Region-Based Memory Management for Real-Time Java

by

William S. Beebee, Jr.

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

Masters of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author . . . .
Department of Electrical Engineering and Computer Science
August 24, 2001

Certified by . . . .
Martin Rinard
Associate Professor of Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Region-Based Memory Management for Real-Time Java

by

William S. Beebee, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on August 24, 2001, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science

Abstract

The Real-Time Specification for Java [3] provides a framework for building real-time systems. We implemented the memory management portion of this specification and wrote sample applications to test it. We discuss the structure of our implementation and problems encountered during implementation and application development. The primary implementation problem arises from a design in which some threads are vulnerable to garbage collection pauses because they can allocate memory in the garbage-collected heap, while other threads must execute independently of the garbage collector to provide acceptable response times even in the face of garbage collection pauses. The Real-Time Specification for Java allows the different kinds of threads share the same resource. Using blocking locks to manage access to these shared resources can lead to unacceptable interactions in which one thread indirectly and unacceptably forces another thread to observe garbage collection pauses. We solved this potential problem by using non-blocking, optimistic synchronization primitives throughout the implementation. A second implementation problem arises from the need to manage the memory associated with the memory management data structures. The lack of a recycling memory area which can be accessed from any execution point led to the development of a new memory area not mentioned in the specification. Because we found it difficult, if not impossible, to correctly use the Real-Time Java extensions without automated assistance, we developed an extensive debugging facility.

Thesis Supervisor: Martin Rinard
Title: Associate Professor of Computer Science

Acknowledgments

This research was supported in part by DARPA/AFRL Contract F33615-00-C-1692 as part of the Program Composition for Embedded Systems program. I would like to acknowledge Scott Ananian for a large part of the development of the MIT Flex compiler infrastructure and the Precise-C backend. Karen Zee implemented stop and copy and mark and sweep garbage collectors, which facilitated the development of no-heap real-time threads. Hans Boehm, Alan Demers, and Mark Weiser implemented the conservative garbage collector which was used for all of the listed benchmarks. Alex Salcianu tailored his escape analysis to verify the correctness of our Real-Time Java benchmarks with respect to scoped memory assignment violations. Brian Demsky implemented a user-threads package for the Flex compiler which improved the performance of some benchmarks. Victor Kuncak, the research paper fairy, bestowed interesting, relevant, and useful related work on my desk in the middle of the night. Peter Beebee, my brother, gave me the idea for a multiple-stack recycling heap based on reference counts, which eventually became the reference count memory allocator. Finally, I would like to acknowledge the support of Martin Rinard, my advisor, during the entire year it took to produce this work, from initially finding a problem to tackle, through the multiple failed implementations and final success, to the insightful and plentiful comments on drafts of this thesis. Thanks for always being there.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Java is a relatively new and popular programming language. It provides a safe, garbage-collected memory model (no dangling references, buffer overruns, or memory leaks) and enjoys broad support in industry. The goal of the Real-Time Specification for Java [3] is to extend Java to support key features required for writing real-time programs. These features include support for real-time scheduling and predictable memory management.

This paper presents our experience implementing the Real-Time Java memory management extensions. The goal of these extensions is to preserve the safety of the base Java memory model while giving the real-time programmer the additional control that he or she needs to develop programs with predictable memory system behavior. In the base Java memory model, all objects are allocated out of a single garbage-collected heap, raising the issues of garbage-collection pauses and unbounded object allocation times.

Real-Time Java extends this memory model to support two new kinds of memory: *immortal memory* and *scoped memory*. Objects allocated in immortal memory live for the entire execution of the program. The garbage collector scans objects allocated in immortal memory to find (and potentially change) references into the garbage collected heap but does not otherwise manipulate these objects.

Each scoped memory conceptually contains a preallocated region of memory that threads can enter and exit. Once a thread enters a scoped memory, it can allocate objects out of that memory, with each allocation taking a predictable amount of time. When the thread exits the scoped memory, the implementation deallocates all objects allocated in the scoped memory (but not necessarily the memory associated with the memory area) without garbage collection. The specification supports nested entry and exit of scoped memories, which threads can use to obtain a stack of active scoped memories. The lifetimes of the objects stored in the inner scoped memories are contained in the lifetimes of the objects stored in the outer scoped memories. As for objects allocated in immortal memory, the garbage collector scans objects allocated in scoped memory to find (and potentially change) references into the garbage collected heap but does not otherwise manipulate these objects.

The Real-Time Specification for Java [3] uses dynamic assignment checks to prevent dangling references and ensure the safety of using scoped memories. If the program attempts to create either 1) a reference from an object allocated in the heap to an object allocated in a scoped memory or 2) a reference from an object allocated in an outer scoped memory to an object allocated in an inner scoped memory, the specification requires the implementation to throw an exception.

1.1 Threads and Garbage Collection

The Real-Time Java thread and memory management models are tightly intertwined. Because the garbage collector may temporarily violate key heap invariants, it must be able to suspend any thread that may interact in any way with objects allocated in the garbage-collected heap. Real-Time Java therefore supports two kinds of threads: real-time threads, which may access and refer to objects stored in the garbage-collected heap, and no-heap real-time threads, which may not access or refer to these objects. No-heap real-time threads execute asynchronously with the garbage collector; in particular, they may execute concurrently with or suspend the garbage collector at any time. On the other hand, the garbage collector may suspend real-time threads at any time and for unpredictable lengths of time.

The Real-Time Specification for Java [3] uses dynamic heap reference checks to prevent interactions between the garbage collector and no-heap real-time threads. If a no-heap real-time thread attempts to manipulate a reference to an object stored in the garbage-collected heap, the specification requires the implementation to throw an exception. We interpret the term “manipulate” to mean read or write a memory location containing a reference to an object stored in the garbage collected heap, or to execute a method with such a reference passed as a parameter.

1.2 Implementation

The primary complication in the implementation is potential interactions between no-heap real-time threads and the garbage collector. One of the basic design goals in the Real-Time Specification for Java [3] is that the presence of garbage collection should never affect the ability of the no-heap real-time thread to run. We devoted a significant amount of time and energy working with our design to convince ourselves that the interactions did in fact operate in conformance with the specification.

The possibility of dead-locks or unacceptable pauses in no-heap real-time threads associated with the use of blocking locks led to the ubiquitous use of non-blocking synchronization primitives as a substitute for blocking locks. Developing non-blocking algorithms to avoid race conditions which result in inconsistent shared state required significant time and energy.

1.3 Debugging

We found it difficult to use scoped and immortal memories correctly, especially in the presence of the standard Java libraries, which were not designed with the Real-Time Specification for Java [3] in mind. We therefore found it useful to develop some debugging tools. These tools included a static analysis which finds incorrect uses of scoped memories and a dynamic instrumentation system that enabled the implementation to print out information about the sources of dynamic check failures.

Chapter 2

Related Work

Christiansen and Velschow suggested a region-based approach to memory management in Java; they called their system RegJava [5]. They found that fixed-size regions have better performance than variable-sized regions and that region allocation has more predictable and often better performance than garbage collection. Static analysis can be used to detect where region annotations should be placed, but the annotations often need to be manually modified for performance reasons. Compiling a subset of Java which did not include threads or exceptions to C++, the RegJava system does not allow regions to coexist with garbage collection. Finally, the RegJava system permits the creation of dangling references.

Gay and Aiken implemented a region-based extension of C called C@ which used reference counting on regions to safely allocate and deallocate regions with a minimum of overhead [1]. Using special region pointers and explicit `deleteregion` calls, Gay and Aiken provide a means of explicitly manipulating region-allocated memory. They found that region-based allocation often uses less memory and is faster than traditional malloc/free-based memory management. Unfortunately, counting escaping references in C@ can incur up to 16% overhead. Both Christiansen and Velschow and Gay and Aiken explore the implications of region allocation for enhancing locality.

Gay and Aiken also produced RC [2], an explicit region allocation dialect of C, and an improvement over C@. RC uses heirarchically structured regions and `sameregion`, `traditional`, and `parentptr` pointer annotations to reduce the reference counting overhead to at most 11% of execution time. Using static analysis to reduce the number of safety checks, RC demonstrates up to a 58% speedup in programs that use regions as opposed to garbage collection or the typical malloc and free. RC uses 8KB aligned pages to allocate memory and the runtime keeps a map of pages to regions to resolve `regionof` calls quickly. Regions have a partial order to facilitate `parentptr` checks.

Region analysis seems to work best when the programmer is aware of the analysis, indicating that explicitly defined regions which give the programmer control over storage allocation may lead to more efficient programs. For example, the Tofte/Talpin ML inference system required that the programmer be aware of the analysis to guard against excessive memory leaks [10, 1]. Programs which use regions explicitly may be more hierarchically structured with respect to memory usage by programmer design than programs intended for the traditional, garbage-collected heap. Therefore, Real-Time Java uses hierarchically-structured, explicit, reference-counted regions that strictly prohibit the creation of dangling references.

