in terms of the order of assignment of expressions to names, the expression assigned to a given name can reference only those names that are strictly older than the assigned name. We leverage that property to build scope-safe RTSJ functions.

We use Lisp as an example, but extensions to other pure functional programming languages are straightforward. Memory is allocated in Lisp programs by a `cons` operator, which creates a memory cell containing at most two references to extant storage. We realize a Lisp program's storage allocation in the RTSJ as follows:

- The RTSJ program prepares to simulate the Lisp program by creating a NHRT in the usual manner. The details need not be provided here, except to say that the program is subsequently able to create scoped-memory areas.

- Each `cons` operator in the Lisp program is simulated by creating and entering a new scoped-memory area with sufficient storage for two references (which we assume could also accommodate non-reference data such as constants). The references for a `cons` cell must be known in the Lisp program when the `cons` cell is instantiated; we populate the RTSJ scope with precisely those references.

While a scoped-memory area per `cons` cell is inefficient in practice, this approach allows us to reason about the nature of storage allocated in the corresponding RTSJ program:

- No scoped-memory area will overflow. This follows from the construction of the scoped-memory areas. Each is populated once and for always by the single `cons` cell that prompted creation of the scope.

- All references created in this manner are scope-safe, as proved by the following theorem.

Theorem 3.6.2 *The RTSJ realization of a Lisp program is scope-safe.*

Proof: *By contradiction:* If a scope-referencing error occurs, one of the following must be its cause:

- A reference is made to scoped memory from an unsuitable memory area (the generic heap). If so, then the program did not launch the NHRT as described above.

- An inappropriate reference is made between scoped-memory areas. There are two cases:
  - The areas are not in an ancestor-descendant relationship. This is a contradiction, since all scopes are created with linear ancestry.
  - The reference is made from an ancestor scope to a descendant scope. This is a contradiction, since the functional program can only have newer cells reference older cells.

■

An important consequence of Theorem 3.6.2 is that an RTSJ developer can consider migration of extant code written in a pure functional language for deployment under the RTSJ, without fear of scoped-memory referencing errors at runtime. As described in Sections 3.7–3.8, data structures such as lists and heaps can be implemented in scopes based on their realization in a functional programming language.

Code migrated as described above creates a linear chain of scopes—one for each `cons` cell. However, the mutator entering those scopes never retreats, so the resulting program never deallocates storage. At any moment, the RTSJ application could suspend its primary activity and enter a phase in which it traces program references through the scope chain, copying the resulting objects into a new chain of nested scopes. Any objects not referenced by the program would not be copied. Such a phase essentially emulates a copying garbage collector, but the intent of using the RTSJ with scoped-memory areas is to avoid garbage collection.

Thus, the pure-functional implementations can serve as a basis for code migration; however, more work is necessary to obtain space-efficient RTSJ implementations. In Sections 3.4 and 3.5 we considered liveness issues for each data structure and each was mindful of reclaiming storage where possible. Generally, liveness of a given data structure, in the context of a real application that uses multiple data structures, must be considered to determine a more sophisticated scope structure and to determine when scopes should be exited so that storage can be reclaimed.

In addition to the storage-reclamation problem, a data structure migrated without due consideration may be unsuitable for a real-time application. The rest of this chapter provides examples that illustrate the advantages and pitfalls of code migration for real-time applications.

As an example of migrating functional language implementations and runtime analyses to the RTSJ, we consider some of the data structure implementations due to Okasaki [69, 70]. Because they were developed for general use, and without regard to real-time requirements, the primary consideration of merit was the normative *average* running time, analyzed over a typical usage pattern. Real-time applications must budget for worst-case conditions. As such, it is important to analyze a data structure's migration from the functional programming paradigm to the RTSJ with an understanding of the resulting asymptotic worst-case behavior.

In Sections 3.4 and 3.5 we suggested particular RTSJ implementations for the stack and queue abstract data types, respectively, and analyzed their time-complexities. The RTSJ scopes (and functional programming languages) behave in a stack-like fashion. As such, an RTSJ implementation for stack follows naturally. Our RTSJ queue implementation is similar to Okasaki's [69] functional language implementation and yields similar time-complexity analysis. Both have been carefully crafted to have properties desirable for real-time applications.

In Sections 3.7 and 3.8 we use the transformation described in Section 3.6 to migrate functional programming implementations of data structures and their time-complexity analyses to the RTSJ.

3.7 List analysis

The *list* ADT is an ADT that formalizes the notion of an ordered collection of entities or items. The fundamental operations of list are:

1. ISLIST-EMPTY(L) - an operation that returns the binary value **TRUE** if list L is empty, and **FALSE** otherwise.
2. SIZE(L) - an operation that returns the number of elements in list L.
3. CREATE(L) - an operation that creates an empty list L.
4. INSERT(L, x) - an operation that inserts item x at the front of list L.
5. HEAD(L) - an operation that returns the item at the front of list L.
6. DELETE-ITEM(L) - an operation that removes the item located at the front of list L and returns a list containing one fewer item. If L is empty, an error condition is reported.
7. LOOKUP(L, i) - an operation that returns the item located at index i of list L. If L contains fewer than i items, an error condition is reported.
8. UPDATE(L, i, x) - an operation that replaces the item at index i of list L with item x. If L contains fewer than i items, an error condition is reported.

3.7.1 Typical implementation of list

In the heap, the singly-linked list or the doubly-linked list data structure is typically used to implement the list ADT. For these implementations the ISLIST-EMPTY, SIZE, CREATE, HEAD, and DELETE-ITEM operations each executes in $O(1)$ time. The LOOKUP and UPDATE operations each requires $O(n)$ time since the list has to be searched to find the requested index.

3.7.2 Scoped-memory implementation of list

We do not suggest a particular scoped-memory implementation as we did in Sections 3.4 and 3.5. Instead, we migrate a functional language implementation of list [69, 70] to the RTSJ using the method described in Section 3.6. The runtime cost analysis also migrates since the transformation described in Section 3.6 is constant for each `cons` operation and linear in the number of such operations. Moreover, the resulting RTSJ implementation is functionally equivalent to the functional programming language implementation.

In his dissertation, Okasaki [70] implemented the list ADT in Standard ML, a functional programming language. The declaration of each operation is similar to those given above. He analyzed the running time for each list operation; we use his analysis and the results from Section 3.6 to give the time complexity for each list operation implemented with the RTSJ scoped-memory areas. The ISLIST-EMPTY, SIZE, CREATE, HEAD, and DELETE-ITEM operations each executes in $O(1)$ time while the LOOKUP and UPDATE operations each requires $O(\log n)$ time.

3.7.3 Cumulative analysis for list

Suppose there exists a list L with n items. We consider computing the running time of executing an intermixed sequence of m LOOKUP and UPDATE operations (the most expensive operations) on list L. In a heap implementation, the running time for this sequence of operations is $O(mn)$. In a scoped-memory implementation, the running time is $O(m \log\ n)$. While it appears that a scoped-memory implementation is more efficient than a heap implementation, the scoped-memory implementation can leak an unbounded amount of memory.

3.8 Heap analysis

The *heap* or *priority queue* is an ADT that, at a minimum, allows the following operations: INSERT, which inserts an element in the heap; and DELETE-MIN, which finds, returns, and deletes the minimum element from the heap. The heap is generally implemented as a tree-based data structure that satisfies the *structure property* and the *heap order property.* The structure property says that a heap is implemented as a binary tree that is completely filled, with the possible exception being the leaves level, which is filled from left to right [92]. The heap order property requires that data in the heap be an ordered set. Since the minimum element needs to be found quickly, the heap order property requires that the smallest element be at the root of the heap. If it is required that every subtree be a heap, then any node in the heap should be smaller than its descendants. The implementation of the heap ADT that honors these properties is called the binary heap.

Another implementation of the heap ADT is the binomial heap. A binomial heap is similar to a binary heap except that the operation that merges two heaps runs faster. Thus, we consider the binomial heap in our analysis. We define the fundamental operations for the heap ADT as follows:

1. CREATE(H) - an operation that creates an empty heap H.
2. ISHEAP-EMPTY(H) - an operation that returns the binary value **TRUE** if heap H is empty, and **FALSE** otherwise.
3. INSERT(H, x) - an operation that inserts an item in heap H.
4. FIND-MIN(H) - an operation that finds and returns the minimum item in heap H.
5. DELETE-MIN(H) - an operation that removes the minimum item from heap H and returns a new heap with one fewer item. If H is empty, an error condition is reported.
6. MERGE(H_1, H_2) - an operation that merges heap H_1 with heap H_2 to form a new heap containing as many items as the sum of the number of items in H_1 and H_2 combined.

3.8.1 Typical implementation of heap

In the heap where dynamic memory management occurs, several options are available for implementing the heap ADT. An array can be used to store the heap; a binary tree can be used to implement the heap; a binomial tree can also be used to implement the heap. We consider the binomial tree implementation, more specifically the binomial heap data structure, in our analysis for the reasons given above. Consequently, the cost associated with each operation is given as follows. The CREATE and ISHEAP-EMPTY operations each takes $O(1)$ time to execute. The other operations each requires $O(\log n)$ time. This is not surprising since the height of the tree used to store the heap is $O(\log n)$, where n is the number of nodes (items) in the tree.

3.8.2 Scoped-memory implementation of heap

As we did for the list ADT, we do not suggest a specific way to implement the heap ADT using the RTSJ scoped-memory areas. Instead, we migrate to this section implementations and cost analyses of the running time of heap operations from the functional programming language community. In particular, we migrate implementations and analyses from Okasaki [70]. Okasaki used a binomial heap implementation for the heap ADT, which he developed in Standard ML. He analyzed the running time of each operation and obtained a complexity of $O(\log n)$ for each operation, except CREATE and ISHEAP-EMPTY, which each executes in $O(1)$ time. Adopting Okasaki's results, we conclude that for an RTSJ scoped-memory implementation of the heap ADT, the operations CREATE and ISHEAP-EMPTY execute in $O(1)$ time. Every other operation, namely INSERT, FIND-MIN, DELETE-MIN, and MERGE, requites $O(\log n)$ time.

3.8.3 Cumulative analysis for heap

Here we consider executing an intermixed sequence of m INSERT, FIND-MIN, DELETE-MIN, and MERGE operations. Interestingly, these operations have the same running time for both a heap implementation and a scoped-memory implementation. Since each operation has a running time of $O(\log n)$ and there are m operations in the sequence, the running time for the sequence of operations is $O(m \log n)$.

Although the results are the same for both implementations, the heap implementation is simple and exists in most data structure texts. Further, scoped-memory implementation of heap is not commonplace. Theorem 3.6.2 allows us to migrate a functional programming language implementation of heap to the RTSJ; however, such implementation can consume an unnecessary amount of memory.

Chapter 4

An Improved on-the-fly reference counting garbage collector

Section 2.2.2 gave an overview of LPC [57, 58], "An On-the-Fly Referencing Counting Garbage Collector for Java", designed for multi-threaded, multi-processor environments. LPC possesses many features of a modern collector, namely incrementality, concurrency, short pause time, low synchronization, and reasonable time overhead. However, improvements can be made to reduce the pause time, to lower the synchronization, to minimize the time cost, and to improve the minimum mutator utilization (MMU) of processors.

LPC executes a series of collections in cycles. During a collection, LPC suspends each mutator 4 times to engage it in a transaction. After each transaction, LPC resumes the mutator. LPC suspends at most one mutator at a time and resumes it before suspending another mutator. A mutator cannot be suspended by LPC if it is executing in the write barrier because the write barrier must be treated with extreme care. The write barrier serves as a synchronization point between mutators and the collector. It is the write barrier the collector uses to force mutators to buffer state information relevant to garbage collection. The notion of suspending mutators, performing transactions with them, and resuming them is known as a handshake. Thus, LPC uses four handshakes per collection. These handshakes are also synchronization points since LPC utilizes them to synchronize its view of the heap with the mutators. Hence, the larger the number of handshakes a collector possesses the higher the synchronization cost. A large number of handshakes does not only increase synchronization cost, but also reduces MMU. A mutator cannot do any work while it is suspended.

We present the Defoe-Deters Collector, an improved on-the-fly collector that reduces pause time, lowers synchronization cost (fewer handshakes), and shortens time overhead. For convenience, we shall hereafter refer to the Defoe-Deters Collector as DDC. DDC is similar to LPC in many respects; however, DDC addresses some of the limitations of LPC.

4.1 Chapter road map

The rest of this chapter is organized as follows. Section 4.2 describes the problem that the LPC and the DDC family of collectors is attempting to resolve. Section 4.3 highlights the mutators' involvement in garbage collection. Sections 4.4 to 4.6 detail the functionality of various versions of the collector. Section 4.7 validates the DDC family of collectors. Section 4.8 discusses implementation issues and Section 4.9 summarizes our results. Section 4.10 describes related work.

4.2 The problem addressed by LPC and DDC collectors

The LPC and the DDC family of collectors use state information buffered by mutators to maintain reference counts of heap objects and to collect such objects when their reference counts become zero. The state information is exchanged between the mutators and collector during the first and/or last handshake of each collection. Since mutators respond to handshakes with the collector on an individual basis, it is possible for the same state information to be concurrently buffered by a mutator that has already had a handshake with the collector and one that has not. This poses a problem since objects can be collected prematurely.

The LPC collector addresses this problem by using two addition handshakes [57, 58] (see Sections 4.3 and 4.6) for a total of four handshakes. Insteading of using additional handshakes, we use a duality approach in the DDC family of collectors. In Sections 4.3 and 4.4 we use a dual buffer approach and in Section 4.5 we use a dual dirty-flags approach.

4.3 Mutators in garbage collection

Mutators affect garbage collection by instantiating new objects and storing references to heap objects in pointer fields. When a mutator instantiates a new object, the object is assigned a default zero heap-reference-count. An object is collected when its reference count is zero; as such, a new object is a candidate for garbage collection. Each mutator T_i is equipped with a local zero-count-table ZCT_i in which it logs objects whose reference counts are zero. Thus, the new object is logged in the current mutator's zero-count-table (ZCT). Figure 4.1 details the instantiation routine.

```
Procedure Instantiate(size: Integer): Object
begin
        // new object obtained from allocator
1.      $obj$ = allocate($size$)
2.      $ZCT_i = ZCT_i \cup \{obj\}$
3.      return $obj$
end
```

Figure 4.1: Instantiation of new object. Similar to procedure **New** in LPC [57, 58].

Each T_i is also equipped with two local buffers, namely Buf_i and TB_i, in which it logs information on pointer fields in objects (pointers for short) that have been updated for the first time since the last collection. Each buffer entry consists of a tuple of addresses: the address of the updated pointer and the address of the object to which it last pointed before it was updated. It is important to note that during a collection, only pointers that are updated for the first time are logged. Other pointers are not logged. There are two paths through the write barrier—a path in which a mutator logs data in its local buffer and a path in which the mutator does not log data in its local buffer. Every pointer has associated with it a dirty flag that indicates whether it has been updated during the current collection or not. If the flag is raised then the pointer has already been updated. Otherwise, the pointer may not yet have been updated. The write barriers represented in Figure 4.2 and Figure 4.3 log the information mentioned above in mutator local buffers. They are only executed for pointer assignments. Notice that the write barriers are identical except they use different buffers. The reasons for the different buffers will become clear when the collector's role is discussed in detail.

```
Procedure Update($s$: Pointer, $new$: Object)
begin
1.   Object $old$ := read($s$)
2.   if $\neg Dirty(s)$ then
3.      $Buf_i := Buf_i \cup \langle s, old \rangle$
4.      $Dirty(s) := true$
5.   write($s$, $new$)
6.   if $Snoop_i$ then
7.      $Locals_i := Locals_i \cup \{new\}$
end
```

Figure 4.2: Reproduced write barrier of LPC [57, 58].

```
Procedure UpdateTwo($s$: Pointer, $new$: Object)
begin
1.   Object $old$ := read($s$)
2.   if $\neg Dirty(s)$ then
3.      $TB_i := TB_i \cup \langle s, old \rangle$
4.      $Dirty(s) := true$
5.   write($s$, $new$)
6.   if $Snoop_i$ then
7.      $Locals_i := Locals_i \cup \{new\}$
end
```

Figure 4.3: Write barrier to be executed by threads released from handshake one. Execution of procedure **Update** by all threads resumes when handshake one completes.

The procedures in Figure 4.1, Figure 4.2, and Figure 4.3 cannot be interrupted by the collector while they are being executed. If they are interrupted, the log entries could be corrupted and garbage collection could fail. These procedures are not necessarily atomic since they can be concurrently executed by multiple mutators; however, they are executed as collector-proof code (called *CP-code*), *i.e.*, code that cannot be suspended by the collector.

Notice also that the write barriers are involved in a *snooping* mechanism—a feature of LPC [57, 58] described as follows. A *sliding view* of the heap is computed during a collection over the interval $[t_1, t_2]$. This view can be perceived as a view of the heap that slides in time. While that view is being computed, pointers are updated. Snooping is a mechanism used to ensure that objects that become targets of pointer updates during the current sliding view computation are not reclaimed at the end of the current collection. Instead, such objects become roots for the current collection and are logged in mutator local buffers denoted $Locals_i$.

4.4 Defoe-Deters Collector

This section describes the main functions of DDC. Pseudocode for each routine is provided.

4.4.1 Initiating a collection

DDC begins a collection by enabling the snooping mechanism described above in Section 4.3. DDC then executes the first handshake, HS_1, during which it performs transactions with each mutator in turn. A transaction consists of the following steps where n is the number of mutators in the system, $1 \leq i \leq n$ is the mutator index, and $k > 0$ identifies the collection.

1. The collector suspends mutator T_i if it is not executing CP-code.

2. The collector retrieves the local buffer, Buf_i, of T_i and consolidates it in a history buffer, $Hist_k$, for the current collection.

3. The collector gives T_i a new buffer Buf_i then resumes T_i.

At the end of the collection, the data in $Hist_k$ is used to adjust the reference counts of heap objects. The procedure for initiating a collection is depicted in Figure 4.4. Before

```
Procedure **InitiateCollection**
begin
1.    for each thread $T_i$ do
2.       $Snoop_i := true$
3.    for each thread $T_i$ do
4.       suspend $T_i$
5.       // retrieve $T_i$'s local buffer ignoring
         // duplicate pointer information
         $Hist_k := Hist_k \cup Buf_i$
6.       // give $T_i$ an empty local buffer
         $Buf_i := \emptyset$
7.       resume $T_i$
end
```

Figure 4.4: Procedure to begin a collection and implement handshake one. This procedure is executed by the collector and similar to procedure **Initiate-Collection-Cycle** of LPC [57, 58].

a mutator is affected by HS_1 it uses the write barrier in Figure 4.2 to update pointers. After encountering HS_1 it uses the write barrier in Figure 4.3 to update pointers until HS_1 completes. When HS_1 completes all threads return to using the write barrier in Figure 4.2. Notice that different local buffers are used in the write barriers. The data in Buf_i is consolidated in $Hist_k$ when HS_1 executes, but the data in TB_i is data collected by mutators since they responded to HS_1. Such data is not consolidated in $Hist_k$ but is eventually added to the history buffer for the next collection, *i.e.*, in $Hist_{k+1}$. Having a write barrier that can be used to collect such data keeps the number of handshakes low.

4.4.2 Resetting dirty flags

The data consolidated in $Hist_k$ was buffered during the previous collection. The pointers listed in $Hist_k$ are exactly the pointers that were updated at least once during the previous collection. To buffer accurately the pointers that are modified the first time during the current collection, the dirty flags of pointers in $Hist_k$ must be reset. This action is necessary to summarize reference-count updates. Only objects affected by pointer updates in successive collections have their reference counts adjusted. Moreover, the write barriers can only be used to buffer pointers with clear dirty flags. The resetting routine is detailed in Figure 4.5.

```
Procedure **ResetDirtyFlags**
begin
1.  for each $\langle s, old\rangle \in Hist_k$ do
2.      $Dirty(s) := false$
end
```

Figure 4.5: Procedure to reset dirty flags of pointers in $Hist_k$. This procedure is executed by the collector and is identical to procedure **Clear-Dirty-Marks** of LPC [57, 58].

4.4.3 Restore dirty flags

DDC targets a multi-thread, multiprocessor environment. This makes it possible for multiple mutators to use the write barriers concurrently to update pointers. After HS_1 completes, one potential consequence is for data on the same pointer(s) to be present in both TB_i and $Hist_k$. This means that the dirty flags of pointers in TB_i may be reset by procedure **Reset-DirtyFlags** in Figure 4.5. But these pointers are a subset of the pointers that were updated at least once for the current collection. Their dirty flags should not be reset. Since their dirty flags are potentially reset, they must be restored. Procedure **RestoreDirtyFlags** in Figure 4.6 restores the dirty flags of affected pointers.

```
Procedure **RestoreDirtyFlags**
begin
1.  $Hist_{k+1} := \emptyset$
2.  $Handled := \emptyset$
3.  local $Temp := \emptyset$
4.  for each thread $T_i$ do
5.      $Temp := Temp \cup TB_i$
6.  for each $\langle s, old\rangle \in Temp$ do
7.      if $s \notin Handled$ then
8.          $Dirty(s) := true$
9.          $Handled := Handled \cup \{s\}$
10.         $Hist_{k+1} := Hist_{k+1} \cup \{\langle s, old\rangle\}$
end
```

Figure 4.6: Procedure retrieves each mutator's temporary buffer TB_i, raises necessary dirty flags and updates $Hist_{k+1}$.

Not only are the dirty flags of the affected pointers restored by procedure **RestoreDirtyFlags**, but the data in the TB_is are also added to $Hist_{k+1}$ - the history buffer for the next collection. The addition is done without duplication. The data could be added to Buf_i instead, but it is more convenient and more efficient to add it to $Hist_{k+1}$. Adding the data to $Hist_{k+1}$ lowers the synchronization cost between the collector and the mutators and also reduces overhead—see Figure 4.4.

4.4.4 Consolidation

```
Procedure Consolidate
begin
1.    local $Temp := \emptyset$
2.    $Locals_k := \emptyset$
3.    for each thread $T_i$ do
4.       suspend $T_i$
5.       $Snoop_i := false$
6.       // retrieve snooped objects
         $Locals_k := Locals_k \cup Locals_i$
7.       // give $T_i$ an empty $Locals_i$ buffer
         $Locals_i := \emptyset$
8.       // copy thread local state and ZCT
         $Locals_k := Locals_k \cup State_i$
9.       $ZCT_k := ZCT_k \cup ZCT_i$
10.      $ZCT_i := \emptyset$
11.      $Temp := Temp \cup Buf_i$
12.      resume $T_i$
13.   // consolidate $Temp$ into $Hist_{k+1}$
      for each $\langle s, old \rangle \in Temp$ do
14.      if $s \notin Handled$ then
15.         $Handled := Handled \cup \{s\}$
16.         $Hist_{k+1} := Hist_{k+1} \cup \{\langle s, old \rangle\}$
end
```

Figure 4.7: Procedure consolidates all the per-thread local information into per-collection buffers. This procedure is similar to handshake four of LPC [57].

The collector engages mutators in a second handshake, HS_2. During HS_2 the snooping mechanism is disabled; mutator local roots are consolidated into a per-collection root buffer; new objects are consolidated into a per-collection ZCT, and mutator local buffers are consolidated, without duplicates, in the history buffer for the next collection. The latter consists of pointers updated since the first handshake.

Consolidation amounts to retrieving the aforementioned mutator buffers, storing them in per-collection buffers, and returning to mutators new buffers to log more data. Consolidation is performed by procedure **Consolidate** in Figure 4.7.

The local state of a mutator in this context, denoted by $State_i$, refers to the collection of pointers to heap objects immediately available to the mutator. The collection includes pointers from the stack, registers, and global variables. Only the mutator and the collector have access to $State_i$. The same is true for $Locals_i$ (defined in Section 4.3). While procedure **Consolidate** serves as HS_2 for DDC, it serves as HS_4 for LPC. **Consolidate** runs faster for DDC than it does for LPC because some of the work of **Consolidate** is done by **RestoreDirtyFlags**.

4.4.5 Adjust reference-count fields

```
Procedure AdjustRC
begin
1.   $Unresolved_k := \emptyset$
2.   for each $\langle s, old \rangle \in Hist_k$ do
3.     $curr := \text{read}(s)$
4.     if $\neg Dirty(s)$ then
5.       $curr.rc := curr.rc + 1$
6.     else
7.       $Unresolved_k :=$
           $Unresolved_k \cup \{s\}$
8.     $old.rc := old.rc - 1$
9.     if $old.rc = 0 \wedge old \notin Locals_k$ then
10.      $ZCT_k := ZCT_k \cup \{old\}$
end
```

Figure 4.8: Procedure adjusts *rc* fields of heap objects identified by pointers in $Hist_k$. This is functionally the same as Figure 9 in LPC [57].

After procedure **Consolidate** completes, DDC has enough information to adjust the reference-counts (*rc*) of heap objects identified by the pointers in $Hist_k$. Objects logged as the ‘old values’ of pointers (last object referenced during the previous collection) in $Hist_k$ have their *rc* fields decremented. Objects referenced by pointers in $Hist_k$ have their *rc* fields incremented. Refer to Figure 4.8 for details. By reading the contents of $Hist_k$ it is not always feasible to identify objects whose reference-counts need to be incremented. Consider, for example, a pointer $p \in Hist_k$ whose dirty flag is raised. The address of object obj, which is the 'old value' of p, is not in $Hist_k$ because p was logged by a mutator after it completed HS_1. The address of obj is thus logged in either a mutator local buffer or $Hist_{k+1}$. In order to identify objects such as obj and correctly adjust their reference counts, additional processing of pointers such as p is required.

```
Procedure **ReadBuffers**
begin
1.     $Peek_k := \emptyset$
2.     for each $T_i$ do
3.     // copy buffers without duplicates
           $Peek_k := Peek_k \cup Buf_i$
end
```

Figure 4.9: Procedure reads mutator local buffers without clearing them. This is the same as procedure **Read-Buffers** in LPC [57, 58].

```
Procedure **ReadHistory**
begin
1.     // copy $Hist_{k+1}$ without duplicates
           $Peek_k := Peek_k \cup Hist_{k+1}$
end
```

Figure 4.10: Procedure reads history buffer of next collection and adds it to $Peek_k$. This is the same as procedure **Merge-Fix-Sets** in LPC [57, 58].

Pointers such as are p are referred to as *undetermined slots* [57, 58]. We prefer to term such pointers *unresolved pointers* because the collector has not yet resolved the objects they reference. Unresolved pointers are logged in buffer $Unresolved_k$ so they can be processed further when the mutator buffers and $Hist_{k+1}$ are accessed. The mutator buffers are read asynchronously by procedure **ReadBuffers** in Figure 4.9. The collector does not clear the buffers, instead it combines the content of the buffers into a per-collection buffer called $Peek_k$. $Hist_{k+1}$ is then read by procedure **ReadHistory**, in Figure 4.10, and added to $Peek_k$ so that $Peek_k$ can be used to determine which objects need to have their *rc* fields incremented. Procedure **IncRC** in Figure 4.11 is used to increment the *rc* fields of those objects.

```
Procedure IncRC
begin
1.   for each $\langle s, old \rangle \in Peek_k$ do
2.       if $s \in Unresolved_k$ then
3.           $old.rc := old.rc + 1$
end
```

Figure 4.11: Procedure increments *rc* fields of objects identified by unresolved pointers. This is the same as procedure **Fix-Undetermined-Slots** in LPC [57, 58].

4.4.6 Reclaim garbage objects

```
Procedure ReclaimGarbage
begin
1.   $ZCT_{k+1} := \emptyset$
2.   for each object $obj \in ZCT_k$ do
3.       if $obj.rc > 0$ then
4.           $ZCT_k := ZCT_k - \{obj\}$
5.       else if $obj.rc = 0 \land obj \in Locals_k$ then
6.           $ZCT_k := ZCT_k - \{obj\}$
7.           $ZCT_{k+1} := ZCT_{k+1} \cup \{obj\}$
8.   for each object $obj \in ZCT_k$ do
9.       Collect(obj)
end
```

Figure 4.12: Procedure determines which objects are garbage and collects them with procedure **Collect**. This is the same as procedure **Reclaim-Garbage** in LPC [57, 58].

After the reference counts of heap objects are adjusted, DDC reclaims garbage objects. Garbage objects are heap objects with zero reference counts that are not marked as roots. An object is marked as a root if it is in $Locals_k$. Objects with pointers in $Hist_{k+1}$ that become garbage are not collected at this time. They are deferred to the next collection since their pointers were last updated during the current collection. Furthermore, they may have become garbage after the current collection started. The procedures responsible for reclaiming garbage objects are **ReclaimGarbage** in Figure 4.12 and **Collect** in Figure 4.13.

```
Procedure **Collect**(*obj:* **Object**)
begin
1.  local $DeferCollection := false$
2.  for each pointer $s \in obj$ do
3.      if $Dirty(s)$ then
4.          $DeferCollection := true$
5.      else
6.          $val := \mathbf{read}(s)$
7.          $val.rc := val.rc - 1$
8.          $\mathbf{write}(s, \mathbf{null})$
9.          if $val.rc = 0$ then
10.             if $val \notin Locals_k$ then
11.                 $\mathbf{Collect}(val)$
12.             else
13.                 $ZCT_{k+1} := ZCT_{k+1} \cup \{val\}$
14. if $\neg DeferCollection$ then
15.     return $obj$ to general purpose allocator
16. else
17.     $ZCT_{k+1} := ZCT_{k+1} \cup \{obj\}$
end
```

Figure 4.13: Procedure collects garbage objects. This is the same as procedure **Collect** in LPC [57].

4.5 Defoe-Deters Collector Version 2

In Section 4.3 and Section 4.4 we described DDC in detail. DDC uses two write barriers and two sets of mutator buffers. In this section, we present DDC version 2—a version that uses one write barrier and dual dirty flags for each pointer. We call this approach DDC-2. DDC-2 uses one set of dirty flags for even-numbered collections and one set of dirty flags for odd-numbered collections. The dirty flags are indexed $\{0, 1\}$. Each mutator T_i is equipped with a field $d_i \in \{0, 1\}$ that indexes the dirty flags. When T_i uses the write barrier to raise a dirty flag for pointer s such that s is used to adjust reference counts during an even-numbered collection, it raises dirty flag d_i ($Dirty_{d_i}(s)$) where $d_i = 0$. When the collection is an odd-numbered collection, T_i raises $Dirty_{d_i}(s)$ where $d_i = 1$. For every mutator T_i, $1 \le i \le n$, d_i is initialized to $k_0(\text{mod } 2)$, where k_0 is the number of the first collection. d_i is subsequently updated as part of the first handshake as illustrated in Figure 4.14.

Figure 4.14 is the same as Figure 4.4 except for line 7 that updates d_i for each T_i. It is important to note that d_i is assigned the value $(k + 1)(\mathbf{mod}\ 2)$ and not k(**mod**

```
Procedure **DualMode-InitiateCollection**
**begin**
**1.**  for each thread $T_i$ do
**2.**      $Snoop_i := true$
**3.**  for each thread $T_i$ do
**4.**      suspend $T_i$
**5.**      // retrieve $T_i$'s local buffer ignoring
            // duplicate pointer information
            $Hist_k := Hist_k \cup Buf_i$
**6.**      // give $T_i$ an empty local buffer
            $Buf_i := \emptyset$
**7.**      // set the dual dirty flag index
            $d_i := (k+1)(\mathbf{mod}\ 2)$
**8.**      resume $T_i$
**end**
```

Figure 4.14: **InitiateCollection** modified to use dual dirty flags for each pointer in an object. i is the thread index and k is the collection number.