The Tofte/Talpin approach uses a single stack of regions to allocate memory [10, 5]. Allocating memory in a single stack of regions facilitates static inference of the containing region for each object based on lexical analysis of a single threaded program. Unfortunately, simply extending this approach to multi-threaded programs may cause memory leaks. If two threads share the same region, the region must be located at the top of the stack and live for the duration of the program [11].

Contaminated garbage collection uses stacks of regions and dynamic region inference instead of static region inference [4]. An assignment of a field of an object contained in a region with a longer lifetime to point to an object contained in a region with a shorter lifetime can cause the object allocated in the region with a shorter lifetime to be moved to the region with a longer lifetime. Thus, the assignment "contaminates" the object by lifting it to a region higher on the stack. The garbage collector places objects accessable from multiple threads in a region with a lifetime of the entire program [4].

Our system has a need for a shared, recyclable memory accessible to multiple threads. We implement this memory by counting the number of times a thread enters a memory area minus the number of times a thread exits a memory area. If this count is zero, our implementation deallocates the objects contained in the memory area. We allow threads to allocate memory from any memory area on a thread's memory area stack provided that the memory area is accessable from the current execution point. Multiple threads can share the memory areas entered by the parent thread.

Our research is distinguished by the fact that Real-Time Java is a strict superset of the Java language; any program written in ordinary Java can run in our Real-Time Java system. Furthermore, a Real-Time Java thread which uses region allocation and/or heap allocation can run concurrently with a thread from any ordinary Java program, and we support several kinds of region-based allocation and allocation in a garbage collected heap in the same system.

Chapter 3

Programming model

Because of the proliferation of different kinds of memory areas and threads, Real-Time Java has a fairly complicated programming model for memory areas.

3.1 Entering and Exiting Memory Areas

Real-Time Java provides several kinds of memory areas: scoped memory, immortal memory, and heap memory. Each thread maintains a stack of memory areas; the memory area on the top of the stack is the thread's default memory area. When the thread creates a new object, it is allocated in the default memory area unless the thread explicitly specifies that the object should be allocated in some other memory area. If a thread uses this mechanism to attempt to allocate an object in a scoped memory, the scoped memory must be present in the thread's stack of memory areas. No such restriction exists for objects allocated in immortal or heap memory.

Threads can enter and exit memory areas. When a thread enters a memory area, it pushes the area onto its stack. When it exits the memory area, it pops the area from the stack. There are two ways to enter a memory area: start a parallel thread whose initial stack contains the memory area, or sequentially execute a run method that executes in the memory area. The thread exits the memory area when the run method returns.

The programming model is complicated somewhat by the fact that 1) a single thread can reenter a memory area multiple times, and 2) different threads can enter memory areas in different orders. Assume, for example, that we have two scoped memories A and B and two threads T and S. T can first enter A, then B, then A again, while S can first enter B, then A, then B again. The objects in A and B are deallocated only when T exits A, then B, then A again, and S exits B, then A, then B again. Note that even though the programming model specifies nested entry and exit of memory areas, these nested entries and exits do not directly translate into a hierarchical inclusion relationship between the lifetimes of different memory areas.

3.2 Scoped Memories

Scoped memories, in effect, provide a form of region-based memory allocation. They differ somewhat from other forms of region-based memory allocation [2] in that each scoped memory is associated with one or more computations (each computation is typically a thread, but can also be the execution of a sequentially invoked run method), with all of the objects in the scoped memory deallocated when all of its associated computations terminate.

The primary issue with scoped memories is ensuring that their use does not create dangling references, which are references to objects allocated in scoped memories that have been deallocated. The basic strategy is to use dynamic assignment checks to prevent the program from creating a reference to an object in a scoped memory from an object allocated in either heap memory, immortal memory, or a scoped memory whose lifetime encloses that of the first scoped memory. Whenever a thread attempts to store a reference to a first object into a field in a second object, an assignment check verifies that:

> If the first object is allocated in a scoped memory, then the second object must also be allocated in a scoped memory whose lifetime is contained in the lifetime of the scoped memory containing the first object.

The implementation checks the containment by looking at the thread's stack of scoped memories and checking that either 1) the objects are allocated in the same scoped memory, or 2) the thread first entered the scoped memory of the second object before it first entered the scoped memory of the first object. If this check fails, the implementation throws an exception.

Let's consider a quick example to clarify the situation. Assume we have two scoped memories A and B, two objects O and P, with O allocated in A and P allocated in B, and two threads T and S. Also assume that T first enters A, then B, then A again, while S first enters B, then A, then B again. Now T can store a reference to O in a field of P, but cannot store a reference to P in a field of O. For S, the situation is reversed: S cannot store a reference to O in a field of P, but can store a reference to P in a field of O.

3.3 No-Heap Real-Time Threads

No-heap real-time threads have an additional set of restrictions; these restrictions are intended to ensure that the thread does not interfere with the garbage collector. Specifically, the Real-Time Specification for Java [3] states that a no-heap real-time thread, which can run asynchronously with the garbage collector, "is never allowed to allocate or reference any object allocated in the heap nor is it even allowed to manipulate the references to objects in the heap." Our implementation uses five runtime heap reference checks to ensure that a no-heap real-time thread does not interfere with garbage collection by manipulating heap references. The implementation uses three of these types of checks, **CALL**, **METHOD**, and **NATIVECALL** to guard against poorly implemented native methods or illegal compiler calls into the runtime. These three checks can be removed if all native and runtime code is known to operate correctly.

- **CALL**: A native method invoked by a no-heap real-time thread cannot return a reference to a heap allocated object.

- **METHOD**: A Java method cannot be passed a heap allocated object as an argument while running in a no-heap real-time thread.

- **NATIVECALL**: A compiler-generated call into the runtime implementation from a no-heap real-time thread cannot return a reference to a heap allocated object.

- **READ**: A no-heap real-time thread cannot read a reference to a heap allocated object.

- **WRITE**: As part of the execution of an assignment statement, a no-heap real-time thread cannot overwrite a reference to a heap allocated object.

Chapter 4

Implementation

In this chapter, we identify the seven data structures central to the allocation system, develop general implementation techniques for non-blocking synchronization and managing a globally-accessible, recycling memory area with real-time performance guarantees, describe the events that the system handles, and elaborate on the limitations and nuances of the implementation imposed by the specification.

We first describe the seven major data structures that participate in memory management: the memory area, memory area shadow, allocation data structure, list allocator, block allocator, real-time thread, and memory area stack. We describe the purposes for their thirty-one component parts. Since the memory management system shares and updates the component parts between multiple threads and cannot use blocking locks, it requires non-blocking synchronization.

Then we present general implementation techniques to handle non-blocking synchronization, including: non-blocking synchronization primitives, optimistic synchronization, status variables, and atomic handshakes. We describe the reference count memory allocator, which uses non-blocking synchronization techniques to manage memory for the data structures which manage memory and handle multiple events in the allocation system.

Then we describe each of the events that can occur within the system: constructing a memory area, entering it, allocating memory, collecting garbage, checking for illegal heap references in no-heap real-time threads or illegal assignments, exiting a memory area, and finalizing data structures. The specification constrains the implementation of these events.

Finally, we describe the problems and limitations created by the specification. The stack of memory areas cannot just be an annotated call stack. Objects may be accessable, but the memory areas that contain them may not be accessable. Thread local state implementations of the specification may run into trouble. The fixed-size of performance-bounded memory areas may limit the portability of real-time libraries. With further research and specification development, libraries, programs, and Real-Time Java implementations may surmount these limitations.

4.1 Data Structures

Our Real-Time Java implementation uses seven data structures to manage memory:

- Memory area: This object represents a region of memory from which the program can allocate objects.

- Memory area shadow: This object is a clone of the original memory area object allocated from immovable memory. It provides a means to access data from a heap-allocated memory area object during the execution of a no-heap real-time thread for the purpose of checking the legality of an assignment.

- Allocation data structure: This data structure is a memory area-independent interface for allocating memory in a memory area, finding the roots into the garbage-collected heap, and finalizing objects allocated in a memory area.

- List allocator: This data structure stores the Java objects allocated by the variable-time allocator in a linked list.

- Block allocator: This data structure stores the Java objects allocated by the linear-time allocator in a stack.

- Real-time thread: This object shows the portion of real-time threads which interact with the Real-Time Java memory allocation system.

- Memory area stack: This object stores previously entered memory areas in a linked list.

4.1.1 Memory Area

The memory area object represents a region of memory which the program can use to allocate objects. The memory area object has five parts:

- finalizer: This function deallocates structures associated with this memory area.
- initial size: This integer specifies the amount of memory committed in the constructor of the memory area.
- maximum size: This integer specifies the maximum amount of memory this memory area can allocate.
- allocation data structure: This data structure provides the capability of allocating memory in the memory area.
- shadow: This object is a no-heap real-time thread accessable version of this memory area.

4.1.2 Memory Area Shadow

The shadow is a clone of a memory area, accessable from no-heap real-time threads which can be finalized by the original memory area finalizer. Thus, the shadow has the same lifetime as the original memory area. The shadow pointer of a shadow points to itself to indicate that it is the shadow of the original memory area.