2) because during collection k mutators buffer data to adjust reference counts of objects during collection $k+1$. DDC-2 also utilizes a single write barrier, depicted in Figure 4.15, to summarize the behaviors of the write barriers in Section 4.3 and averts the procedure in Figure 4.6. Figure 4.15 raises the right dirty flags and eliminates the need to restore dirty flags that inadvertently get reset.

```
Procedure **UpdateDM**(*s:* **Pointer**,
                       *new:* **Object**)
**begin**
**1.**  Object *old* := **read**($s$)
**2.**  if $\neg Dirty_0(s) \wedge \neg Dirty_1(s)$ then
**3.**      $Buf_i := Buf_i \cup \langle s, old \rangle$
**5.**      // *dual dirty flags*
            $Dirty_{d_i}(s) := \mathbf{true}$
**6.**  **write**($s$, *new*)
**7.**  if $Snoop_i$ then
**8.**      $Locals_i := Locals_i \cup \{new\}$
**end**
```

Figure 4.15: Write barrier modified to use dual dirty flags. i is the thread index and $d_i \in \{0, 1\}$ is the dirty flag to raise in this collection.

Resetting of dirty flags in DDC-2—see Figure 4.16—is very specific. Only one set of dirty flags gets reset, *i.e.*, the flags raised before handshake one of the current collection.

The set of flags modified since handshake one are not inadvertently reset, so they do not need to be restored. This explains why the procedure in Figure 4.6 is not needed.

```
Procedure DualMode-ResetDirtyFlags
begin
1.   for each $\langle s, old \rangle \in Hist_k$ do
2.       $Dirty_{k \pmod 2}(s) := false$
end
```

Figure 4.16: Procedure **ResetDirtyFlags** modified to use dual dirty flags. k is the collection number.

The key idea in this approach is the following: when raising and resetting dirty flags, the collector and mutators only operate on one set of dirty flags, the dirty flags relevant for the current collection. When *checking* the dirty status of a pointer, as in the write barrier of Figure 4.15, both dirty flags are checked. Checking both flags is necessary to know whether the pointer is dirty, from the current collection or from the previous collection. Besides, raising and resetting only one set of flags ensures that there is no interference between collections that can invalidate reference count updates.

Prior to resetting dirty flags of pointers in collection k, for each pointer s in the system, if

$$Dirty_{k \pmod 2}(s) = true$$

then $s \in Hist_k$. After resetting dirty flags of pointers in collection k, for all pointer $s \in Hist_k$,

$$Dirty_{k \pmod 2}(s) = false.$$

Thus, after resetting dirty flags of pointers in collection k, for all pointer s in the system,

$$Dirty_{k \pmod 2}(s) = false.$$

This observation helps us realize that in order to complete DDC-2 we only need to make small changes to the rest of DDC. In particular, we only need to modify the procedures in

Figure 4.8 and Figure 4.13. Both line 4 in Figure 4.8 and line 3 in Figure 4.13 become

$$\ldots\ if\ \neg Dirty_{k+1\,(\mathrm{mod}\,2)}(s)\ then\ \ldots$$

These changes are adequate since the collector knows that for all pointer s in the system,

$$Dirty_{k\,(\mathrm{mod}\,2)}(s) = false.$$

4.6 The Levanoni-Petrank Collector

Consider DDC as presented in Section 4.3 and Section 4.4. Although it is not our goal to transform DDC into a four-handshake collector, the Levanoni-Petrank Collector [57, 58], LPC, uses four handshakes. Before explaining the necessity for the additional handshakes, we describe the differences between LPC and DDC.

```
Procedure **RestoreConflicts**
begin
1.    $ConflictSet_k := \emptyset$
2.    // handshake 2 of ILPC-4
      for each thread $T_i$ do
3.        suspend $T_i$
4.        $ConflictSet_k := ConflictSet_k \cup Buf_i$
5.        resume $T_i$
6.    for each $s \in ConflictSet_k$ do
7.        $Dirty(s) := true$
8.    // handshake 3 of ILPC-4
      for each thread $T_i$ do
9.        suspend $T_i$
10.       nop
11.       resume $T_i$
end
```

Figure 4.17: Procedure embraces handshakes two and three LPC [57, 58].

1. We chose different names for some of the procedures in LPC.

2. LPC uses the procedure in Figure 4.2 as its single write barrier.

3. LPC replaces Figure 4.7 with 4.18.
4. LPC does not use procedure **RestoreDirtyFlags** of Figure 4.6.
5. LPC uses DDC's handshake two as its handshake four.
6. LPC requires two additional handshakes, handshake two and handshake three, which are included in Figure 4.17.

```
Procedure Consolidate*
begin
1.    local $Temp := \emptyset$
2.    $Locals_k := \emptyset$
3.    // handshake 4 of ILPC-4
      for each thread $T_i$ do
4.       suspend $T_i$
5.       $Snoop_i := false$
6.       // retrieve snooped objects
         $Locals_k := Locals_k \cup Locals_i$
7.       // give $T_i$ an empty $Locals_i$ buffer
         $Locals_i := \emptyset$
8.       // copy thread local state and ZCT
         $Locals_k := Locals_k \cup State_i$
9.       $ZCT_k := ZCT_k \cup ZCT_i$
10.      $ZCT_i := \emptyset$
11.      $Temp := Temp \cup Buf_i$
12.      resume $T_i$
13.   $Hist_{k+1} := \emptyset$
14.   local $Handled := \emptyset$
15.   // consolidate $Temp$ into $Hist_{k+1}$
      for each $\langle s, old\rangle \in Temp$ do
16.      if $s \notin Handled$ then
17.         $Handled := Handled \cup \{s\}$
18.         $Hist_{k+1} := Hist_{k+1} \cup \{\langle s, old\rangle\}$
end
```

Figure 4.18: Procedure consolidates all the mutator local buffers into per-collection buffers. * This procedure is the same as handshake four of LPC [57, 58].

LPC targets a multi-threaded, multiprocessor environment. As such, it is possible for multiple mutators to execute the write barrier concurrently. No restriction is placed on which mutator may use the write barrier at a particular time. Consequently, it is possible for mutators affected by handshake one to execute the write barrier concurrently with mutators that have not yet been affected by the same handshake. Should that be the case, the potential problem is that both sets of mutators log pointers in their local buffers. Since each pointer is associated with a single dirty flag, when the dirty flags of pointers are reset by procedure **ResetDirtyFlags** in Figure 4.5 some pointers in thread local buffers will have their dirty flags reset. Since this result is undesirable for fear that it can foil the collection, handshake two of Figure 4.17 is used to restore the dirty flags of the affected pointers. Handshake three ensures that the restoration is visible to all application mutators.

4.7 Validating the DDC family of collectors

We have presented several variations of an on-the-fly referencing counting garbage collector for Java. We now show that the DDC family of collectors is correct.

Levanoni and Petrank [58] have proved that LPC is correct. We use their results to show that the DDC collectors are correct. Our approach involves demonstrating that it is safe to replace handshakes two and three of LPC with the components of the DDC collectors that make them different from LPC.

We make two assumptions that are essential for the correctness of the algorithms.

1. The reads and writes of the dirty flags in the write barriers are not reordered, *i.e.*, they are executed in the order they appear.

2. The raising of the snoop flag for each mutator is visible to the associated mutator before it actually begins handshake one.

4.7.1 Definition of concepts

Collections Let Col_k denote collection k, the current collection. The previous collection is thus denoted by Col_{k-1} and the next collection by Col_{k+1}.

Buffer writing Mutator T_i writes to $Buf_{i,k}$ if T_i logs $\langle s, oldvalue\rangle$ in Buf_i during collection k. However, if T_i logged $\langle s, oldvalue\rangle$ in Buf_i during Col_{k-1}, we say T_i wrote to $Buf_{i,k-1}$.

Dirty flag Every pointer s is associated with at least one dirty flag denoted by $\mathit{Dirty}(s)$.

Handshakes $\mathit{HS}_m(k)$ denotes handshake m of collection k, where $1 \leq m \leq 4$.

4.7.2 The Defoe-Deters Collector is correct

We prove that DDC is correct by first establishing some characteristics of LPC. We then show that it is safe to replace handshakes two and three of LPC with the components of DDC that make DDC different from LPC. In particular, we have added a second write barrier to DDC and a procedure to restore dirty flags. See Figure 4.3 and Figure 4.6, respectively. We have also replaced handshake four (Figure 4.18) of LPC with a shorter handshake namely, the one in Figure 4.7. We show that these modifications are sufficient to eliminate handshakes two and three of LPC and do not break the collector.

Axiom 4.7.1 *Procedure* ***Update*** *in Figure 4.2 and procedure* ***Instantiate*** *in Figure 4.1 are executed as collector-proof code, i.e., code that cannot be suspended by the collector.*

Axiom 4.7.2 *If mutator T_i and mutator T_j concurrently log $\langle s, v_i\rangle$ and $\langle s, v_j\rangle$ in local buffers $\mathit{Buf}_{i,k}$ and $\mathit{Buf}_{j,k}$ respectively, then $v_i = v_j$.*

Axiom 4.7.1 is fundamental to all three members of the family of on-the-fly collectors. It ensures that logging completes and that dirty flags are properly updated. Axiom 4.7.2 is also fundamental since it allows the collector to pick as its Hist_k entry for pointer s any log entry from any mutator whose local buffer contain s. Hist_k is the buffer in which the collector keeps the consolidated history of mutator local buffers for collection k.

Lemma 4.7.3 *The Pointers in Hist_k, the consolidated history for collection k, are unique, i.e., for every pointer $s \in \mathit{slots}(\mathit{Hist}_k)$, s appears once. More formally:*

$$s \in \mathit{slots}(\mathit{Hist}_k) \longrightarrow \left|\left\{\langle sl, v\rangle \in \mathit{Hist}_k \,\middle|\, sl = s\right\}\right| = 1 .$$

Proof: The only thread that updates $Hist_k$ is the collector, T_c. T_c updates $Hist_k$ by adding $\langle s, oldvalue\rangle$ tuples from Buf_i for all mutators T_i, $1 \leq i \leq n$, to $Hist_k$. These updates occur in the last handshake of Col_{k-1} and in the first handshake of Col_k.

Notice from Axiom 4.7.2 that when multiple mutators concurrently log a $\langle s, oldvalue\rangle$ tuple in their local buffers, they all log the same *oldvalue*. Notice also that the collector does not add duplicate buffer entries to $Hist_k$. Further, the abstract data structure used to model $Hist_k$ is the set, which does not allow for multiple occurrence of an element. Hence, it follows that pointers in $Hist_k$ are unique. ■

Lemma 4.7.4 *Let P be the point in the collector code that corresponds to the end of execution of $HS_1(k)$. At P, $\forall s \in slots(Hist_k)\,, Dirty(s) = true$.*

Proof: When a mutator executes the write barrier, it raises the dirty flag of every pointer it adds to its local buffer(s). This follows directly from the write barrier code and Axiom 4.7.1. During Col_{k-1}, dirty flags of pointers are reset after HS_1 with procedure **ResetDirtyFlags** of Figure 4.5. It is possible for dirty flags of buffered pointers to be reset for the reasons given in Section 4.6. However, those dirty flags are restored by $HS_2(k-1)$. By $HS_3(k-1)$ the dirty flags of all the pointers in the buffers are raised. Thus, when those pointers are added to $Hist_k$, in $HS_4(k-1)$, the dirty flags of all pointers in $Hist_k$ are raised.

It is also possible for other pointers to be added to Buf_i for each mutator T_i after T_i interacts with the collector in $HS_4(k-1)$. Such pointers are added to $Hist_k$ in $HS_1(k)$. Observe that the dirty flags of those pointers are all raised in the write barrier. Thus, by the end of handshake one of collection k, it follows that $\forall s \in slots(Hist_k)\,,\ Dirty(s) = true$.

■

Lemma 4.7.5 *No pointer with a clear dirty flag has its dirty flag reset.*

Proof: We know from Lemma 4.7.4 that prior to the execution of **ResetDirtyFlags**, the dirty flags of all pointers in $Hist_k$ are raised. We also know that **ResetDirtyFlags** executes after $HS_1(k)$ and, when it does, it resets only the dirty flags of all pointers in $Hist_k$. Since

ResetDirtyFlags is executed once per collection, is the only procedure used to reset dirty flags, and the pointers of $Hist_k$ are unique, it follows that only pointers with raised dirty flags have their dirty flags reset. Thus, no pointer with a clear dirty flag has its dirty flag reset. ∎

Lemma 4.7.6 *The dirty flag of every pointer $s \in slots(Hist_k)$ is reset exactly once during collection k.*

Proof: Lemma 4.7.6 follows directly from Lemma 4.7.3 and the fact that procedure **ResetDirtyFlags** of Figure 4.5 is executed once per collection. ∎

Lemma 4.7.7 *Pointer s has its dirty flag reset after handshake one of collection k only if $s \in slots(Hist_k)$.*

Proof: By the end of $HS_1(k)$ the updating of $Hist_k$ is complete, *i.e.*, no more $\langle pointer, oldvalue \rangle$ tuples are added to $Hist_k$. The collector then resets the dirty flags of all the pointers in $Hist_k$ by executing procedure **ResetDirtyFlags** of Figure 4.5. This is the only time in Col_k in which the collector resets dirty flags. Since the collector only resets dirty flags with procedure **ResetDirtyFlags**, if $s \in slots(Hist_k)$ then the dirty flag of s is reset. ∎

Lemma 4.7.8 *If pointer s whose dirty flag $Dirty(s)$ is reset after $HS_1(k)$ has $Dirty(s)$ raised thereafter, then $Dirty(s)$ remains raised until after $HS_1(k+1)$ when **ResetDirtyFlags** is executed by the collector.*

Proof: The only place in the code where dirty flags are reset is in procedure **ResetDirtyFlags** which is executed once, after HS_1, of each collection. Thus, if $Dirty(s)$ is raised after being reset in Col_k, it remains raised until **ResetDirtyFlags** executes again, in Col_{k+1}. ∎

We have validated important characteristics that are shared by the DDC collectors and LPC. However, we have not yet shown that it is safe to eliminate handshakes two and three. To show that we can safely eliminate these handshakes, we only need to show that our modifications to LPC do not violate the characteristics presented above. We do so with the following theorem:

Theorem 4.7.9 *Handshakes two and three of LPC can safely be eliminated by incorporating the approach presented in Section 4.7.2.*

Proof: Denying every mutator T_i that has already responded to $HS_1(k)$ access to procedure **Update** by the approach presented in Section 4.4 ensures that no pointers reset during Col_k reside in Buf_i. This means that no pointer s in Buf_i needs to have $Dirty(s)$ restored. Pointers that potentially need to have their dirty flags restored are logged in TB_i for each T_i instead with procedure **UpdateTwo**. Procedure **UpdateTwo** is executed only by mutators released from handshake one from the time of their release until handshake one completes for all mutators. Since no mutator T_i is allowed to use TB_i after handshake one completes for all mutators, no more pointers are added to TB_i. The collector restores the dirty flags of the pointers in TB_i for each mutator T_i. It does so without a handshake using procedure **RestoreDirtyFlags**. Procedure **RestoreDirtyFlags** thus replaces HS_2 of LPC.

Since HS_2 can be eliminated safely, so can HS_3. HS_3 is predicated by the presence of HS_2. We only need to show that our modifications to LPC do not violate the characteristics presented above.

Axioms 4.7.1 and 4.7.2 are not violated because LPC's write barrier is not modified. Procedure **UpdateTwo** is very similar to procedure **Update** and is executed as CP-code.

Lemma 4.7.3 is honored by our approach since pointers are added to $Hist_k$ in the same fashion, *i.e.*, during Col_{k-1} and Col_k. The only modification we make in this regard is adding pointers to $Hist_k$ a bit earlier—before the last handshake of Col_{k-1}. Notice that the dirty flags of those pointers are raised before they are added to $Hist_k$. If a pointer already exists in $Hist_k$, it is not added to $Hist_k$ a second time.

Lemma 4.7.4 is not violated for the simple fact that our approach only allows pointers with raised dirty flags to be added to mutator local buffers. Since $Hist_k$ consists of pointers that were in mutator local buffers and pointers that were added before the last handshake of Col_{k-1}, all pointers in $Hist_k$ at the point described in Lemma 4.7.4 have their dirty flags raised.

Lemmas 4.7.7, 4.7.5, 4.7.6, and 4.7.8 are also not violated because our approach does not alter the behavior of procedure **ResetDirtyFlags**, neither does our approach affect when **ResetDirtyFlags** executes. **ResetDirtyFlags** continues to be executed once per collection, after HS_1.