4.1.3 Allocation Data Structure

The allocation data structure provides a memory area non-specific interface to allocate and manage the memory associated with a memory area. The structure has nine parts:

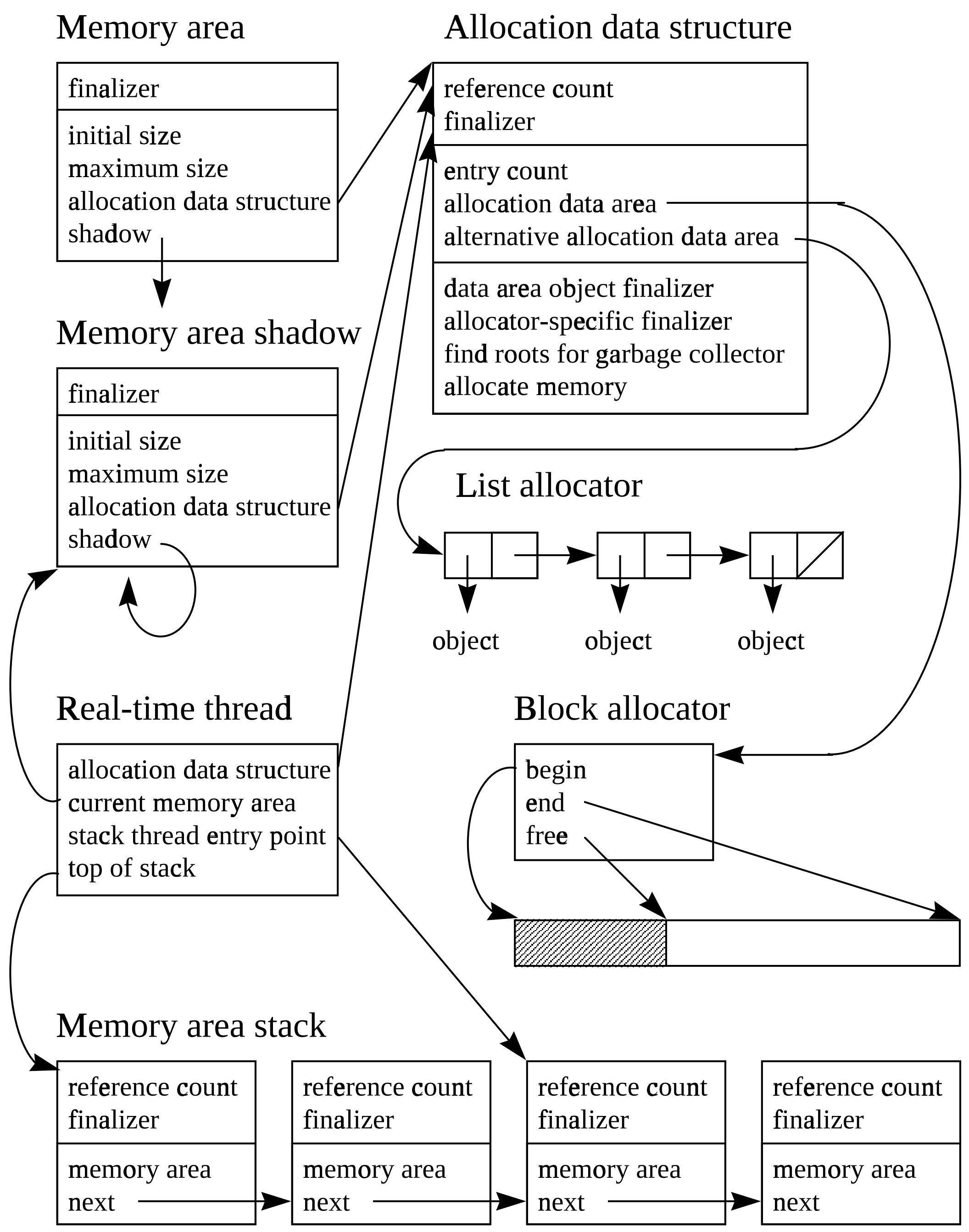

Figure 4-1: Data structures associated with the life cycle of an LTMemory

- reference count: This integer determines when to deallocate the allocation data structure.
- finalizer: This function determines how to finalize allocation data structures in general.
- entry count: This integer stores the difference between the number of times a thread entered this memory area and the number of time a thread exited this memory area. When the entry count is zero, our Real-Time Java implementation finalizes the objects in the memory area.
- allocation data area: This pointer is storage for the function to allocate memory.
- alternative allocation data area: This pointer is alternative storage used by a second allocator for this memory area.
- data area object finalizer: This function finalizes the objects in the memory area.
- allocator-specific finalizer: This function determines how to deallocate the allocation and alternative allocation data areas.
- function to find garbage collector roots: This function collects heap references in the memory area to add to the garbage collector rootset.
- function to allocate memory: This function allocates memory for new objects from the allocation data areas.

4.1.4 List Allocator

The list allocator data structure is a linked list of Java objects allocated by the variable-time allocator. The variable-time allocator can call `malloc` and atomically add a new object to the beginning of the list. Immortal memory areas, variable-time allocation memory areas, and even linear-time allocation memory areas use the list allocator data structure to allocate memory. The linear-time allocation memory areas only use the variable-time allocator for allocation of data if the size requested cannot be allocated by the block allocator.

4.1.5 Block Allocator

The block allocator data structure is a fixed-size stack of memory used by the linear-time allocator to allocate Java objects. The begin, free, and end pointers point to the beginning of the block, the first free byte of storage, and the end of the block respectively. The linear-time allocation memory area uses the block allocator data structure to store objects allocated in the memory initially committed by the constructor of the linear-time allocation memory area.

4.1.6 Real-Time Thread

The real-time thread keeps track of the current memory area, a history of all memory areas entered since the start of the thread (thread stack entry point), a pointer to the memory area stack frozen at the beginning of the thread's execution, and a pointer to the current allocation data structure to facilitate memory allocation.

4.1.7 Memory Area Stack

The memory area stack has a reference count and a finalizer to determine when and how this memory area stack object should be deallocated. The memory area pointer points to an active memory area, and the next pointer points to the next item in the memory area stack. The memory area of the next memory area stack object is the parent memory area of the memory area of this memory area stack object.

4.2 General Implementation Techniques

The requirements of the Real-Time Specification for Java [3] regarding interactions between the garbage collector, real-time threads, and no-heap real-time threads restrict