Our approach as presented in Section 4.4 violates none of the characteristics of LPC. Thus, the Defoe-Deters collector is correct. ■

4.7.3 Defoe-Deters Collector Version 2 is correct

In Section 4.7.2 we showed that DDC is a correct collector. We now prove that DDC-2 is also correct.

Theorem 4.7.10 *By using dual dirty flags as an alternative approach to that described in Section 4.4, handshakes two and three of LPC can be eliminated.*

Proof: LPC [57, 58] uses a single dirty flag per pointer. These flags are raised only by mutators executing the write barrier when they modify associated pointers. They are reset only by the collector in procedure **ResetDirtyFlags**, which is executed after HS_1. The key invariant maintained by LPC after HS_2 and HS_3 (and, in fact, the purpose for their existence) is that pointers marked dirty must have been modified after HS_1 (and are therefore logged in a mutator local buffer), and, more importantly, all pointers modified since HS_1 are marked dirty.

DDC-2 maintains two separate sets of dirty flags—one for even-numbered collections and one for odd-numbered collections. Each pointer has two dirty flags, one from each set. When checking pointer dirty flags, DDC-2 observes *both* flags and considers the pointer dirty if *either* flag is raised (and non-dirty only if they are both not raised). When raising a dirty flag, mutators raise the dirty flag appropriate for the relevant collection, Col_k if the mutator has not yet responded to HS_1, Col_{k+1} otherwise. Similarly, when resetting dirty flags, DDC-2 only resets dirty flags for the relevant collection, Col_k.

The key invariant of LPC after HS_3 is satisfied by the DDC-2 design before HS_2 is performed. If a mutator, after responding to HS_1, modifies a pointer and marks it dirty, DDC-2 will never reset that dirty flag during the current collection: DDC-2 resets only dirty flags for Col_k, while mutators (after HS_1) raise dirty flags for Col_{k+1}. Therefore, dirty flags need never be restored (as in handshake two) since modifications after HS_1 are not reset by DDC-2. This invariant is satisfied without performing handshakes two and three; hence, they are unnecessary. ■

4.8 Implementation issues

A number of implementation design decisions need to be made in implementing a Levanoni-Petrank style collector. Here, we explore the space of possible high-performance implementations and discuss our implementation in the **GNU Compiler for the Java^{TM} Programming Language** (GCJ) [40] version 4.1.0, which is bundled with the **GNU Compiler Collection** (GCC) [39].

4.8.1 Reference counting field

In any reference-counting garbage collector, every object is equipped with a reference-counting field, which the collector uses to store a count of references to the object. Although we are only concerned with references from the garbage collected heap, every object still needs to be augmented to include such a field.

We can elect to search for a contiguous collection of spare, unused bits in each object header and use them to store the reference count of the object. However, there is no guarantee that we are able to find enough bits for that purpose. Moreover, even though we may be able to find enough bits in the object layout of a particular implementation of a language to store the object's reference count, different implementations of the same language may lay out objects differently making it difficult to find the same set of bits at the same location in every layout.

We decide to extend the object layout by installing a reference-counting field at the same offset in the object header. This approach increases the storage overhead of an object by the size of the reference-counting field. This is not terrible since we are only using four bits to store the reference count. Additionally, the fact that the reference count is at a fixed offset in the object header makes it very accessible. In Java every object is derived from `java.lang.Object`. Thus, installing the reference counting field in the `java.lang.Object` class fixes its position in every Java object and makes it uniformly accessible to the collector.

4.8.2 Buffer representation

There are many possibilities for buffer representation in implementing LPC-style collectors. The pseudocode in the original papers [57, 58] suggest a *set* data structure. The performance of the write barrier is affected by the selection of a buffer representation, as is the method with which the collector swaps buffers with mutators.

We choose to represent buffers as per-thread arrays of log entries. We maintain two such arrays for each mutator, used alternately in collections: while the mutator logs entries in one array, the collector operates on the other. During the first handshake of a collection, the mutator is instructed to switch to the other array and the collector is guaranteed exclusive access to the previous. In this way, buffer-swapping is a constant-time operation, just a few pointer manipulations. The avoidance of mutator blocking is discussed in Section 4.8.5.

4.8.3 Maintaining dirty flags

We consider two possible ways to store dirty flags for pointers in objects. As discussed in earlier sections, implementations of DDC must reserve one bit to store the dirty flag for each pointer (in an object); implementations of DDC-2 must reserve two bits.

Store dirty bits in pointers

Pointers to Java objects are typically word-aligned, so on 32-bit systems the lower two bits of a pointer are effectively unused (they are always zero). These two bits can be used for storing two dirty bits for a DDC-2 implementation or one dirty bit for a DDC implementation.

This approach has several advantages. There is no object size overhead for dirty flags, and the dirty flags for a given pointer are easy to find. Their placement does not depend on the pointer to the top of the containing object or the containing object's type. However, where hardware support does not exist, the compiler must arrange to mask out these bits on every pointer dereference and every pointer comparison. This is a constant-time operation, but it increases code size and degrades performance due to the large number of such operations. In systems with specialized memory addressing hardware that automatically masks out some parts of dereferenced pointers, this is an ideal approach to storing dirty flags.

Store dirty words in objects

Dirty flags could be placed in an associated data structure and objects could point to them. This correspondence would necessarily be one-to-one, and would require the overhead of at least one additional pointer per object. We find this unsatisfactory, though we admit that such an implementation might be simpler than the one we present. We do not consider that approach further in this dissertation.

The approach we used in our implementation stores dirty flags elsewhere in the object. We pack 32 bits of dirty flags into a *dirty word* and distribute such words throughout objects as necessary. For DDC implementations, each dirty word stores dirty flags for 32 pointers while for DDC-2 implementations, each dirty word stores dirty flags for 16 pointers.

There are severe constraints on the placement of dirty words. Type substitution in object-oriented languages (using a pointer-to-`B` as if it were a pointer-to-`A`, where `B` is a subclass of `A`) requires that the runtime memory layout of a subclass contains, as a prefix, a valid object of superclass type. Strictly speaking, this object "prefixing" is not necessary for type substitution; indeed, in the presence of multiple inheritance (in languages that support it), another approach is necessary, involving adjustment of the `this` pointer by a statically-known offset. However, in Java implementations, and for the first inherited type in languages supporting multiple-inheritance, the method of object prefixing described is generally used. Therefore, once a dirty word is allocated for a class, that word must exist at the same offset in all subclasses.

If we limit ourselves to solutions with as few extraneous, unused bits of dirty-field information as possible ("leftovers" from the final dirty word), this can be achieved in one of two reasonable ways. First, we could place dirty words above the object pointer as depicted in Figure 4.19: object fields are laid out as usual, with superclass fields appearing first; dirty words are laid out at negative offsets from the object base pointer, with superclass dirty words appearing closer to the base and subclass fields appearing at lower addresses. This is an attractive solution. It is simple, and maintains compatibility with traditional object layout schemes.

But the approach has two drawbacks. First, a pointer is separated by some distance from its associated dirty word. They almost certainly sit on a different cache line, and may even reside on different pages (if the allocator carelessly allocates such objects). Further, such a layout complicates some implementations of object-oriented languages (especially

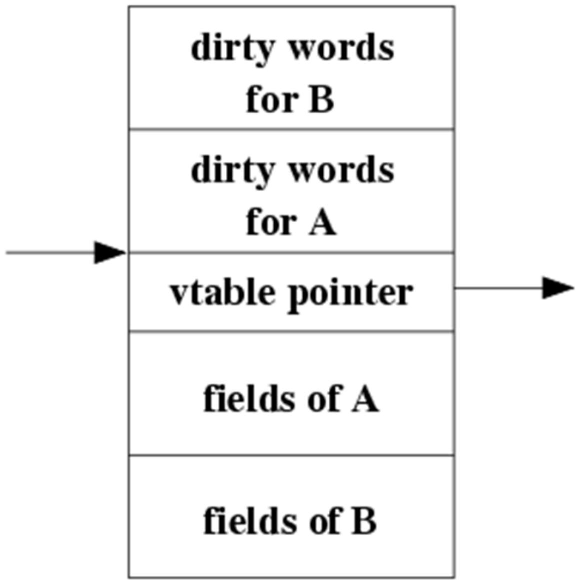

Figure 4.19: Placement of dirty words *above* object pointer. Layout of an object of type `B` is shown (which derives from type `A`). The `A`-part of the object is laid out first, with positive offsets for fields and negative offsets for dirty words. Then the `B`-part of the object is laid out.

implementations of multiple-inheritance languages like C++) in which information of varying length is already laid out above the object pointer, depending on the type of the object or the kind of type inheritance employed.

Another approach is to sprinkle dirty words among the fields of the object as depicted in Figure 4.20. In this approach, dirty flags are placed in a dirty word that appears before all fields for which it contains dirty flags. Whereas for a DDC implementation a dirty word has dirty flags for the next 32 fields of pointer type, for a DDC-2 implementation a dirty word contains dirty flags for the next 16 fields of pointer type. Unused portions of dirty words in superclass layouts are used for subclass fields of pointer type before new dirty words are allocated. The first dirty word of each object is actually only partially used for storing dirty flags. The 4 right most bits accommodate the object's reference counting field. Thus, the first (partial) dirty word holds dirty flags for the next 28 fields of pointer type in DDC implementations, or 14 fields in the case of DDC-2.

Sprinkling dirty words, as needed, among the fields of an object is an excellent approach for storing dirty flags in objects. However, this approach is not ideal for arrays.

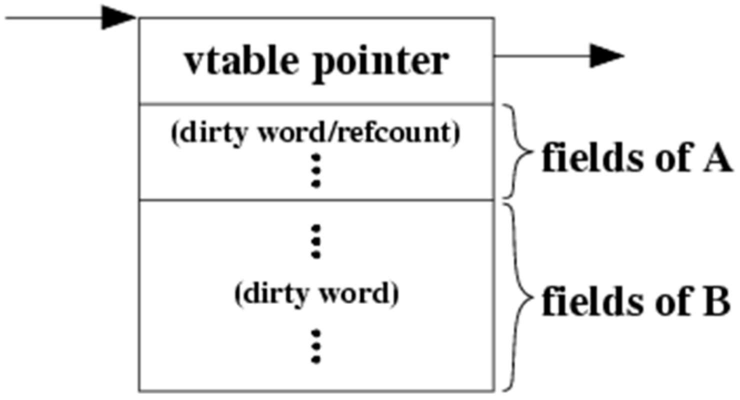

Figure 4.20: Placement of dirty words among the fields of an object. Layout of an object of type `B` is shown (which derives from type `A`). Each part of the object is laid out in turn, with a dirty word injected for every 16 or 32 fields of pointer type.

In an array, the fields or elements are laid out and indexed in a natural way such that the next element appears at the next entry or slot (and index) of the array. Sprinkling dirty words among the elements of the array interferes with this natural layout and complicates the easy access of array elements.

What we do instead is store dirty words at negative offsets, above the top of the array, as described above for objects. Although this approach has some drawbacks, as identified above, it seems to be a more natural way to lay out the dirty words in arrays. Accessing dirty flags would involve computing the (negative) offset of the dirty word associated with an element and accessing the relevant dirty bit(s) in that dirty word.

4.8.4 Pointer modifications in log entries

The LPC algorithm represents each buffer entry produced by the write barrier as a pair $\langle ptr, oldvalue \rangle$. However, indicating the pointer of interest requires extra information when its dirty flag is not contained in the pointer itself (or at a fixed offset for all pointers). In particular, buffer entries in our implementation are given as a 4-tuple $\langle ptr$, $dirty\_word$, $dirty\_bit$, $oldvalue \rangle$. The dirty word associated with a pointer must be indicated. The relevant dirty bit(s) for the pointer in that dirty word must also be indicated.

This approach has the effect of doubling the size of each buffer entry. Although this buffer overhead seems a bit much for a pointer update, it is not too costly as buffer space is recycled between collections. One way to keep buffer overhead low is to use the pair $\langle ptr, oldvalue\rangle$ as Levanoni and Petrank [58, 57] did. But doing so would require storing dirty flags in the unused bits in the pointers themselves, an approach that is potentially more expensive for architectures without specialized hardware support for masking out dirty bits, as noted in Section 4.8.3.

4.8.5 Non-blocking write barrier and handshaking mechanism

Much care must be taken in data structure selection and development of the write barrier. The write barrier must be safe, yet non-blocking.

In our implementation, the write barrier does not perform any locking, and the collector may perform operations in parallel. Some of these operations, during handshakes, are unsafe and may lead to the collector having an incorrect view of the mutator's state. This *unsafety*, however, is rare, and more pertinently, it is detectable in the collector. When detected, the collector is able to undo and repeat the handshake until a safe handshake is performed. *Mutators never loop in our write barrier implementation*, and such collector performance degradation will generally be invisible to the mutators. They may in certain circumstances observe a longer collection phase and thus, a longer time before they are provided a clean buffer.

Such unsafety is not addressed in LPC, as generally a mutator is not permitted to execute the write barrier while the collector is performing a handshake with it. In fact, the mutator's execution is suspended entirely. The write barrier in LPC is CP-code, so the mutator suspension is prohibited from occurring in the write barrier. This causes the collector to block; the mutator then blocks when it exits the write barrier. Our design eliminates mutator blocking, but effectively causes the collector to block until a safe handshake can be performed. In our implementation, we have removed the overhead of scheduling decisions, and even that of unnecessary system calls. What we provide is a primitive *try-retry* mechanism that effectively takes the place of a heavier lock.

4.8.6 Root scanning

When the reference count of an object gets to zero, that object becomes a candidate for collection. If such an object is not a root object, as described in Section 4.3, before it is actually collected the mutator stacks and data segment (including registers) must first be scanned for references to it. Such an object is only collected when there are no references to it.

Scanning data segment

The data segment contains global variables, register information, and other information global to the application. The collector determines (once) the start and end of the data segment and *conservatively* scans that segment (once per collection) for pointers to Java objects. We use the concept "conservative" in the sense that the runtime system conservatively determines pointer. The compiler is privy to type information and is easily able to differentiate pointers from non-pointers; however, the runtime system is ignorant of such information. Consequently, the runtime system treats any entry in the data segment that looks like a pointer as a pointer. More specifically, if the least significant two bits of a data segment entry are zero and the entry either falls within the heap address range or looks like the address of a large object, then it is treated as a pointer to a Java object.

Most Java objects are typically allocated in the garbage collected heap although very large objects, including large arrays, are treated differently. Such objects are stored in a large object space, separate from the heap.

Scanning mutator stack

Each mutator is equipped with a local stack in which it stores local variables for methods or functions it executes. Each function's local data are stored in the stack frame associated with that function. Frames on mutator stacks may contain references to heap objects or objects allocated in the large object space. Consequently, each mutator's stack must be scanned once per collection for such references. Several approaches can be used to scan these stacks.

One approach to stack scanning involves suspending each mutator in turn, as in a handshake, and having the collector scan the stack of the suspended mutator. This approach would require making scheduling decisions on when to suspend and resume each mutator, issuing system calls to actually suspend and resume each mutator, preserving processor affinity for each mutator, and potentially swapping out mutators from their preferred processors so the collector can steal cycles from them to scan their stack. Mutators would have to be swapped back in, of course, when the collector is through scanning their stacks. Alternatively, the collector could run on its own processor while it scans mutator stacks; however, mutators would be idle while their stacks are being scanned.

Another approach is to allow each mutator to incrementally scan its stack each time it executes a function and to keep track of the stack frame associated with each function call. That method, which at face value appears to have good real-time characteristics, would incur unnecessary overhead since the stack grows and shrinks as mutators execute and return from functions. Procedures would take longer to execute and a lot of unnecessary scanning would be done.

The approach we use in our implementation is to allow mutators to execute as usual and to allow the collector to signal them to scan their own stack exactly at the point where stack scanning is needed. Although the collector effectively steals cycles from each mutator, no mutator is ever suspended and the mutator is shielded from what is actually happening. The approach seems complicated but it is easily implemented using signals.

4.9 Experimentation

In this section we report measurements we collected from running an instrumented collector that generates statistics for measurable features of interest. Our primary instrumentation goal was to study the behavior of the write barrier and the handshaking mechanism. To achieve realistic results, we compiled the collector with its *internal* debugging turned off (not to be confused with the debugging done by GCC) and various flags that tell the collector to generate statistics turned on. That way, executions are realistic and desired statistics are still generated. We used two Java programs, namely SortNumbers [37], and SimpleThreads [86] to benchmark the collector.

SortNumbers This single threaded application demonstrates how to sort numbers using selection sort.

SimpleThreads This application illustrates the interactions between two threads. The main thread is common to every Java application. It creates and starts a new thread (the child thread) then waits for it to complete its assigned task (in this case the child thread prints an array of strings to the screen, one at a time). If the child thread takes too long to complete its task, the main thread interrupts it.

4.9.1 Configuration and compilation

The experiments were performed on a Dell Precision 530 workstation with the following system specification:

- Processor: Two physical Xeon 2.4 GHz CPUs with Intel's hyper-threading that supports two threads per CPU.
- Memory: 512MB physical memory and 2GB of swap space.
- Operating System: Debian distribution with Linux kernel 2.6.
- Heap: The heap size specified for each run of the benchmarks is 32MB.

- Clock: The exact clock tick as reported in `/proc/cpuinfo`) is 2392.791 MHz. This is equivalent to 0.418 *ms* (milliseconds) per tick.

We configured GCC (and GCJ version 4.1.0) for compilation with the following options.

```
> home/gcc-repo/configure --prefix=home/gcc-prefix
--enable-languages=c,c++,java
```

This tells the build process to build compilers for the C, C++ and Java programming languages and to store the binaries in `home/gcc-prefix`. To build the binaries we run the following two commands.

```
> make BOOT_FLAGS='-g -O0' CFLAGS_FOR_TARGET='-g -O0'
  CXXFLAGS_FOR_TARGET='-g -O0' GCJFLAGS='-g -O0'
> make BOOT_FLAGS='-g -O0' CFLAGS_FOR_TARGET='-g -O0'
  CXXFLAGS_FOR_TARGET='-g -O0' GCJFLAGS='-g -O0' install
```

The `'-g -O0'` options are to build with debugging turned on and with optimization turned off. This is important since enabling various optimizations can make it difficult to follow exactly how the collector is working.

To compile and execute programs with a version of GCJ that uses our collector as the sole garbage collector, we run the following commands.

```
> setenv LD_LIBRARY_PATH home/gcc-prefix/lib/
> home/gcc-prefix/bin/gcj --main=SimpleThreads -o SimpleThreads
  SimpleThreads.java
> ./SimpleThreads  4 -ms=32MB
```

The first command sets the `LD_LIBRARY_PATH` environment variable to the directory where the dynamic linker can find the standard Java libraries. This command needs to be executed once. The second command compiles the SimpleThreads program and the third command executes it. For our experiments we run each program more than once with different parameters.

4.9.2 Collector Overhead

We used our implementation of DDC to investigate the overhead associated with our collector. To estimate such overhead, we ran the SortNumbers application on the Dell system 100 times with and 100 times without our collector. For each run of SortNumbers, 1000 randomly generated doubles were sorted and the results were printed on the screen. We used the UNIX command `/usr/bin/time` to time the executions. The results from this experiment are given in Figure 4.21.

	Without DDC collector	With DDC collector	overhead	% overhead
Time (s)	15.91	16.91	1	6.29

Figure 4.21: Overhead for executing SortNumbers 100 times with and without our collector.

The overhead associated with the DDC collector seems small. From the experiments described above it is only 6.29 %. This is low considering the amount of work performed during garbage collection. This low overhead may be atributed to the observation that the Dell computer's configuration is ideal for the number of threads present in the SortNumbers application and the collector—the Dell system has two physical processors and the application and the collector together have a total two of threads.

To gain insight into the overhead distribution in the DDC collector, we used the x86 assembly language instruction `rdtsc` to count the number of clock ticks for each collector operation we desired to measure, namely the write barrier and the handshaking mechanism. Each clock tick measures 0.418 *ms*. Figure 4.22 records the measurement for DDC's overhead distribution.

	clock tick count	Time (μs)	Time (s)	% overhead
Handshake Time	205332	85.813	$8.581 * 10^{-5}$	0.009 %
Write Barrier Time (logging)	1082804796	452527.946	0.453	45.253 %
Write Barrier Time (no logging)	403531284	168644.601	0169	16.865 %

Figure 4.22: Overhead distribution for DDC when executing SortNumbers 100 times with the collector enabled. The figure gives total time for each operation for the 100 runs of the application.

As illustrated in Figure 4.22, there is a significant difference between the handshake overhead and the write barrier overhead. Even between the two paths through the write barrier (see Section 4.3) there is much difference between overhead values. To better understand why the difference between these overheads is so significant, we performed further experiments and analyzed their results below, in Section 4.9.3.

Notice also from Figure 4.22 that 37.873 % of overhead is not accounted for. This overhead is distributed among the other operations of the collector. We do not explore the exact distribution of this 37.873 % overhead further because the other operations of the collector are not the main contribution of this research. The main contributions of this research are the non-blocking write barrier and handshaking mechanism.

4.9.3 Investigation of Overhead

To better understand the distribution of the overhead associated with the DDC collector, we performed further experiments. In particular, we ran the SortNumbers application (as described above) 100,000 times and for each run we timed and counted the number of handshakes and write barrier executions for each path through the write barrier. We also ran the same experiments with the SimpleThreads application to help us determine whether the overhead distribution differs when multiple threads are involved. Here, we report and analyze the results of these experiments. Additional results are provided in Appendix A.

We compare the mean time per handshake with the mean time per write barrier execution for each path through the write barrier to determine whether such comparison accounts for the difference in overhead distribution. We compute the time per operation by counting the number of executions of each operation, measuring the total time spent in each operation, and dividing the total time by the count. Figure 4.23 summarizes our results.

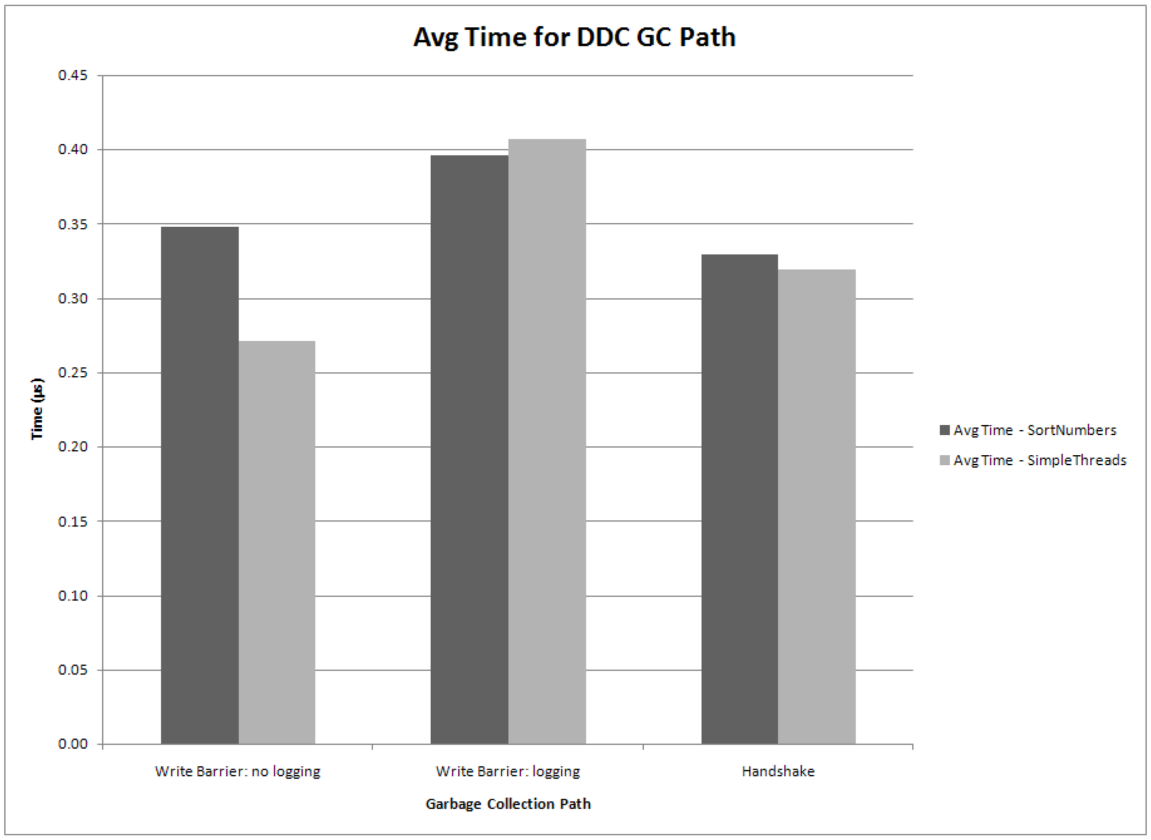

Figure 4.23: Average time cost for an operation, observed for the 100,000 runs of the application when the collector is enabled and the operation is executed.

In both Figure 4.23 and Figure A.1 we observe that for each application, the average time for the non-logging path through the write barrier is smaller than the same for the logging path, as expected. We also observed that the number of mutators (one for SortNumbers and two for SimpleThreads) does not affect the time spent in a write barrier path. Minimum write barrier times depicted in Figure A.5, in Appendix A seem to support these results.

The non-blocking write barrier and handshaking mechanism described in Section 4.8.5 pay dividends, as evidenced by Figure 4.23 and Figure A.1. The handshake runs as fast as the write barrier and neither the collector nor the mutators block. The fast handshakes can be attributed to two reasons. First, mutators are never suspended or blocked when executing in the write barrier. Such blocking can be expensive if system calls are made to suspend and resume mutators. Blocked mutators also fail to make progress and throughput suffers. The other reason that the handshake is fast is that the collector does not use any heavy locks or other heavy synchronization mechanism to perform a handshake. Instead, the collector uses a *try-retry* mechanism that effectively takes the place of a heavier lock. Although the collector may momentarily block on a handshake, we did not observe such blocking in our experiments. Levanoni and Petrank [58, 57] noted that the write barrier execution is fast. Experimental results from our implementation suggest that the handshaking mechanism can be just as fast.

Since the average (and minimum) time overhead for a handshake is comparable to the average (and minimum) time overhead for a path through the write barrier, we calculated the standard deviation of the time overhead for each operation to determine how spread out these overhead values are. Figure 4.24 and Figure A.2 summarize this statistic. These figures illustrate that for each application, the non-logging path through write barrier is the most variable[1] path through the garbage collector while the handshake is the least variable path, as expected. Although we expected the handshaking path through the garbage collector to be the least variable (since the collector never blocked in the experiments), we were surprised that the logging path through the write barrier was less variable than the non-logging path. The logging path incurs more memory traffic than the non-logging path since data is buffered on behalf of the garbage collector. Memory reads and writes can be expensive, especially in the presence of page faults and cache inconsistency. Further investigation is required to determine why the non-logging path through the write barrier is more variable than the logging path.

[1]By variable we mean the overhead values are spread out.

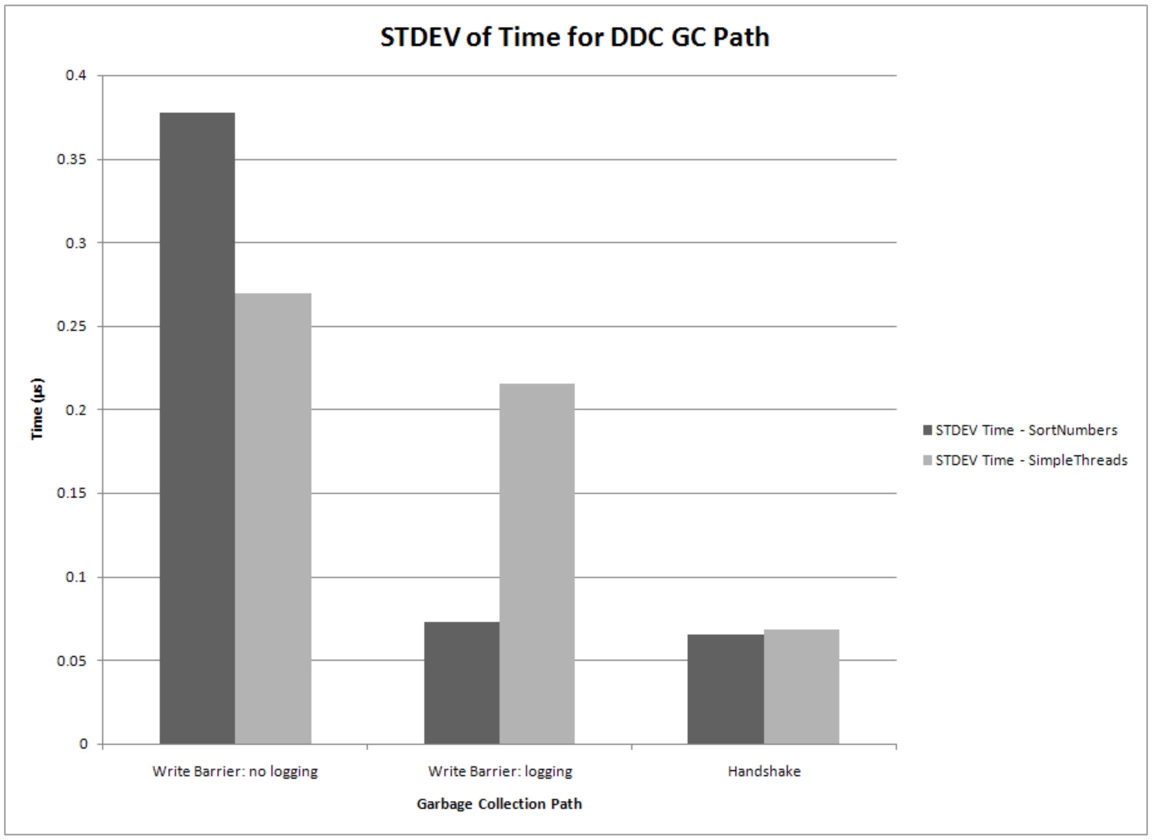

Figure 4.24: Standard deviation of the time cost for an operation, observed for the 100,000 runs of the application when the collector is enabled and the operation is executed.