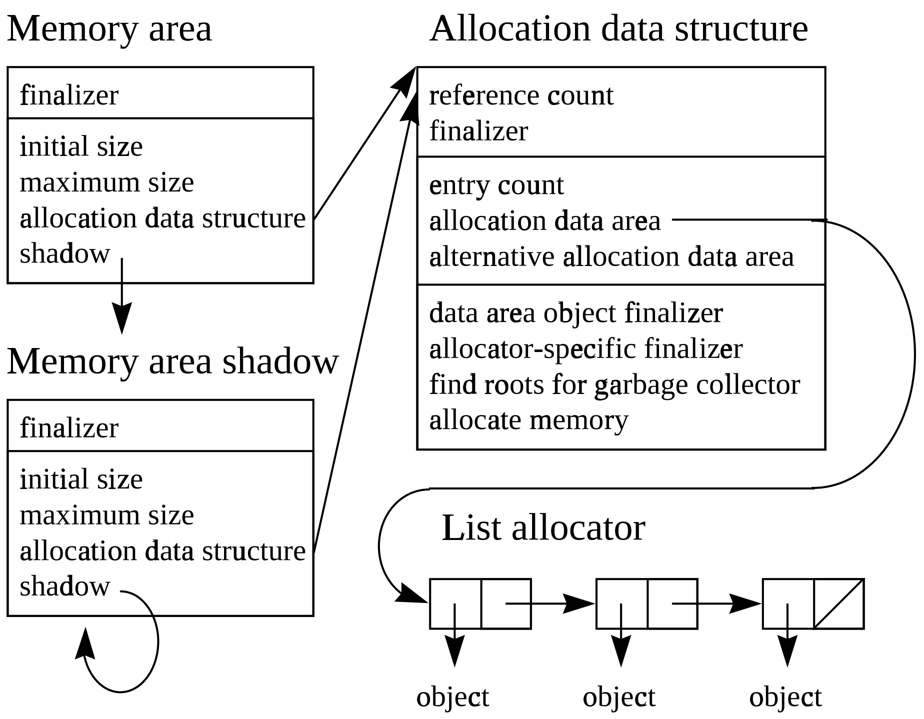

Figure 4-2: Data structures associated with a VTMemory or ImmortalMemory

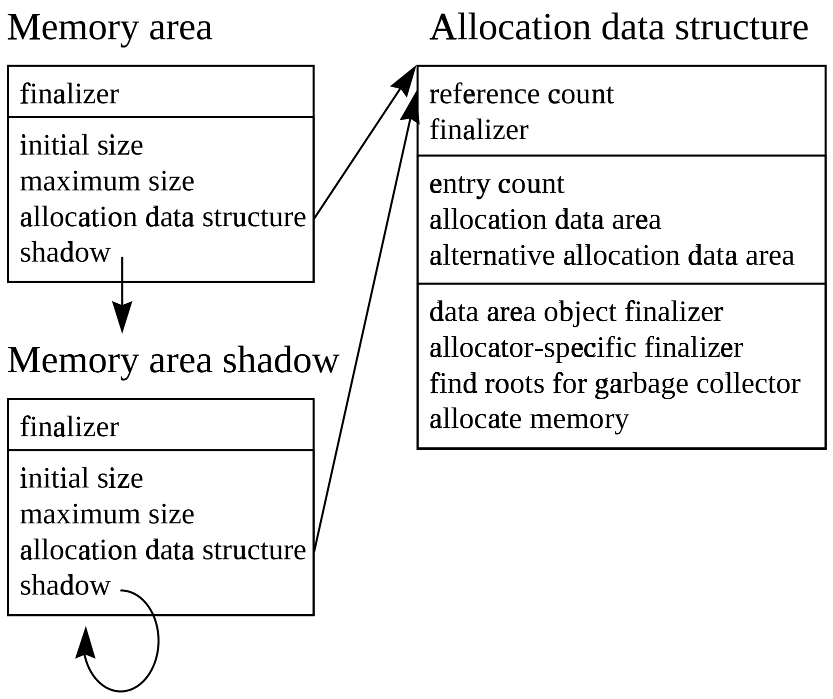
Memory area
finalizer
initial size
maximum size
allocation data structure
shadow
Memory area shadow
finalizer
initial size
maximum size
allocation data structure
shadow
Allocation data structure
reference count
finalizer
entry count
allocation data area
alternative allocation data area
data area object finalizer
allocator-specific finalizer
find roots for garbage collector
allocate memory

Figure 4-3: Data structures associated with a HeapMemory

the space of correct implementations. The development of general implementation techniques used liberally throughout the implementation expedited development.

4.2.1 Atomic operations

The Real-Time Specification for Java [3] permits memory allocation from the same memory area from multiple threads. Therefore, multiple threads possibly running on different processors must share a single mutable resource. The standard approach to manage a single mutable resource shared between threads is to use blocking locks. Only one thread can use the resource at a time, and all other threads which need the resource must wait until the thread finishes using the resource. Unfortunately, the use of blocking locks in the allocation system may cause unintended interactions between no-heap real-time threads, the garbage collector, and real-time threads. We avoid this problem by using non-blocking synchronization. We identify four standard non-blocking approaches to managing shared resources: non-blocking synchronization primitives, optimistic synchronization, status variables, and atomic handshakes. [6] Our implementation uses three of the four approaches, since the success of atomic handshakes may depend on the number of processors or the thread scheduling algorithm. The implementation did not require atomic handshakes, so we chose not to constrain the use of our system unnecessarily.

To see how the use of blocking locks in a Real-Time Java memory allocator can cause problems, consider that to allocate memory, the memory allocator manipulates allocation state shared between multiple threads. The allocator must update the shared state atomically, or multiple concurrent updates could cause inconsistencies in the allocator state. The standard solution is to obtain a lock to block all other threads from allocating memory, update the state, then release the lock. Unfortunately, the standard solution does not work. To see what happens if threads block during allocation, consider the following scenario. A real-time thread starts to allocate memory, acquires the lock, is suspended by the garbage collector, which is then suspended by a no-heap real-time thread that also attempts to allocate memory from the same allocator. This is an example of a general problem known as priority inversion, but the standard solution, priority inheritance, will not work. The scheduler cannot raise the priority of the blocking thread to that of the thread being blocked, since attempts to run the real-time thread fail because the garbage collector is in the middle of a collection and the heap may be in an inconsistent state. Unless the implementation does something clever, it could either deadlock or force the no-heap real-time thread to wait until the garbage collector releases the real-time thread to complete its memory allocation. Therefore, blocking locks held between real-time threads and no-heap real-time threads can be problematic. Faced with this problem, we used non-blocking synchronization primitives instead of blocking locks.

Non-blocking synchronization primitives can help avoid race conditions. For example, the implementation uses an atomic add and exchange instruction to increment or decrement a reference count and return the previous value of the reference count (to determine whether the reference-counted object was in use). The linear time allocator uses the same instruction to add an offset to a stack pointer and return a pointer to the allocated memory. Non-blocking synchronization primitives work in this example, but notice that the problem involves contention for only a single, pointer-sized memory location. In general, non-blocking synchronization primitives can only resolve contention for a single, pointer-sized memory location.

Optimistic synchronization can resolve contention for more than a single pointer-size location. The basic approach is to assume optimistically that a condition is true, perform a computation based on that assumption, and atomically retest the initial condition before committing changes. An atomic compare and exchange can retest the initial condition atomically before a destructive mutation occurs. If the initial condition that was true at the beginning of the computation is no longer true by the end of the computation, the computation can be aborted before the erroneous results affect other computations. For example, when adding an object to the front of a linked list, the compare and exchange instruction first tests that the next pointer has been properly set to the previous front of the linked list before setting the front to point to the new item. If the operation fails, the system retries it. Optimistic synchronization works in this example, but assumes infrequent contention for the shared resource during any operations performed on it.

Status variables can be useful if contention is potentially widespread and long lasting. Complex operations such as setting multiple stack pointers atomically consist of simpler operations, each associated with a different status. The program updates the status variable using non-blocking synchronization primitives to represent the current status of the operation. Using status variables, many thread can make progress on a single computation without blocking. Since the status variable can change after the thread reads it, status variables can indicate that certain steps of the complex operation have already been performed, but cannot indicate that other operations have not been performed. Status variables can be used in conjunction with optimistic synchronization to ascertain the true status of the computation atomically before committing changes and then conditionally updating the status atomically. If every thread must finish the complex operation before using its results, then the desirable effects of atomicity with respect to complex operations can be obtained without blocking. For example, when updating three stack pointers atomically in order, first set the status to one. If the status is one, atomically test for the old value of the first stack pointer and update it with the new value. Atomically test for a status of one and set the status to two. If the status is two, atomically test for the old value of the second stack pointer and update it with a new value. Atomically test for a status of two and set the status to three. If the status is three, atomically test and change the third stack pointer. Atomically test for a status of three and set the status to four. If the status is four, all three stack pointers are up to date. The above approach is very robust against potentially great contention for a shared resource. Unfortunately, status variables alone can only indicate which operations have already been performed, not which operations are pending.

Atomic handshakes between threads can be useful if a potentially large amount of information must be passed from one thread to another thread atomically. Typically one thread announces a condition using non-blocking synchronization primitives. When the other thread notices the condition it stores the required information in a shared space and announces an acknowledgement using non-blocking synchronization primitives. When the first thread notices the acknowledgement, it retrieves the information from the shared space. On-the-fly garbage collectors such as the DLG garbage collector [6], used in multi-processor Java implementations, use handshakes between program threads and the garbage collector thread to atomically transfer the rootset. Unfortunately, atomic handshakes depend on the concurrent execution of the two threads. If the thread scheduler runs the first thread to completion before the second thread runs, no handshakes between the threads can succeed. Thus, atomic handshakes depend on parallel scheduling of the handshakes or a small time quanta for context switches between the handshaking threads. Our Real-Time Java implementation does not use atomic handshakes, since our implementation makes no assumptions about the thread scheduler or number of processors, and we chose not to constrain the use of our system unnecessarily.

4.2.2 Reference Count Memory Area

To enable a thread to allocate at any point during its execution, the Real-Time Java system must be capable of accessing the state necessary for allocation from any method running in any type of thread after entering any type of memory area. When we developed the allocation system, we had to figure out where to store the allocation state. Potential candidates included the memory areas. Note that only one type of memory area can contain objects that the allocation system can access or modify at any point during program execution, the immortal memory area. Unfortunately, the allocation system cannot deallocate objects allocated in an immortal memory area. So, we invented our own type of memory area which can allocate objects that the program can access from anywhere, and deallocate objects when they are no longer needed. We explain the reason for this below.

We first expand on why immortal memory areas cannot fulfill the needs of our implementation. For example, memory areas allocate stack objects when entered. Without some recycling scheme, this memory leak could result in an out of memory error. Even if some recycling scheme were in place, once the allocation system allocates an object from immortal memory, the system can never deallocate it. The system would retain storage for the maximum number of stack objects that were used at one time for the entire length of the program, and could not reuse that memory for other objects. The Real-Time Java thread scheduling algorithm must access and mutate a thread list and the garbage collector must access the state associated with memory areas from potentially any step during program execution. Both could benefit from the ability to deallocate storage.

We therefore introduce a new kind of memory area, called a reference count memory area, which fills our implementation's need for a globally-accessable, recycling memory area where the memory area cannot move objects while in use, but can deallocate them using a pause-free, non-blocking, non-locking memory manager. Allocation does not have any time guarantees, but all assignments must take at most constant time (on a single processor). The program must break cycles either in or through a reference count memory area. (Static analysis can prove some programs to not contain cycles.) Conceptually, the reference count memory area counts references to each object it contains. For example, when the program changes a field of object A, from pointing to object B, to point to object C, the reference count memory area decrements the reference count of B and increments the reference count of C. When the reference count of an object reaches zero, the reference count memory area may finalize the object and deallocate the storage associated with it.

The reference count memory area uses a reference count memory allocator to allocate memory. A reference count allocator can allocate an object, increment the object's reference count, decrement the object's reference count, possibly freeing the object if the count becomes zero, and iterate through the active objects. All four operations can be performed simultaneously by different threads; any set of instructions from any of the operations can be interleaved with any set of instructions from any other operations. A reference count memory allocator cannot use locks, since the priorities of the individual threads cannot be changed. Also, no thread can block; every thread must always be able to make forward progress regardless of the state of any other thread. Finally, incrementing and decrementing reference counts must take a (small) constant amount of time (on a single processor), since assignments may be frequent in Java [].

In general, decomposing problems in time or space that seem to require multiple concurrent updates to occur atomically in a non-blocking fashion may yield non-blocking synchronization primitives and a simplified implementation. The use of status variables decomposes synchronization problems in time. The use of multiple stacks decomposes synchronization problems in space. The reference count allocator decomposes the problem in space using multiple stacks.

Conceptually, a reference count memory allocator can be implemented using five stacks. The first stack contains live objects with a non-zero reference count that are currently in-use by the user program, the garbage collector, or any other procedure which may be iterating through them. The second stack contains all the live objects and some recently-dead objects. The third stack contains objects waiting to be deallocated, but may have iterators, such as the garbage collector or the debugging system, actively iterating through them. The fourth stack contains objects waiting to be deallocated that no iterator is currently iterating through. The fifth stack represents free space. A new object is allocated out of the fifth stack and placed on the first (and second) stack. If the object's reference count drops to zero, the allocator immediately removes the object from the first stack, leaving it only on the second stack. The allocator later moves the object to the third stack and removes it from the second stack. When no iterators are currently iterating through active objects, the third stack can be moved to the fourth stack and the third stack can be cleared. The objects from the fourth stack can then be safely deallocated to the fifth stack at any time.

Our implementation of a reference count allocator, which deallocates an object when its reference count becomes zero, stores the reference count as part of an object's header. The object's header also contains a pointer to the next object to form a linked-list of objects that can be scanned during garbage collection. Thus, the reference count and the linked list implement the first and second stacks. An object with a non-zero reference count is conceptually on the first and second stacks. When the object's reference count drops to zero, it is conceptually only on the second stack, in the sense that a procedure iterating over the elements will skip any object with a zero reference count. A separate linked list of free objects implements the third stack. When an iterator reference count becomes zero, the free objects can be moved to a linked list of collectable objects which implements the fourth stack. The fifth stack simply represents the heap, as managed by `malloc` and `free`.

4.3 Overview of Events

The events in the the life cycle of a memory area demonstrate the rest of the implementation of the Real-Time Java allocation system:

- Constructing a memory area object
- Entering the memory area
- The current thread allocates memory for an object.
- The garbage collector scans the memory area.
- The current thread performs a heap reference check.
- The current thread performs an assignment check.
- The current thread exits the memory area.
- The memory area object becomes inaccessable.

4.4 Constructing a memory area object

Any thread can create an instance of a memory area in any of its active memory areas, initializing it with an optional initial size and maximum size. If the thread specifies no size for the memory area, then the amount of memory that the memory area can allocate is limited only by available memory. A linear time scoped memory area must have a specified initial size to ensure linear time allocation, but a variable time scoped memory area does not. The Real-Time Specification for Java [3] specifies that variable time scoped memory areas must have an initial size and a maximum size. Since our implementation permits either fixed or variable-sized memory areas, it is more general than the Real-Time Specification for Java. [3]

The constructor for memory areas calls a native method to set up the allocation data structure and associates it with the memory area by performing the following steps:

- The method allocates the memory associated with the memory area and an allocation data structure which points to it.

- The method inflates the memory area object by allocating memory associated with the memory area object for an additional field known only to the runtime. It stores a pointer to the allocation data structure in the inflated object.

- The method initializes the entry count to either zero or one depending on whether the memory area objects should be finalized by the Real-Time Java memory allocation system or not. For instance, immortal memory areas start with an entry count of one, since objects allocated in the area live for the duration of the program. Heap memory areas also start with an entry count of one, since the objects are finalized by the garbage collector, not the Real-Time Java system. All scoped memory areas have an entry count initialized to zero, since finalization of the contained objects occurs when all threads have exited the memory area.

- The method stores pointers in the allocation data structure to functions which can allocate objects, find the roots for the garbage collector, finalize all objects in the memory area, and deallocate storage for any memory area specific data areas and initial storage allocated in the native method.

- If no-heap real-time thread support is enabled and the memory area object was allocated in heap memory, a native method allocates a shadow (clone) of the memory area outside garbage-collected space. The shadow prevents a no-heap real-time thread from illegally accessing the heap when examining a heap-allocated memory area that allocated a non-heap allocated object to determine the legality of an assignment between two memory areas. The shadow does not contain any pointers to any heap allocated objects, so it can just be allocated with `malloc` and deallocated with `free` when the memory area finalizer runs.

The system has initialized the memory area state, so the program can now enter the memory area.

4.5 Entering the memory area

A thread can enter a given memory area in three ways: 1) through the memory area's `enter` method, which does not start a new thread, 2) through starting a thread which inherits the current default memory area for the thread that started the thread, and 3) through starting a thread after passing a memory area to the thread's constructor.

Entering a memory area attaches the memory area to the current thread for the duration of the invoked `run` method (which may either be an overriden `run` method on the real-time thread or the `run` method of a `Runnable` which can be passed into an `enter` method or passed into the constructor of a real-time thread. When the `run` method completes, the memory area exits and the system reattaches the memory area previously attached to the current thread. The distinctions between different ways of entering a memory area or running a `run` method do not affect the core implementation of the Real-Time Java system and we shall largely ignore them except for cases where a specific example restricts the design space of the Real-Time Java system.

On entering a memory area, the memory area is pushed on the memory area stack and the memory area, memory area allocation routine, and memory area allocation data structures are attached to the current thread as follows:

- To push a memory area on the stack, the Real-Time Java system allocates a new immutable memory area stack object out of a reference count memory area and stores pointers to the memory area pushed and the rest of the stack.

- If either the memory area or the next memory area stack object were allocated out of a reference count memory area, the system increments the memory area or stack object's reference count.

- The system allocates the memory area allocation data structures out of a special reference count memory area which is not directly scanned by the garbage collector, to allow each memory area to provide a specialized function to scan the data area for garbage collector roots, and to prevent infinite recursion in the garbage collector.

- The memory area allocation data structures are attached to the current thread by inflating the thread object and storing a pointer from the inflated thread to a top-level structure.

- Finally, the thread increments the memory area's entry count.

Since the allocation data structure is attached to the current thread, the program can now allocate memory in the entered memory area.

4.6 Memory Allocation

Memory can be allocated in different ways, and the current allocator can be different for different threads and change over time, yet the overhead of finding the correct memory allocation function must be kept to a small constant. Therefore, memory allocation for all Java objects uses a function pointed to by the allocation data structure associated with the current thread. When the program creates a new object, it finds the current thread and calls the allocation function on the allocation data structure associated with the thread. The type of the currently active memory area for the thread determines the allocation function. Linear-time allocation memory areas, variable-time allocation memory areas and immortal memory areas use two memory allocators: a stack allocator and a `malloc`-based allocator.

4.6.1 Memory Allocation Algorithms

We have implemented two simple allocators for scoped memory areas: a stack allocator and a `malloc`-based allocator. The current implementation uses the stack allocator for instances of `LTMemory`, which guarantee linear-time allocation, and the `malloc`-based allocator for instances of `VTMemory`, which provide no time guarantees.

The stack allocator starts with a fixed amount of available free memory. It maintains a pointer to the next free address. To allocate a block of memory, it increments the pointer by the size of the block, then returns the old value of the pointer as a reference to the newly allocated block. Our current implementation uses this allocation strategy for instances of the `LTMemory` class, which guarantees a linear time allocation strategy.

Our current implementation uses a lock-free, non-blocking atomic exchange-and-add instruction to perform the pointer updates. Note that on an multiprocessor in the presence of contention from multiple threads attempting to concurrently allocate from the same memory allocator, this approach could cause the allocation time to depend on the precise timing behavior of the atomic instructions. We would expect some machines to provide no guarantee at all about the termination time of these instructions.

The `malloc`-based allocator simply calls the standard `malloc` routine to allocate memory. Our implementation uses this strategy for instances of `LTMemory`. To provide the garbage collector with a list of heap references, our implementation keeps a linked list of the allocated memory blocks and can scan these blocks on demand to locate references into the heap. The linked list uses an atomic compare and swap instruction to add a new memory block to the front of the linked list.

Our design makes adding a new allocator easy; the `malloc`-based allocator required only 25 lines of C code and only 45 minutes of coding, debugging, and testing time. Although the system is flexible enough to support multiple dynamically-changing allocation routines, variable-time allocation memory areas use the `malloc`-based allocator, while linear-time allocation memory areas use the stack-allocator.