In Section 4.9.2 we noted that there was a noticeable difference between the overhead distribution among the two write barrier paths and the handshake, but we did not know why. We conducted more experiments to help us appreciate the overhead distribution. We compared per-handshake overhead with per-write-barrier overhead and concluded that the per-handshake overhead is comparable to the per-write-barrier overhead. The non-logging path through the write barrier is faster than the logging path, as expected. So the question remains: why the difference in overhead distribution so significant? To answer this question, we counted the number of times each operation was executed during the 100,000 runs of each application. The results are summarized in Figure 4.25 and Figure A.3. These figures illustrate that there are about four orders of magnitude more write barrier executions when

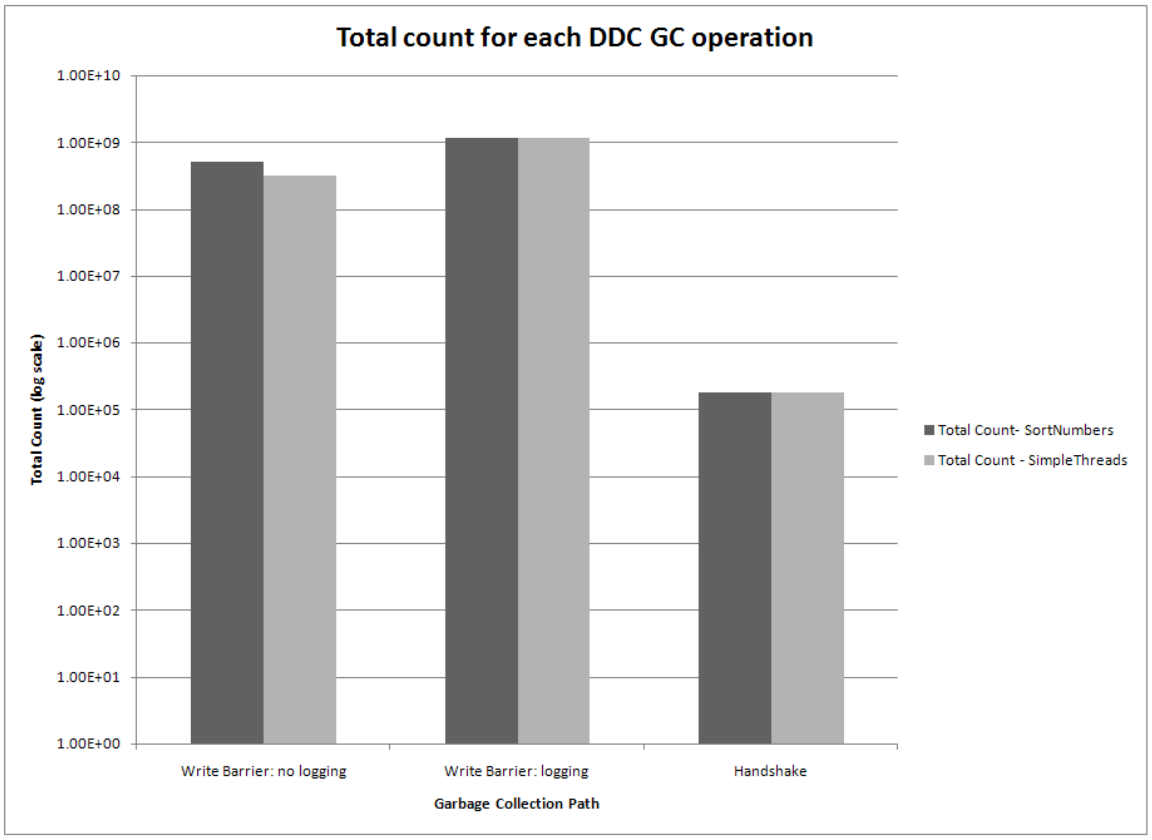

Figure 4.25: Total count of the number of times each operation executed, observed for the 100,000 runs of the application when the collector is enabled and the operation is executed.

the logging path is taken compared to handshake executions and at least three orders of magnitude more write barrier executions when the non-logging path is taken compared to handshake executions. The number of write barrier executions when the logging path is taken is also larger than the number of write barrier executions when the non-logging path is taken. These numbers account for the disparity between write barrier and handshake overhead presented in Section 4.9.2.

4.9.4 Comparing DDC with LPC

The write barrier of DDC and the write barrier of LPC are very similar. The only difference between these write barriers concerns dirty flag updating. In DDC one of two dirty flags is updated, whereas in LPC only one dirty flag is ever updated. Consequently, the overhead associated with the write barrier execution in DDC is comparable with the same in LPC.

To compare the handshaking mechanism of the two collectors, we need to estimate the cost of executing a handshake in LPC. During a handshake in LPC *e.g.*, handshake one, the collector suspends each mutator in turn, copies its buffered entries to a global consolidated history buffer, and resumes the mutator. During a handshake in DDC the collector swaps out each mutator's buffer and swaps in a new buffer without suspending the mutator. In order to use DDC's handshakes to estimate LPC's average handshake overhead we multiply DDC's minimum handshake overhead by the number of write barrier executions when the logging path is taken and divide that product by the number of handshakes. This works out to be 711.27 μs for the SortNumbers application and 714.36 μs for the SimpleThreads application — three orders of magnitude difference between the collectors in each case. Figure 4.26 gives more details.

Clock Tick Count		
	DDC's Handshake	LPC's Handshake
AVG Time - SortNumbers	786.9382466	1701920.629
AVG Time - SimpleThreads	764.2954444	1709303.41
Wall Clock Time (μs)		
AVG Time - SortNumbers	0.328878806	711.27
AVG Time - SimpleThreads	0.319415881	714.36

Figure 4.26: Comparison of LPC's average handshake time (per handshake) with DDC's average handshake time, observed for the 100,000 runs of the application when the collector is enabled and the handshake mechanism is used.

Figure 4.26 illustrates that the handshaking mechanism in LPC is at least three orders of magnitude more costly than the similar mechanism in DDC although these costs do not factor in overhead associated with mutator suspension and resumption (mutator blocking). DDC uses two handshakes while LPC uses four. This, combined with the observation that the write barrier cost for the collectors is comparable results in a shorter per-collection time for DDC relative to LPC. Moreover, the short handshakes, the non-mutator-blocking write barrier and handshaking mechanism, and the fewer handshakes push

DDC in the direction of designing, implementing, and deploying multiprocessor real-time garbage collectors.

4.10 Related work

The notion of reference-counting garbage collection dates back to the early nineteen-sixties when Collins [26] developed a method for the erasure of lists in LISP programs. Since its inception, reference counting has been adopted by several systems: examples include early versions of Smalltalk [42] and InterLisp; Modula 2+ [30]; SISAL [19]; and the Unix utilities awk [1] and perl [90].

4.10.1 Deferred Reference Counting

The overhead incurred in adjusting reference counts is high, even though the cost of reference counting may be amortized over the entire computation [53]. This makes reference counting a less attractive option for memory management than a tracing collection [46]. To reduce reference-counting overhead and make reference counting a more attractive memory management option, **Deferred Reference Counting** (DRC) was introduced [31]. The idea behind DRC is based on the observation that most reference count adjustments are due to stores in local pointers (in stacks and registers). Keeping account of such pointers is very expensive and is not necessary for the correctness of the reference counting algorithm. Instead of keeping account of all such pointers, DRC was invented to keep account of pointers only in the heap. The resulting reference count is denoted *heap reference count.*

Prior to the introduction of DRC, objects were collected as soon as their reference counts dropped to zero. In DRC, objects are not collected in such fashion. Instead, objects with zero reference counts are placed in a **Zero Count Table** (ZCT) that is recycled periodically. Typically, objects in a ZCT are collected at the end of a collection if there are no local pointers to them. If an object in a ZCT receives a reference from a heap object, its reference count is incremented and it is removed from the ZCT. The collectors we discussed in this chapter employ DRC.

4.10.2 Limited-field Reference Counting

Another notable contribution to the reference-counting algorithm is the notion of limited-field reference counting. The idea is based on the observation that reference counting grows every object by a field large enough to hold the maximum number of potential references to it. In theory that field can be as large as a pointer. Thus, the space overhead can be significant for small objects. To minimize this overhead, limited-field reference counting is employed. This technique suggests the use of a few bits (one or more) in the object header to store its reference counting field. If the reference counting field overflows, a tracing collector or some other algorithm is used to restore it.

More radically, researchers have suggested restricting the reference-counting field to a single bit [96, 82, 24, 95]. The reference-counting bit simply determines whether an object is shared or unique [53]. The goals of *One-bit Reference Counting* are to delay garbage collection as long as possible and to reduce the storage overhead of reference-counting garbage collection to that of the mark-sweep algorithm.

The collectors described in this chapter are not One-bit Reference Counting Collectors, but limited-field reference counting collectors. Four bits are used to hold reference counts in the family of DDC collectors.

4.10.3 Multi-threaded Multiprocessor Reference Counting

When Collins [26] designed the classical reference-counting collector, he designed it to work on systems that were available at the time, *i.e.*, uniprocessor systems. In the last few decades multiprocessor systems have proliferated. As such, garbage collection techniques including reference-counting techniques have been developed to leverage such systems.

Concurrent Collection

Once class of garbage collection techniques that comes to mind is the class of concurrent garbage collectors (see Section 1.2). Since the list of such collectors is long we refer the reader to a few [81, 4, 30, 32, 71]. These collectors execute in parallel with mutators, for the most part, and perform best in multi-threaded environments. One shortcoming of such collectors, though, is that there is a period (usually at the beginning or end of a collection) during which they suspend all mutators at roughly the same time to compute a snapshot of the heap. Even though in most cases a snapshot of just the interesting portions of the heap is computed, scalability becomes an issue as the number of mutators increases.

On-the-fly Collection

To overcome the drawbacks of concurrent collectors, on-the-fly collection [7, 11, 58, 57] was proposed. Section 1.2 provides a description of on-the-fly collectors. The class of DDC collectors discussed in this chapter is based on LPC [58], which uses the notion of a sliding view (see Section 2.2.2) instead of a snapshot. The improvements offered by the DDC collectors were noted in previous sections and are reflected in the conclusions.

Chapter 5

Taxonomy of Garbage Collectors

Chapters 1 and 2 gave an overview of the contributions made by the garbage collection community to the field of memory management, especially dynamic memory management. The proliferation of dynamic memory management was also highlighted. Moreover, mention was made of the current and future trend in dynamic memory management—a trend toward concurrency. A brief history of garbage collection exposes the motivation of our predecessors and puts our work in context; we now present a taxonomy of garbage collectors.

5.1 Introduction

Researchers have performed tremendous work on designing, implementing, and evaluating garbage collection algorithms. Their work is essential since it produces tools that automate the reclamation of objects when they are no longer reachable in a program. Those tools save developers time and effort in addressing memory management issues explicitly. Memory management is a complex and delicate process that requires expertise and careful attention.

In addition to designing, implementing, and evaluating garbage collection algorithms, researchers produce bibliographies, comparisons, surveys, and reviews of garbage collectors [93, 94, 53, 52]. While these contributions are noteworthy, little or no work has gone into producing a taxonomy of garbage collectors. Producing such a taxonomy can serve as a tool to help developers determine the most appropriate collectors for their applications. They only need to be aware of certain features of their applications in order to decide which collectors are most appropriate for those applications. Examples of such features include allocation behavior, degree of computation, presence of cyclic data structures, longevity of objects' liveness (or object mortality rate), and *curspace*. Each of these application features plays a role in determining the most appropriate garbage collector for managing the memory of an application. The idea of curspace is not prevalent in the literature; thus, we first give a formal definition for curspace.

5.1.1 Curspace

Curspace refers to the number of bytes occupied by live objects at a point in time t, $starttime \leq t < endtime$, where $starttime$ is the time the program starts executing and $endtime$ is the time the program stops executing. Curspace increases over time with object allocation and decreases with object mortality. Some collectors are not adversely affected by curspace, but certain collectors experience degraded performance as curspace increases. That effect can be costly for embedded systems since their memory footprint is limited.

Curspace is by definition a number of bytes; as such, curspace at time t, cur_t, can be measured by

$$cur_t = \sum_{i=1}^{n_t} sizeof(obj_{t,i})$$

where n_t is the number of live objects in the system at time t and $obj_{t,i}$ represents the ith such object.

Although a collector cannot explicitly control the curspace in an application, a small curspace facilitates garbage collection. Unless an application allocates an excessive number of large objects (which is unlikely) a large value of curspace implies a large number of live objects. Tracing collectors, or collectors with complexities proportional to the number of live objects are seriously impacted by a large curspace. Such collectors incur increased time overhead and reduced throughput since tracing a large number of live objects can be costly.

5.2 Taxonomy category list

We present a taxonomy of garbage collectors, `GC-Tax`, with the following categories. Although this list is not exhaustive, it captures many of the concerns shared by the garbage collection and software development communities.

5.2.1 Incrementality

A garbage collector displays *incrementality* if it performs garbage collection in small increments of time, between which it is suspended so the mutator can progress. Incrementality calls for an interleaving between mutator and collector executions. Each is expected to make progress during the time quantum in which it executes.

Another perspective on incrementality is the idea that the mutator does work on behalf of the collector when it executes. After some prescribed condition is met the collector takes over and completes the collection work in small increments of time.

Yet another perspective on incrementality is the notion that collection work is deferred until some time t when the collector performs collection in small increments of time. Prior to t the collector may negotiate with the mutator to have the mutator do work on its behalf. While these perspectives appear different to a certain extent, the common theme that ties them together is that collection work is done in small increments of time between which the mutator experiences progress. This theme is what makes a collector incremental. Incrementality may be good for real-time systems if garbage collection increments can be bounded and a real-time system can budget for garbage collection. However, if the increments are too short, they may need to be very frequent to keep up with storage demands.

To measure the incrementality of a collection or the average incrementality of n collections for a given run of an application when the collector is managing its memory, we provide the metrics

$$inc_i = \frac{c_i}{t_i}$$

$$\overline{inc} = \frac{\sum_{i=1}^{n} \frac{c_i}{t_i}}{n}$$

where c_i is the number of garbage collection increments in collection i, and t_i is the time to complete collection i. t_i includes mutator time when the mutator and collector interleave their execution. However, when the mutator monopolizes the CPU for an extended amount of time, such mutator time is not included in the computation of t_i. In the metrics above, we have illustrated how to compute absolute and average values of incrementality. Minimum or maximum values of incrementality can also be computed by sorting the values of inc_i, where $1 \leq i \leq n$, and the collector is used to manage memory for a complete run of an application.

Observation of the incrementality metrics reveals that they are biased toward collectors that execute a number of collections in cycles (like LPC and the DDC family of collectors). Careful examination suggests that they can easily be adapted to suit other collectors. For example, consider the metrics

$$Inc = \frac{c}{t}$$

$$\overline{Inc} = \frac{\sum_{i=1}^{N} \frac{c}{t}}{N}$$

In this case Inc, c and t are not per-collection parameters but parameters for the entire run of the collector. N denotes the number of such executions of the collector. High measures of incrementality are desirable for incremental collectors.

5.2.2 Immediacy

If storage is reclaimed immediately as a heap object becomes garbage, the underlying garbage collector exhibits excellent immediacy of garbage collection. Not all collectors reclaim storage immediately; instead, some collectors postpone garbage collection until some preset condition is met. Such collectors trigger garbage collection in either a *work-based* fashion, a *time-based* fashion, or an *exception-based* fashion. Work-based triggering of garbage collection refers to postponing collection until a preset fraction of the heap is consumed. Time-based triggering of garbage collection, on the other hand, refers to postponing collection until the mutator has been running for a certain amount of time. Exception-based triggering of garbage collection allows the collector to wait until a particular exception occurs, *e.g.*, out of memory exception, before the collector attempts to reclaim storage. Applications that take advantage of recycled storage benefit significantly from garbage collectors that collect garbage immediately. Embedded systems also benefit from immediacy of garbage reclamation—they use small memory footprints.

In his master's thesis, Hampton [43] defined *Rot-Time* as "the amount of time that passes between the point in the program at which an object is no longer reachable and the point at which the garbage collector is able to collect the object". Although he uses the concept of Rot-Time in his thesis, Hampton is actually defining immediacy of garbage collection. We adopt his definition of Rot-Time as the standard definition of immediacy. Immediacy can be measured in wall clock time or processor clock cycles. If Imm_j denotes the measure of immediacy for object j then the measure of the average immediacy for m objects is given as

$$\overline{Imm} = \frac{\sum_{j=1}^{m} Imm_j}{m}$$

where m is the total number of objects collected by the collector. Maximum or minimum values for the measure of immediacy can be computed by considering the values of Imm_j where $1 \leq j \leq m$ and the collector is used to manage memory for a complete run of an application. Applications in which objects must be finalized before they are actually reclaimed may benefit from small measures of immediacy. Such objects hold on to system resources until they are finalized. If they are known to be garbage soon after they become such, they can run their *finalizer* methods and release system resources.

5.2.3 Pause time

On a uniprocessor system, only one mutator executes at a time. The collector cannot execute while the mutator is executing, and *vice versa*. The pause time on such systems is a measure of the interval during which the mutator is suspended so the collector can execute. This notion of pause time also extends to multi-threaded, multiprocessor environments where an entire application may be suspended if the collector needs to compute a snapshot of the heap.

For on-the-fly or concurrent collectors, the notion of pause time is a per-mutator pause time. Such pause time corresponds to the interval during which the collector suspends a mutator to perform some transaction with it. Regardless of processor architecture, short pause times benefit interactive systems, distributed systems, and real-time systems. The exact benefits vary by application domain.