4.6.2 Allocating Memory Outside the Current Memory Area

The memory area's `newInstance` and `newArray` methods can allocate memory in an arbitrary memory area that is accessable from the current memory area. The methods use allocation information from the specified memory area without actually entering the memory area. Since the constructor for any object created in this manner is run in the preexisting current memory area for the current thread, `newInstance` and `newArray` do not actually enter the specified memory area. Objects allocated through these methods have the same lifetime as other objects allocated in the same memory area.

4.7 Garbage Collection

References from heap objects can point both to other heap objects and to objects allocated in immortal memory. The garbage collector must therefore recognize references to immortal memory and treat objects allocated in immortal memory differently than objects allocated in heap memory. In particular, the garbage collector cannot change the objects in ways that that would interact with concurrently executing no-heap real-time threads.

Our implementation handles this issue as follows. The garbage collector first scans the immortal and scoped memories to extract all references from objects allocated in these memories to heap allocated objects. The scan iterates through the allocation data structures allocated by the reference count memory allocator, calling each function to find the heap references in each allocation data area. This scan is coded to operate correctly in the presence of concurrent updates from no-heap real-time threads. The scan iterates through a linked-list of objects. If a concurrently running no-heap real-time thread allocates a new object after the iterator has started, the allocator adds the new object to the beginning of the list. The iterator does not iterate through this new object to find roots, but the new object cannot contain any roots into garbage collected space since only no-heap real-time threads, which cannot create or manipulate roots, have run since garbage collection began. Since the fields which are roots will remain roots and the fields which are not roots will still not point to the heap during the scan and the scan will always be able to distinguish between roots and non-roots, the scan can properly extract the heap references from the allocation data area. The garbage collector uses the extracted heap references as part of its root set.

Note a potential interaction between the garbage collector and no-heap real-time threads. The garbage collector may be in the process of retrieving the heap references stored in a memory area when a no-heap real-time thread (operating concurrently with or interrupting the garbage collector) allocates objects in that memory area. The garbage collector must operate correctly in the face of the resulting changes to the underlying memory area data structures. The system design also cannot involve locks shared between the no-heap real-time thread and the garbage collector (the garbage collector is not allowed to block a no-heap real-time thread). But the garbage collector may assume that the actions of the no-heap real-time thread do not change the set of heap references stored in the memory area.

During the collection phase, the collector does not trace references to objects allocated in immortal memory. If the collector moves objects, it may need to update references from objects allocated in immortal memory or scoped memories to objects allocated in the heap. It performs these updates in such a way that it does not interfere with the ability of no-heap real-time threads to recognize such references as referring to objects allocated in the heap. Note that because no-heap real-time threads may access heap references only to perform heap reference checks, this property ensures that the garbage collector and no-heap real-time threads do not inappropriately interfere.

4.8 Runtime Checks

The implementation uses dynamic checks to ensure that no-heap real-time threads do not interfere with the garbage collector and the program does not create dangling references. Heap reference checks ensure that no-heap real-time threads do not manipulate references to the heap. Assignment checks prevent the creation of dangling references by forbidding assignments that could cause a field of an object to point to an object with a shorter lifetime.

4.8.1 Heap Reference Check Implementation

The implementation must be able to take an arbitrary reference to an object and determine the kind of memory area in which it is allocated. To support this functionality, our implementation adds an extra field to the header of each object. This field contains a pointer to the memory area in which the object is allocated.

One complication with this scheme is that the garbage collector may violate object representation invariants during collection. If a no-heap real-time thread attempts to use the field in the object header to determine if an object is allocated in the heap, it may access memory rendered invalid by the actions of the garbage collector. We therefore need a mechanism which enables a no-heap real-time thread to differentiate between heap references and other references without attempting to access the memory area field of the object.

We first considered allocating a contiguous address region for the heap, then checking to see if the reference falls within this region. We decided not to use this approach because of potential interactions between the garbage collector and the code in the no-heap real-time thread that checks if the reference falls within the heap. Specifically, using this scheme would force the garbage collector to always maintain the invariant that the current heap address region include all previous heap address regions. We were unwilling to impose this restriction on the collector.

We then considered a variety of other schemes, but eventually settled on the (relatively simple) approach of setting the low bit of all heap references. The generated code masks off this bit before dereferencing the pointer to access the object. With this approach, no-heap real-time threads can simply check the low bit of each reference to check if the reference points into the heap or not.

Our current system uses the memory area field in the object header to obtain information about objects allocated in scoped memories and immortal memory. The basic assumption is that the objects allocated in these kinds of memory areas will never move or have their memory area field temporarily corrupted or invalidated.

Figure 4-4 presents the code that the compiler emits for each heap reference check; Figure 4-5 presents the code that determines if the current thread is a no-heap real-time thread. Note that the emitted code first checks to see if the reference is a heap reference — our expectation is that most Real-Time Java programs will manipulate relatively few references to heap-allocated objects. This expectation holds for our benchmark programs (see Section 6).

4.8.2 Assignment Check Implementation

The assignment checks must be able to determine if the lifetime of a scoped memory area A is included in the lifetime of another scoped memory area B. The implementation searches the thread's stack of memory areas to perform this check. It first searches for the occurrence of A closest to the start of the stack (recall that A may occur multiple times on the stack). It then searches to check if there is an occurrence of B between that occurrence of A and the start of the stack. If so, the assignment check succeeds; otherwise, it fails.

The current implementation optimizes this check by first checking to see if A and B are the same scoped memory area. Figure 4-6 presents the emitted code for the assignment checks, while Figure 4-7 presents some of the run-time code that this emitted code invokes.

4.9 Exiting the memory area

On exiting a memory area, the thread pops the memory area off of the memory area stack and its associated allocation routine and data structures are attached to the current thread. When a thread pops a memory area off of the stack, it sets the top of memory area stack pointer to point to the next item on the memory area stack and decrements the reference count of the previous memory area stack object. When an object's reference count in a reference count memory area becomes zero, the reference

READ	**WRITE**
use of `*refExp` in `exp`	`*refExp = exp;`
becomes:	becomes:
`heapRef = *refExp;`	`heapRef = *refExp;`
`if (heapRef&1)`	`if (heapRef&1)`
`   heapCheck(heapRef);`	`   heapCheck(heapRef);`
`[*heapRef/*refExp] exp`	`refExp = exp;`

NATIVECALL	**CALL**
`refExp = nativecall(args);`	`refExp = call(args);`
becomes:	becomes:
`heapRef = nativecall(args);`	`heapRef = call(args);`
`if (heapRef&1)`	`if (heapRef&1)`
`   heapCheck(heapRef);`	`   heapCheck(heapRef);`
`refExp = heapRef;`	`refExp = heapRef;`

METHOD

```
method(args) { body }

              becomes:

method(args) {
   for arg in args:
      if (arg&1)
         heapCheck(arg);
   body }
```

Figure 4-4: Emitted Code For Heap Reference Checks

```
#ifdef DEBUG
  void heapCheck(unwrapped_jobject* heapRef, const int source_line,
                 const char* source_fileName, const char* operation) {
#else       /* operation = READ, WRITE, CALL, NATIVECALL, or METHOD */
  void heapCheck(unwrapped_jobject* heapRef) {
#endif
    JNIEnv* env = FNI_GetJNIEnv();
    /* determine if in a NoHeapRealtimeThread */
    if (((struct FNI_Thread_State*)env)->noheap) {
       /* optionally print helpful debugging info */
       /* throw exception */
    }
  }
```

Figure 4-5: The `heapCheck` function

New Object (or Array):

```
  obj = new foo(); (or obj = new foo()[1][2][3];)
```

becomes:

```
  ma = RealtimeThread.currentRealtimeThread().getMemoryArea();
  obj = new foo(); (or obj = new foo()[1][2][3];)
  obj.memoryArea = ma;
```

Assignment check:

```
  obj.foo = bar;
```

becomes:

```
  ma = MemoryArea.getMemoryArea(obj);
  /* or ma = ImmortalMemory.instance(), if a static field) */
  ma.checkAssignment(bar);
  obj.foo = bar;
```

Figure 4-6: Emitted Code for Assignment Checks

In MemoryArea:

```
public void checkAssignment(Object obj) {
  if ((obj != null) && (obj.memoryArea != null) &&
       obj.memoryArea.scoped) {
    /* Helpful native method prints out all debugging info. */
    throwIllegalAssignmentError(obj, obj.memoryArea);
  }
}
```

Overridden in ScopedMemory:

```
public void checkAssignment(Object obj) {
  if (obj != null) {
    MemoryArea target = getMemoryArea(obj);
    if ((this != target) && target.scoped &&
        (!RealtimeThread.currentRealtimeThread()
         .checkAssignment(this, target))) {
        throwIllegalAssignmentError(obj, target);
    }
  }
}
```

In RealtimeThread:

```
boolean checkAssignment(MemoryArea source, MemoryArea target) {
  MemBlockStack sourceStack = (source == getMemoryArea()) ?
    memBlockStack : memBlockStack.first(source);
  return (sourceStack != null) &&
         (sourceStack.first(target) != null);
}
```

Figure 4-7: Code for performing assignment checks

count memory area runs its finalizers for the object. The default finalizer for an object in a reference count memory area traces the objects for other pointers to reference count memory area allocated objects and decrements their references. When a thread exits, the thread exits all memory areas entered during the duration of the thread. If the memory area stack objects are not shared with any other thread, the reference count memory area can reclaim the memory associated with them, thus removing any roots to the associated memory areas from the root set sent to the garbage collector. Also, when exiting a memory area, a thread decrements the memory area's entry count.

If the entry count becomes zero, then the memory area runs the finalizers for all contained objects and resets the memory associated with them (which may involve zeroing memory). Any attempt to reenter the memory area while the contained objects' finalizers are running will block until the finalizers finish, in accordance with the Real-Time Specification for Java [3]. This blocking creates a potential danger for the Real-Time Java programmer.

Consider the following situation. A thread exits a memory area, causing its reference count to become zero, at which point the implementation starts to invoke finalizers on the objects in the memory area as part of the deallocation process. While the finalizers are running, a no-heap real-time thread enters the memory area. According to the Real-Time Java specification, the no-heap real-time thread blocks until the finalizers finish running. There is no mention of the priority with which the finalizers run, raising the potential issue that the no-heap real-time thread may be arbitrarily delayed. A final problem occurs if the no-heap real-time thread first acquires a lock, a finalizer running in the memory area then attempts to acquire the lock (blocking because the no-heap real-time thread holds the lock), then the no-heap real-time thread attempts to enter the memory area. The result is deadlock — the no-heap real-time thread waits for the finalizer to finish, but the finalizer waits for the no-heap real-time thread to release the lock.

If the entry count is zero, all user inaccessable references to the memory area object are destroyed to help shorten the lifetime of memory area objects. If the thread reenters the memory area, the thread reestablishes the references to the memory area object.

4.10 The memory object becomes inaccessable

If the Real-Time Java implementation does not store any pointers to user-allocated memory area objects, then the memory area objects may be garbage-collected at an inappropriate time. The Real-Time Specification for Java [3] requires that a scoped memory's `getOuterScope()` method return a pointer to the enclosing scoped memory of the current memory area, or to the current memory area if there is no enclosing scope. However, the user may allocate a scoped memory area object in garbage-collected heap memory. If the system does not keep a reference to this object (reachable from the garbage collector rootset), the memory area object may become inaccessable to the user, and the garbage collector could deallocate it. For instance, a scoped memory area object may be referenced only by a local variable in a method. The method may start a thread which enters the scoped memory. The thread may enter another scoped memory later. The user may invoke the `getOuterScope()` method and the object may not be found. The user may also invoke a real-time thread's `getMemoryArea()` method, which returns the current memory area even if another memory area was entered and exited, returning to the original memory area. Therefore, all heap-allocated memory area objects that were entered but not exited in any thread must be kept alive with a reference to the original memory area object that is scanned by the garbage collector, even if shadowed memory areas or unique ID's are used internally to avoid memory area inaccessability problems when doing assignment checks in no-heap real-time threads.

The lifetime of a memory area should be at least the maximum of the lifetime on the memory area stack, the lifetime of the references to it from user space, and the internal references which can be returned to user space, such as the current memory area object attached to a particular thread which can be returned by `getMemoryArea()`. Linear-time allocated scoped memory commits a specified initial amount of memory by the completion of the constructor, which must be available for allocation even if all objects in the memory area have been finalized and the memory area reset. The memory from the initial requirement cannot be reclaimed until the memory area is no longer user-accessable and cannot be found on the memory area stack. Since the memory area could have been heap allocated, garbage collector finalizers must run to deallocate the storage associated with the memory area. To prevent costly memory leaks, the garbage collector and the memory area implementations must provide support for some form of finalization (not necessarily the finalization described in the Java Language Specification). Since immortal memory finalizers cannot be run until the program has finished, memory areas that have a lifetime less than the entire program should not be allocated in immortal memory. No-heap real-time threads cannot access heap memory, so limited-lifetime scoped memory area objects used in no-heap real-time threads should be allocated in scoped memory or reused if allocated in immortal memory.

If the entry count is zero and the memory area object becomes inaccessable, the memory area object finalizer decrements the allocation data structures reference count. The memory area object finalizer also reclaims memory associated with its shadow if the memory area object was allocated in garbage collected space, and deflates the memory area object. When no iterator of allocation data structures, such as the garbage collector, is currently running, then the allocation data structures with zero reference count can be finalized.

The only reason for the allocation data structure reference count is to eliminate the problem of the garbage collector (or any other iterator) containing a reference to the allocation data structures for scanning purposes while the allocation data structures are finalized and the memory associated with them are freed. Since the finalizable allocation data structures cannot contain any live objects at this time, the garbage collector does not need to scan any objects contained in the finalizable allocation data structures. Therefore, any solution which ensures that the garbage collector does not attempt to dereference a pointer to finalized allocation data structures while scanning for roots is acceptable.

After the allocation data structure reference count drops to zero and no iterator is scanning the allocation data structures (the reference count memory area for allocation data structures' iterator count is zero), the reference count memory area for allocation data structures runs a generic allocation data structure finalizer. This finalizer runs a memory area specific allocation data structure finalizer which can reclaim the memory associated with the memory area. The generic allocation data structure finalizer can then deallocate the rest of the allocation data structure.

4.11 Implementation Nuances

Although the Real-Time Specification for Java [3] consistent real-time performance, and explicit control over memory through an approach that is backwards-compatible with existing Java programs, it restricts the design space of the implementation and may limit the portability of libraries offering real-time performance guarantees in the following ways:

- Since assignment checks may require information from previous threads frozen at the start of a new thread to determine the legality of an assignment in the new thread, the memory area stack cannot simply be a call stack annotated with memory areas.
- Since no-heap real-time threads cannot access heap-allocated scoped memory area objects, an object may be accessable but the memory area which allocated it may not. Therefore, assignment checks cannot always access the memory area that allocated an object.
- Thread-local state implementations may be more complicated than implementations which deal with synchronization issues directly.
- Since a linear-time allocation memory area is the only type of memory area with allocation performance guarantees, and it requires an initial memory size to perform consistently, the Real-Time Java library developer must use compiler and runtime platform-specific information to ensure statically the efficient

and consistent operation of real-time library functions which allocate memory internally. The size requirement may limit the portability of libraries offering real-time performance guarantees.

Future research may provide solutions to overcome current limitations implied by the specification.

4.11.1 The memory area stack

The order of memory area entry provides a thread-local context for determining whether an object in one scoped memory area can have a pointer to an object in another scoped memory area. Since the assignment check relationship between scoped memory areas is dependent on the order the scoped memory areas were entered for the given thread, scoped memory areas can be shared between threads, and scoped memory areas can be entered in any arbitrary order, storing a pointer to a memory area's parent in a field of the memory area is neither sufficient for determining the assignment relationship between scoped memory areas nor for determining which memory area to return after exiting the current memory area.

For example, thread A enters scoped memory area one then enters scoped memory area two. Thread B enters scoped memory area two then enters scoped memory area one. In thread A, scoped memory area two's parent is scoped memory area one. In contrast, in thread B, scoped memory area one's parent is scoped memory area two. If thread C enters scoped memory area three then enters scoped memory area one, which is the parent for scoped memory area one? In thread B, the parent is scoped memory area two, but in thread C, the parent is scoped memory area three. Therefore, the "parent" of a memory area is a thread-local concept, which can be maintained by keeping a stack of memory areas per thread.

Since the stack of memory areas must be maintained per thread, and memory areas are entered when a `run` method is executed and exited when the `run` method returns, an implementor of a Real-Time Java system may be tempted to just use thread's call stack to keep track of the current memory area and the stack of previous memory areas. Unfortunately, information from prior threads may be necessary to determine whether a particular scoped memory to scoped memory assignment check is valid. For example, thread A enters scoped memory area one and starts thread B which enters scoped memory area two. Thread A then exits scoped memory area one and enters scoped memory area two. Thread B attempts to make an assignment from scoped memory area two to scoped memory area one. To determine the legality of this assignment, thread B must examine not only its own call stack, but the call stack of thread A *at the time which thread B was started.* If thread B examines thread A's current call stack, it will not find an instance of scoped memory area one (which was replaced on the call stack by scoped memory area two), and will conclude that the assignment is illegal, when it is in fact legal. Therefore, thread B's memory area stack must not only contain pointers to active memory areas entered during the execution of thread B, but also pointers to all active previous memory areas of all threads prior to thread B, frozen at the point of spawning the next thread.

A naive implementation may just copy the previous call stacks, making a thread-local list of call stacks to be used in addition to the current call stack. Copying call stacks can needlessly waste memory. For example, thread A enters scoped memory area one and spawns thread B. Thread B enters scoped memory area two and spawns thread C... until thread G. Thread G has seven call stacks, F has six, E has five... and A has one. Twenty-eight call stacks have been allocated for only seven memory areas and seven threads. If each thread maintains a singly-linked list of previous memory areas, the threads can share portions of this list, forming a cactus stack of memory areas. In the prior example, using a cactus stack, the Real-Time Java system would allocate only seven memory area stack objects, each containing a pointer to a memory area and a pointer to the next stack object, in addition to the seven thread-local call stacks instead of twenty-eight entire call stacks each augmented with memory area pointers.