Like immediacy (Section 5.2.2) pause time can be measured in either wall clock time or processor clock cycles. Given that pt_i denotes the measure of pause time i, where $1 \leq i \leq n$ and n represents the total number of pause times, the mean pause time is given as

$$\overline{pt} = \frac{\sum_{i=1}^{n} pt_i}{n}$$

The minimum pause time is given as the minimum pt_i while the maximum pause time is given as the maximum pt_i. Real-time systems are concerned with budgeting for worst-case execution. Consequently, the pause time of interest to such systems is the maximum pause time in a given run of the system. Real-time systems require small, provably bounded, maximum pause times.

5.2.4 Completeness

All dead storage is collected eventually by any complete collector. A complete collector may reclaim all garbage during a single collection or it may require multiple collections. Objects that die or become garbage after a collection starts are usually not detected until the next collection. Such objects require at least one more collection for them to be detected so they can eventually be collected. These objects become *floating garbage* and remain in the system until they are collected. The number of collections needed to guarantee reclamation of an object depends on the collector. While that observation is noteworthy, not all collectors are complete. Incomplete collectors are unable to reclaim some objects regardless of the number of collections they execute. Such collectors are sometimes bundled with a complete collector that executes occasionally to reclaim floating garbage. In environments where pointer determination is conservative, *e.g.*, the Boehm collector [14] managing memory for a C program, the garbage collector is incomplete.

To measure a collector's completeness we need to be able to measure the amount of garbage g_s (in bytes) in the system and the amount of garbage g_c detected and eventually collected by the collector. Immediacy is not included in the measure of completeness. The ratio

$$comp = \frac{g_c}{g_s} * 100\%$$

gives the completeness of the collector. This ratio allows us to reason more thoroughly about the completeness of collectors. To compute the mean, maximum, or minimum completeness of a collector, an application that uses the collector to manage its memory must be run n times; for each run the completeness of the collector is computed. These values should then be used to determine the mean, maximum, or minumum completeness of the collector. A high degree of completeness is good for the collector.

Systems that benefit most from high completeness are systems with small memory footprint, like embedded systems. Real-time systems may also benefit from high completeness since fulfilling real-time guarantees require careful management of limited resources, including memory.

5.2.5 Overhead

While garbage collection alleviates software engineering concerns, it is not free. There are time and storage costs associated with garbage collection. Some collectors are optimized for time while others are optimized for storage. The most cost-effective collector is optimized for both time and storage. A collector optimized for time may incur unnecessary storage overhead and *vice versa*, but a collector optimized for both time and storage tries to minimize both types of overhead.

To measure the time overhead oh_t of a collector we use the expression

$$oh_t = \frac{tot_{wc} - tot_{nc}}{tot_{nc}}$$

where tot_{nc} denotes the total time it takes an application (with a given input) to execute without the garbage collector and tot_{wc} denotes the total time it takes the application to execute (with the same input) with the garbage collector. To measure tot_{nc} garbage collection must be turned off and the application must run with a heap large enough to not require garbage collection. To measure tot_{wc} garbage collection must be turned on and the heap must be large enough to accommodate *maxlive*[1] and any other storage required during collection. Storage overhead oh_s is computed as

$$oh_t = \frac{tos_{wc} - tos_a}{tos_a}$$

where tos_{wc} is the summation of all storage used during a run of the application when garbage collection is enabled and tos_a is the total number of bytes allocated by the application. Real-time systems, distributed systems, and interactive systems benefit significantly from low time overhead while embedded systems benefit most from low storage overhead. Other systems may also benefit from low overhead; however, it is not always practical to minimize both time and storage overhead at the same time.

[1] *maxlive* is the maximum live storage in a program at any instant during its execution.

5.2.6 Concurrency

A collector is concurrent if it executes in parallel with mutators. Concurrency is different from incrementality in that concurrency requires multiple processors. Concurrent execution does not suggest an interleaving of collector and mutator on a shared processor, but the execution of the two at the same time on separate processors. Whereas some concurrent collectors suspend all mutators at the same time to compute a snapshot of the heap, others suspend mutators on an individual basis to perform transactions with them. Perfectly concurrent collectors would never suspend mutators for any reason. At the time of this writing we did not know of the existence of any perfectly concurrent collector. Concurrency in garbage collection, however, is beneficial to multi-threaded, multiprocessor applications.

Appel defined concurrency as the extent to which a collector can do its work in parallel with the mutator [4]. We extend this definition by defining concurrency as the extent to which the collector does its work in parallel with any mutator. This definition gives insight into a metric suitable for measuring concurrency. Concurrency *con* can be measured as

$$con = \frac{tot_p}{tot_c}$$

where tot_c is the total time for which the collector executes and tot_p is the total time for which the collector runs in parallel with any mutator. The *quality* of concurrency, which is the fraction of the total number of mutators the collector executes in parallel with, can also be measured. Here, we are only concerned with measuring the extent to which the collector runs in parallel with any mutator. Computing average, maximum, or minimum concurrency requires multiple runs of the collector for the same application with the same input. A desirable collector is one with a high measure of concurrency.

5.2.7 Throughput

Howe [49] defined throughput as “The rate at which a processor can work expressed in instructions per second or jobs per hour or some other unit of performance.” The implication is that for a given application, collectors that achieve high throughput do not slow down computation. Said otherwise, if the same application is executed on the same system twice with the only variable being the garbage collector, the collector that achieves the higher throughput is the one that lowers the total execution time of the application. In this dissertation we define throughput as the fraction of total execution time consumed by the application. If the total execution time is consumed by the application then the throughput is 1 or 100%. If the collector runs for some time and extends the running time of the application, the fraction of the time during which the collector does not interfere with the application constitutes the throughput achieved with the collector. Having the capability to achieve high throughput is usually a desirable feature of a garbage collector. High throughput means low time overhead and *vice versa.* As a matter of fact, throughput and time overhead have an inverse relation to each other. However,

$$tp = \frac{tot_a}{tot_{wc}}$$

serves as a good metric for measuring throughput where tot_a represents the execution time consumed by the application when garbage collection is turned on and tot_{wc} is defined in Section 5.2.5. Furthermore,

$$\begin{aligned} tot_a &= tot_{wc} - (tot_c - tot_p) \\ &= tot_{wc} - tot_c + tot_p \\ &= tot_{wc} + tot_p - tot_c \end{aligned}$$

Thus, tp can also be rewritten as

$$\begin{aligned} tp &= \frac{tot_{wc} - (tot_c - tot_p)}{tot_{wc}} \\ &= 1 - \frac{tot_c - tot_p}{tot_{wc}} \end{aligned}$$

As is the case with concurrency, calculating the average, maximum, and minimum throughput require multiple runs of the collector. High throughput is usually a desirable feature of a collector.

5.3 Comparing extant collectors with `GC-Tax`

Each category or garbage collection feature in `GC-Tax` is measured empirically with the metric defined for that category. Using a uniform hardware and software platform to measure these metrics for each collector where the same benchmarks and workloads are deployed would be the most appropriate way to use `GC-Tax` to compare extant collectors. Unfortunately, we do not have an implementation for most of the collectors we are interested in comparing. As such, we are not able to instrument the collectors to measure these metrics. Consequently, our best option is to use a subset of the categories in `GC-Tax` to do qualitative comparisons of a few extant collectors including the DDC family of collectors.

We use the lattice Excellent, Very good, Good, Fair, Poor with each garbage collection feature to compare Collins's [26] reference counting garbage collection algorithm, McCarthy's [61] mark-sweep garbage collection algorithm, the Levanoni and Petrank [58, 57] on-the-fly reference counting garbage collector, and our family of collectors. Our results are summarized in Figure 5.1. The following subsections elaborate on the results presented in Figure 5.1.

Features	Reference counting	Mark-sweep	LPC	DDC collectors
Incrementality	Very good	Poor	Fair	Fair
Immediacy	Very good	Poor	Good	Good
Pause Time	Good	Poor	Very good	Very good
Completeness	Fair	Excellent	Fair	Fair
Time Overhead	Bad	Excellent	Good	Very good
Concurrency	Poor	Poor	Very good	Excellent
Throughput	Fair	Excellent	Good	Very good

Figure 5.1: Using `GC-Tax` to compare extant garbage collectors.

5.3.1 Comparing collectors with incrementality

When incrementality is high, pause time is usually low. Such is the typical case with the reference counting collector (RCC). However, when RCC frees large objects or recursively reclaims objects, both its incrementality and pause time suffer. Several approaches, including a work-list approach can be used to incrementalize object collection. But incrementalizing object collection was not a design goal for the original reference counting algorithm.

The mark-sweep collector (MS) is not an incremental collector. MS runs when memory is not available to satisfy the next allocation request. When MS runs, it suspends the entire application thereby pausing the application for the duration of its run. MS touches all live data as it runs. Thus, the application pause can be very long.

LPC and the DDC family of collectors are not incremental collectors in the common case since they were designed with a multiprocessor environment in mind. However, they can function as incremental collectors in a uniprocessor environment.

While high incrementality generally means low pause time, high incrementality can adversely affect time overhead. For every pointer assignment in RCC, two reference count updates must be done. This can increase memory traffic if updates are done frequently, thus unnecessarily adding to overhead.

5.3.2 Comparing collectors with immediacy

RCC generally does a very good job at reclaiming objects as soon as they become garbage. It is a direct approach to garbage collection in the sense that it knows exactly when an object becomes garbage, as soon as its reference count drops to zero. However, when objects reference each other such that they form a cycle, RCC cannot reclaim such objects (because the reference counts for objects in a cycle never get to zero) and its immediacy suffers.

MS does poorly at immediacy because it does not perform garbage collection unless there is not enough memory to satisfy the next allocation request. Objects that become garbage before the collector runs remain as garbage in the system for potentially a long time, thereby consuming memory and increasing storage overhead.

LPC and the DDC family of collectors do reasonably well with immediacy in the absence of cycles. Since they are reference counting collectors, they suffer the same plight as RCC. But in the typical case they do almost as well as RCC in the area of immediacy. The only reason RCC performs slightly better than these collectors is that they reclaim objects at the end of collections, after they update reference counts in objects.

5.3.3 Comparing collectors with pause time

RCC generally has a short pause time except when collecting large objects or when recursively reclaiming objects. In this and previous sections we noted that RCC can suffer in certain areas when recursively reclaiming objects, but we have not yet described what it means to do so. We now elaborate on this phenomenon. When the reference count of an object gets to zero, before it is collected all its pointers are scanned so the objects to which they point can have their reference counts decremented. Such reference count updates can potentially lead to other objects being collected before the initial object is actually collected.

The pause time for MS can be very long. When MS suspends an application for garbage collection, the application remains paused for the duration of the collection. Since the duration of garbage collection is unbounded, such pauses are not acceptable in real-time environments.

In LPC and the DDC collectors pause times are short and relatively infrequent. They occur a constant number of times during a collection and do not invade application time.

5.3.4 Comparing collectors with completeness

MS is the only complete collector among the collectors we are comparing. Whenever it runs it collects every object that becomes garbage before it starts. The other collectors are incomplete collectors because they cannot collect cycles. In applications where cycles are not prevalent those collectors reclaim most of the garbage.

When benchmarking a collector many researchers use MS as the base collector with which they compare throughput and completeness. MS is noted as a collector that gives high throughput and low time overhead. There may be a correlation between completeness and these other features, though investigation of that issue is left as future work.

5.3.5 Comparing collectors with overhead

Overhead, especially time overhead, has a direct relation to throughput. The greater the overhead, the longer it takes an application to complete execution; hence, the smaller the throughput. Garbage collectors that incur a lot of overhead take longer to perform collection work. As such, they prolong execution time and reduce throughput.

RCC incurs considerable overhead during pointer assignments. Every time a pointer is updated two reference count updates must be performed. MS incurs little overhead since it rarely runs and reclaims all objects that become dead before it starts running. The multi-thread multiprocessor collectors LPC and DDC incur some overhead but pointer updates are summarized. If a pointer references several objects during the course of a collection, only reference count updates for the initial object the pointer pointed from and the last object it points to are required. DDC by design incurs less time overhead than LPC because it has fewer handshakes.

5.3.6 Comparing collectors with concurrency

High concurrency is a desirable feature of a garbage collector. As a matter of fact, the current trend in garbage collection technology is toward concurrency. High concurrency improves mutator utilization of the CPU, decreases pause times, and increases throughput. These are all desirable features of a modern garbage collector.

Among the collectors listed in Figure 5.1 RCC and MS are not concurrent. These collectors were targeted for uniprocessor environments. LPC and DDC were designed for multi-thread, multiprocessor targets. As such their level of concurrency is high. Both collectors do well with concurrency; however DDC does better by design for two reasons: DDC has fewer handshakes and DDC uses a non-mutator-blocking write barrier and handshaking mechanism.

5.3.7 Comparing collectors with throughput

RCC does not have high throughput because of the time it spends updating reference counts during pointer assignments. Frequent pointer updates increase memory traffic, consume time, and reduce overhead. MS incurs less overhead than the other collectors since it runs infrequently. Moreover, it runs only when it needs to so memory can be reclaimed for the next allocation.

Both LPC and DDC have high throughput. However, DDC does better than LPC by design because DDC has fewer handshakes, which are comparable in speed to the write barrier.

Given an application with certain characteristics, application developers should be able to use `GC-Tax` to help them determine which of these collectors would be most suitable to manage memory for their applications.

Chapter 6

Conclusions and future work

This chapter summarizes the research contributions of this dissertation to the field of memory management. This chapter also suggests ideas for future research in memory management.

6.1 Research summary

This dissertation presents asymptotic time-complexity analysis for RTSJ scoped-memory areas and NHRTs. One approach to complexity analysis suggests implementations in RTSJ for abstract data types like stack and queue and determines asymptotic bounds for their execution. These results allow us to compare scoped memory with other memory models and to reason more thoroughly about the differences among those models.

One assumption for this approach considers one element per scope. While we do not recommend this restriction in practice for efficiency reasons, the analysis holds even when we allow multiple elements per scope. Consider, for example, a maximum of $4k$ elements per scope. Suppose $4k$ elements are already stored on a queue and another element needs to be enqueued. Since the current scope has no more available storage to accommodate the new element a new scope needs to be instantiated. That enqueue operation suffers cost linear in the number of elements already on the queue.

Another approach to providing asymptotic time-complexity analysis for RTSJ scopes and NHRTs is to migrate extant functional programming language implementations of data structures to RTSJ. This code migration also allows us to migrate time-complexity analysis for data structures implemented in a functional programming language to RTSJ. Using this approach allows us to discover that for certain data structures, runtime complexity analysis for RTSJ is comparable to runtime complexity analysis for the heap. Moreover, using RTSJ forces the developer to think more carefully about memory management since RTSJ scopes can leak an unbounded amount of memory. Our work is the first work to point this out.

Another contribution of this research is the design of a high performance, on-the-fly referencing garbage collector for Java. Like its predecessor [58], this garbage collector targets a multi-threaded, multiprocessor environment. It offers low synchronization with mutators, minimal handshaking, low runtime overhead, high throughput, and high concurrency. To evaluate our collector we implemented it in the GNU C compiler and used it to manage memory for selected Java benchmarks. The results support our claims.

The third contribution of this dissertation is the development of a taxonomy of garbage collectors, called `GC-Tax`. Prior to this work, garbage collection research has focused mainly on designing, implementing, and evaluating garbage collectors. Bibliographies, comparisons, surveys, and reviews of garbage collectors were also done. However, little or no work was done on producing a taxonomy that unifies the theory of garbage collection. This dissertation presents a taxonomy that helps to unify the theory of garbage collection and provides software developers a tool to decide which garbage collectors are most suitable for their applications.

6.2 Future work

As part of our future work we would like to implement and supply the RTSJ community with data structure packages that reflect the runtime complexities presented in Chapter 3. For RTSJ data structure implementations with runtime complexities that are worse than heap implementations, we would like to explore ways to reduce their complexities. To accomplish this goal, we need to address the memory management issues associated with data structure implementations. In particular, we need to determine when scopes should be exited so their storage can be reclaimed. To do this determination, further analysis must be performed to determine when data are no longer needed. Ideas from the garbage collection community may be helpful.

In the not-to-distant future we would like to transform our high-performance, on-the-fly reference counting garbage collector into a real-time garbage collector. To accomplish this goal we need to bound the number of objects that get collected at the end of each collection. We would also need to bound or *incrementalize* mutator stack scanning, global data segment scanning, and reference count updates. Although most of these operations (except stack scanning) are performed concurrently with mutator execution, bounding them or making them incremental would improve minimum mutator utilization of the processors. Moreover, the collector might be just as efficient stealing cycles from mutators as it is executing on a dedicated processor. Of course, there are potentially other unforeseen issues that could make the transformation difficult.

Finally, we would like to extend our taxonomy of garbage collectors to include other memory management efforts. In particular, we would like to be able to use the taxonomy to compare memory management efforts like manual memory management, RTSJ scoped memory, and hybrid approaches. Further, since each feature in the feature set of `GC-Tax` is associated with some metric that can be measured, we would like to implement `GC-Tax` so that developers can use it to perform empirical analysis for garbage collectors that interest them. The **Java Virtual Machine**^{TM} **Tools Interface** (JVMTI) [85] might be a reasonable platform to target for the implementation of `GC-Tax`.

Appendix A

Data from Experimentation

This appendix provides data obtained from experiments described in Section 4.9.3 of the dissertation. Appropriate captions are used to describe the data contained in each figure presented below.

Clock Tick Count			
	WB: no logging	WB: logging	Handshake
Avg Time - SortNumbers	833.0985301	948.2355453	786.9382466
Avg Time - SimpleThreads	648.7342234	973.0214094	764.2954444
Wall Clock Time (μs)			
Avg Time - SortNumbers	0.348170204	0.396288495	0.328878806
Avg Time - SimpleThreads	0.271120304	0.406647053	0.319415881

Figure A.1: Average time cost for an operation, observed for the 100,000 runs of the application when the collector is enabled and the operation is executed.

Clock Tick Count			
	WB: no logging	WB: logging	Handshake
STDEV Time - SortNumbers	904.0913347	174.453802	155.5590813
STDEV Time - SimpleThreads	645.3218328	515.85487	163.0090096
Wall Clock Time (μs)			
STDEV Time - SortNumbers	0.377839659	0.072908082	0.065011562
STDEV Time - SimpleThreads	0.269694191	0.215587099	0.068125051

Figure A.2: Standard deviation of the time cost for an operation, observed for the 100,000 runs of the application when the collector is enabled and the operation is executed.

	WB: no logging	WB: logging	Handshake
Total Count - SortNumbers	510184324	1149215676	179974
Total Count - SimpleThreads	310411583	1154935030	180000

Figure A.3: Count of the number of times each operation is executed, observed for the 100,000 runs of the application when the collector is enabled and the operation is executed.

Clock Tick Count			
	WB: no logging	WB: logging	Handshake
Total Time - SortNumbers	425034000000	1089730000000	141628424
Total Time - SimpleThreads	201375000000	1123780000000	137573180
Wall Clock Time (s)			
Total Time - SortNumbers	177.63	455.42	0.06
Total Time - SimpleThreads	84.16	469.65	0.06

Figure A.4: Total time cost for total number of executions of an operation, observed for the 100,000 runs of the application when the collector is enabled and the operation is executed.

Clock Tick Count			
	WB: no logging	WB: logging	Handshake
MIN Time - SortNumbers	516	740	296
MIN Time - SimpleThreads	520	744	296
Wall Clock Time (μs)			
MIN Time - SortNumbers	0.215647752	0.30926228	0.123704912
MIN Time - SimpleThreads	0.21731944	0.310933968	0.123704912

Figure A.5: Minimum time cost for an operation, observed for the 100,000 runs of the application when the collector is enabled and the operation is executed.

References

[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. 1988.

[2] Zakarya Alzamil. Application of Computational Redundancy in Dangling Pointers Detection. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA'06)*, page 30. IEEE Computer Society, 2006.

[3] Andrew W. Appel. Garbage Collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 89–100. The MIT Press, Cambridge, MA, 1991.

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-Time Concurrent Collection on Stock Multiprocessors. *SIGPLAN*, 23(7):11–20, 1988.

[5] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, MA, 2000.

[6] John Backus. Can programming be liberated from the Von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[7] David F. Bacon, C. Richard Attanasio, Han Lee, V. T. Rajan, and Stephen Smith. Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–103, 2001.

[8] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling Fragmentation and Space Consumption in the Metronome, a Real-time Garbage Collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 81–92, San Diego, California, June 2003.

[9] David F. Bacon, Perry Cheng, and V. T. Rajan. The Metronome: A Simpler Approach to Garbage Collection in Real-time Systems. In R. Meersman and Z. Tari, editors, *Proceedings of the OTM Workshops: Workshop on Java Technologies for Real-time and Embedded Systems*, volume 2889 of *Lecture Notes in Computer Science*, pages 466–478, Catania, Sicily, November 2003. Springer-Verlag.

[10] David F. Bacon, Perry Cheng, and V. T. Rajan. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *Proceedings of the 30th Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 285–298, New Orleans, Louisiana, January 2003.

[11] David F. Bacon and V. T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. *Lecture Notes in Computer Science*, 2072:207–235, 2001.

[12] Henry G. Baker. List Processing in Real-time on a Serial Computer. *Communications of the ACM*, 21(4):280–94, 1978.

[13] Henry G. Baker. The Treadmill: real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–70, 1992.

[14] Hans-Juergen Boehm. Space Efficient Conservative Garbage Collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 197–206, New York, NY, USA, 1993. ACM Press.

[15] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[16] Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.

[17] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 256–262, New York, NY, USA, 1984. ACM Press.

[18] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison Wesley, April 1991.

[19] D. C. Cann, J. T. Feo, A. D. W. Bohoem, and Rod R. Oldehoeft. *SISAL Reference Manual: Language Version 2.0*, 1992.

[20] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Notices*, 27(8):15–42, 1992.

[21] Bill Catambay. The Pascal Programming Language. `http://pascal-central.com/ppl/index.html`, 2001.

[22] A. M. Cheadle, A. J. Field, S. Marlow, S. L. Peyton Jones, and R. L. While. Non-stop Haskell. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 257–267, New York, NY, USA, 2000. ACM Press.

[23] Perry Cheng and Guy E. Blelloch. A Parallel, Real-time Garbage Collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 125–136, New York, NY, USA, 2001. ACM Press.

[24] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *4th International Conference on Logic Programming*, pages 276–293, 1987.

[25] Jacques Cohen. Garbage Collection of Linked Data Structures. *Computing Surveys*, 13(3):341–367, September 1981.

[26] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[28] Delvin C Defoe, Rob LeGrand, and Ron K Cytron. Asymptotic Analysis for Real-time Java Scoped-memory Areas. In *CCCT 2006: The 4th International Conference on Computing, Communications and Control Technologies*, volume II, pages 131 – 138, Orlando, FL, July 2006. International Institute of Informatics and Systemics.

[29] Morgan Deters and Ron K. Cytron. Automated Discovery of Scoped Memory Regions for Real-time Java. In *ISMM '02: Proceedings of the 3rd International Symposium on Memory Management*, pages 25–35, New York, NY, USA, 2002. ACM Press.

[30] John DeTreville. Experience with Concurrent Garbage Collectors for Modula-2+. Technical Report 64, Digital, Systems Research Center, August 1990.

[31] L. Peter Deutsch and Daniel G. Bobrow. An Efficient Incremental Automatic Garbage Collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[32] Damien Doligez and Xavier Leroy. A Concurrent Generational Garbage Collector for a Multi-threaded Implementation of ML. In *Principles of Programming Languages (POPL'93)*, pages 113–123, January 1993.

[33] Michael Eisenberg. *Programming in Scheme.* MIT Press, June 1988.

[34] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Professional, January 1990.

[35] Richard A. Eyre-Todd. The Detection of Dangling References in C++ Programs. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):127–134, 1993.

[36] Robert R. Fenichel and Jerome C. Yochelson. A Lisp Garbage Collector for Virtual Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[37] David Flanagan. *Java Examples in a Nutshell, 2nd Edition.* O'Reilly Media, Inc., 2000.

[38] International Organization for Standardization. *ISO 7185:1990: Information technology — Programming languages — Pascal.* pub-ISO.

[39] Free Software Foundation. GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF). `http://gcc.gnu.org/`, 2007.

[40] Free Software Foundation. GCJ: The GNU Compiler for Java - GNU Project - Free Software Foundation (FSF). http://gcc.gnu.org/java/, 2007.

[41] Max Goff. Celebrating 10 years of Java and our technological productivity: A look back on the last 10 years of the network age. http://www.javaworld.com, May 2005.

[42] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[43] Matthew P. Hampton. Using contaminated garbage collection and reference counting garbage collection to provide automatic storage reclamation for real-time systems. Master's thesis, Washington University in St. Louis, 2003. Available as Washington University Technical Report WUCSE-2003-31.

[44] David R. Hanson. Storage management for an implementation of SNOBOL4. *Software: Practice and Experience*, 7(2):179–192, 1977.

[45] Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Transaction on Programming Languages and Systems*, 15(2):211–252, 1993.

[46] Pieter H. Hartel. *Performance Analysis of Storage Management in Combinator Graph Reduction.* PhD thesis, Department of Computer Systems, University of Amsterdam, Amsterdam, 1988.

[47] Barry Hayes. Using key object opportunism to collect old objects. In *OOPSLA '91: Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, pages 33–46, New York, NY, USA, 1991. ACM Press.

[48] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181, New York, NY, USA, 2003. ACM Press.

[49] Denis Howe. Free On-Line Dictionary Of Computing. http://foldoc.doc.ic.ac.uk/foldoc/index.html, 1993.

[50] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):1–52, 1992.

[51] Kathleen Jensen and Niklaus Wirth. *Pascal - User Manual and Report (Lecture Notes in Computer Science).* Springer, October 1974.

[52] Richard Jones. The Garbage Collection Bibliography. http://liinwww.ira.uka.de/bibliography/Compiler/gc.html, 1996.

[53] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Ltd, 1996.

[54] Alan C. Kay. The Early History of Smalltalk. In *HOPL-II: The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 69–95, New York, NY, USA, 1993. ACM Press.

[55] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, February 1978.

[56] Donald E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms. Addison Wesley, Reading, Massachusetts, 1963.

[57] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 367–380. ACM Press, 2001.

[58] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Transaction on Programming Languages and Systems*, 28(1):1–69, 2006.

[59] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[60] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 193–202, New York, NY, USA, 2005. ACM Press.

[61] John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Communications of the ACM*, 3:184–195, April 1960.

[62] John McCarthy. History of LISP. In Richard L. Wexelblat, editor, *History of Programming Languages*, chapter IV, pages 173–197. ACM Monograph, 1981.

[63] B Meyer. Eiffel: programming for reusability and extendibility. *SIGPLAN Notices*, 22(2):85–94, 1987.

[64] Marvin L. Minsky. A LISP garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.

[65] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 217–226, New York, NY, USA, 1993. ACM Press.

[66] Kelvin Nilsen. Issues in the design and implementation of real-time Java. *Java Developer's Journal*, 1(1):44, 1996.

[67] Kelvin Nilsen. A type system to assure scope safety within safety-critical Java modules. In *JTRES '06: Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 97–106, New York, NY, USA, 2006. ACM Press.

[68] Kelvin D. Nilsen. Doing firm-real-time with J2SE APIs. In *OTM Workshops*, pages 371–384, 2003.

[69] Chris Okasaki. Functional Data Structures. In *Advanced Functional Programming, LNCS 1129*, pages 131–158. Springer, August 1996.

[70] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, September 1996. This research was sponsored by the Advanced Research Projects Agency (ARPA) under Contract No. F19628-95-C-0050.

[71] James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. Technical Report MIT–LCS–TR–570 and CMU–CS–93–138, MIT and CMU., 1993. Also LFP94 and OOPSLA93 Workshop on Memory Management and Garbage Collection.

[72] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time Java Scoped Memory: Design Patterns and Semantics. In *IEEE International Symposium on Object-oriented Real-time Distributed Computing*, pages 101–110, Vienna, Austria, May 2004. IEEE Computer Society.

[73] Atanas Radenski. A voyage to Oberon. *SIGCSE Bulletin*, 25(3):13–18, 1993.

[74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[75] Robert Harper David MacQueen Robin Milner, Mads Tofte. *The Definition of Standard ML (Revised)*. MIT Press, May 1997.

[76] David Robson. Smalltalk. In *ACM 83: Proceedings of the 1983 Annual Conference on Computers : Extending the Human Resource*, page 133, New York, NY, USA, 1983. ACM Press.

[77] Walter Savitch. *Problem Solving with C++*. Addison Wesley, 2nd edition, 1999.

[78] James A Saxon and William S Plette. *Programming the IBM 1401, a self-instructional programmed manual*. Prentice-Hall, 1962.

[79] Michigan Historical Reprint Series. *A short account of the history of Mathematics, by W. W. Rouse Ball*. Scholarly Publishing Office, University of Michigan Library, December 2005.

[80] Andrew Shalit. *The Dylan reference manual : the definitive guide to the new object-oriented dynamic language*. Apple Computer, Inc., April 1998.

[81] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[82] Will R. Stoye, T. J. W. Clarke, and Arthur C. Norman. Some practical methods for rapid combinator reduction. pages 159–166.

[83] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, January 1986.

[84] Bjarne Stroustrup. Bjarne Stroustrup's FAQ. `http://public.research.att.com/~bs/bs_faq.html`, 2006.

[85] Inc. Sun Microsystems. JVM^{TM} Tool Interface: Version 1.0. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`, 2004.

[86] Inc Sun Microsystems. The SimpleThreads Example The Java^{TM} Tutorials > Essential Classes > Concurrency. `http://java.sun.com/docs/books/tutorial/essential/concurrency/simple.html`, 2007.

[87] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[88] D Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, 1986.

[89] David Ungar. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *SDE 1: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, New York, NY, USA, 1984. ACM Press.

[90] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Inc., 1991.

[91] David H D Warren, Luis M. Pereira, and Fernando Pereira. Prolog - the language and its implementation compared with Lisp. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 109–115, New York, NY, USA, 1977. ACM Press.

[92] Mark A. Weiss. *Data Structures and Algorithm Analysis in C*. Addison-Wesley Longman, Inc., Menlo Park, CA, second edition, 1997.

[93] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.

[94] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.

[95] David S. Wise. Stop and one-bit reference counting. Technical Report 360, Indiana University, Computer Science Department, March 1993.

[96] David S. Wise and Daniel P. Friedman. The one-bit reference count. *BIT*, 17(3):351–9, 1977.

[97] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.

Curriculum Vitae

Delvin Curvin Defoe

Date of Birth	November 28, 1973
Place of Birth	La Plaine, Dominica
Degrees	B.S. Magna Cum Laude, Mathematics and Computer Science, May 2001, Midwestern State University, Wichita Falls, Texas.
	M.S. Computer Science, December 2003, Washington University, Saint Louis, Missouri.
Professional Societies	Association for Computing Machinery
Publications	Delvin C Defoe, Rob LeGrand, and Ron K Cytron. Asymptotic Analysis for Real-time Java Scoped-memory Areas. In *CCCT 2006: Proceedings of the 4th International Conference on Computing, Communications and Control Technologies*, pages 131-138, Orlando, FL, July 2006.
	Delvin C. Defoe, Sharath R. Cholleti, and Ron K. Cytron. Upper bound for defragmenting buddy heaps. In *LCTES05: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 222–229, New York, NY, USA, 2005. ACM Press.
	Delvin C. Defoe. Effects of Coalescing on the Performance of Segregated Size Storage Allocators. Masters thesis, Washington University in St. Louis, 2003. Available as Washington University Technical Report WUCSE-2003-69.
	Delvin Defoe, Ranette Halverson, Nelson Passos, Richard Simpson, and Reynold Bailey. “A Study of Software Pipelining for Multi-dimensional Problems”, in *Proceedings of the AeroSense-Aerospace/Defense Sensing, Simulation and Controls*, Orlando, FL, April 2001.

Delvin Defoe, Ranette Halverson, Nelson Passos, Richard Simpson, and Reynold Bailey. "Theoretical Constraints on Multi-Dimensional Retiming Design Techniques", in *Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, August 2000.

August 2007

Short Title: Exploration of Dynamic Memory Defoe, Ph.D. 2007