4.11.2 Memory area accessability and lifetime

Unfortunately, according to the Real-Time Specification for Java [3], an object can be accessable while its memory area may be inaccessable. For instance, an object can be allocated out of a heap-allocated scoped memory. The object is accessable from a no-heap real-time thread, but its memory area is not. While tracing the stack of memory areas from a scoped memory to another scoped memory to determine whether an assignment is possible, the Real-Time Java implementation may attempt to dereference a pointer to a heap-allocated memory area object in a no-heap real-time thread.

Two possible solutions to this problem are: 1) to store unique ID's instead of the actual memory areas internally and create maps from ID's to the state contained in the memory area, allocated in accessable memory areas, or 2) create a shadow object of each heap-allocated memory area which has the same lifetime as the original object, but is allocated in non-movable memory and contains pointers to the original, non-movable allocator data structures. Our implementation uses the second, simpler approach.

4.11.3 Thread-local state implementations beware

Possible dead-locks or pauses caused by the use of blocking locks held between real-time threads, no-heap real-time threads, and the garbage collector limits memory allocation algorithms to use only non-blocking synchronization based on primitives which possibly have no performance guarantees at all on some multi-processor systems. Perhaps the implementation of the memory allocation system could be based on thread-local state, avoiding all synchronization issues and multiprocessor contention, simultaneously making implementation of allocation routines easier, permitting more design freedom and early reclamation of some thread-local objects in memory areas, and facilitating reuse of portions of the memory allocated for memory areas. In a thread-local state implementation, the garbage collector may not have to scan portions of memory areas associated with objects local to no-heap real-time threads, further improving overall program performance.

We initially attempted a thread-local state implementation, hoping for the aforementioned advantages. We created a new block every time a thread entered a memory area. The implementation deallocated the blocks when a thread exited. To determine the lifetime of objects allocated outside the current memory area, the implementation scanned for an active block corresponding to the closest copy of the desired memory area on the memory area stack and allocated the memory from that block. We failed for the following reasons:

- Unfortunately, two threads could allocate memory in the same block, since two threads with a common parent can share the same history of entered memory areas. If the implementation created a new thread-local block for an object allocated outside the current memory area, then when can the implementation deallocate that block?
- Memory area entry counts must also be shared between threads, since otherwise the implementation may not be able to determine when the memory area is inactive.
- Real-time threads can store heap references in fields of objects contained in no-heap real-time thread allocated blocks. Therefore, no-heap real-time thread allocated blocks must be scanned by the garbage collector.

Considering the tremendous complexity of the flawed thread-local implementation designed to avoid the complexity of dealing with possible race conditions between threads, we decided to settle on a simpler implementation that dealt with contention between threads directly using non-blocking synchronization.

4.11.4 Fixed-size memory areas

To determine the size of linear-time allocation memory areas, the benchmarks use input from the user and crash if the program runs out of memory. This obviously cannot be a long-term solution to determining the size of memory areas. Unfortunately, the size of objects is not specified by the Java Language Specification. Therefore, many compilers optimize objects in different ways to increase performance, leading to different object sizes. A programmer cannot determine a priori how much space a program will use.

How shall a programmer determine the size of memory areas? The compiler may be able to calculate the size of a memory area as a function of input parameters to a method, provided that the memory area does not contain a `while` loop which allocates memory and terminates depending on a condition not predictable statically (like when a robot runs into the wall). Perhaps the memory usage of some portions of a program can be determined while the memory usage of other areas cannot, restricting the use of linear-time allocation memory areas. In some Java implementations, objects can be inflated, or grow in size dynamically to accomodate additional fields that are known only to the runtime. Our runtime does not allocate the extra space out of linear-time allocation storage, but the real-time programmer must be aware that the memory used by the runtime can grow with the number of objects allocated beyond the sum of the sizes of the memory areas. The compiler may determine the size of a given memory area, but the programmer may wish to know that size to help determine statically what input size may cause the program to run out of memory, accounting for the possible overhead of object inflation.

A standard library may facilitate programming with no-heap real-time threads. To make the standard library usable in no-heap real-time threads, it must have strict performance guarantees. Therefore, the programmers of the standard library may wish to use linear-time allocation to satisfy performance requirements. No other form of allocation in the Real-Time Specification for Java [3] has performance bounds. However, the sizes of the linear-time allocation memory areas the standard library may use are not portable between compilers, limiting the portability of the standard library. If the compiler specifies the sizes of the linear-time allocation memory areas used in the standard library, the programmer may wish to know the memory usage of various library functions to optimize the memory usage of the program, particularly if the memory usage cannot be exactly described in the specification for the library routines. However, if the programmer uses this information to optimize the memory usage of the program, the program may not perform consistently across platforms.

A solution to the problem of specifying the sizes of linear-time allocation memory areas may be a prerequisite to the wide-scale use of performance bounded allocation in Real-Time Java.

Chapter 5

Developing Real-Time Java Programs

An additional design goal becomes extremely important when actually developing Real-Time Java programs: ease of debugging. During the development process, facilitating debugging became a primary design goal. In fact, we found it close to impossible to develop error-free Real-Time Java programs without some sort of assistance (either a debugging system or static analysis) that helped us locate the reason for our problems using the different kinds of memory areas. Our debugging was especially complicated by the fact that the standard Java libraries basically don't work at all with no-heap real-time threads.

5.1 Incremental Debugging

During our development of Real-Time Java programs, we found the following incremental debugging strategy to be useful. We first stubbed out all of the Real-Time Java heap reference checks and assignment checks and special memory allocation strategies, in effect running the Real-Time Java program as a standard Java program. We used this version to debug the basic functionality of the program. We then added the heap reference checks and assignment checks, and used this version to debug the memory allocation strategy of the program. We were able to use this strategy to divide the debugging process into stages, with a manageable amount of bugs found at each stage.

It is also possible to use static analysis to verify the correct use of Real-Time Java scoped memories [9]. We had access to such an analysis when we were implementing our benchmark programs, and the analysis was very useful for helping us debug our use of scoped memories. It also dramatically increased our confidence in the correctness of the final program, and enabled a static check elimination optimization that improved the performance of the program.

5.2 Additional Runtime Debugging Information

Heap reference checks and assignment checks can be used to help detect mistakes early in the development process, but additional tools may be necessary to understand and fix those mistakes in a timely fashion. We therefore augmented the memory area data structure to produce a debugging system that helps programmers understand the causes of object referencing errors.

When a debugging flag is enabled, the implementation attaches the original Java source code file name and line number to each allocated object. Furthermore, with the use of macros, we also obtain allocation site information for native methods. We store this allocation site information in a list associated with the memory area allocation structure which allocated the object. Given any arbitrary object reference, a debugging function can retrieve the debugging information for the object. Using a list of all allocation memory area allocation structures, the debugging system can dump a map of all allocated objects and which memory area and allocation site allocated them. Combined with a stack trace at the point of an illegal assignment or reference, the allocation site information from both the source and destination of an illegal assignment or the location of an illegal reference can help quickly determine the exact cause of the error and the objects responsible. The debugging system can also display allocation site information at the time of allocation to provide a program trace which can help determine control flow, putting the reference in a context at the time of the error.

5.3 Interaction With the Debugging System

When initially debugging the benchmark Array with no-heap real-time threads enabled, the program suddenly crashed. The debugging system produced the following error message:

```
attempted heap reference 0x40262d71 in Java code at
MemoryArea.java:139, pointer at 0x40262d71,
found in MemBlock = 0x083485e8,
allocated during the main thread in the initial HeapMemory
at location ImmortalMemory.java:30 pointing to a location
of size 41 bytes
```

After looking at the initial allocation of the `ImmortalMemory` object on line 30 of `ImmortalMemory.java`, we immediately understood that the system had allocated the `ImmortalMemory` instance from heap memory and should allocate it from immortal memory. We fixed the problem within five minutes. The debugging system also alerted us to the fact that native methods must also allocate objects from memory areas, and the Java Native Interface (JNI) functions must allocate objects using functions aware of the current Real-time Java allocation function. The informative error messages promoted rapid development of both the benchmarks and the Real-Time Java memory management system.

5.4 Experience With the Sun JDK

Unfortunately, the Sun JDK version 1.3 was never designed to support Real-Time Java. The first problem encountered was that every `Thread` adds itself to the main `ThreadGroup`. Unfortunately, the main `ThreadGroup`, allocated from heap memory, cannot store pointers to `Threads` allocated in scoped memory. In fact, any Real-Time Java thread manager written in Java cannot store pointers to program-allocated `Thread` objects. How can a thread manager written in Java determine which thread to run if it cannot maintain an active thread list in Java? The answer is to maintain the list in native code and be very careful about dereferencing pointers to thread objects, especially when interrupting the garbage collector to run a no-heap real-time thread.

The Sun JDK also presented other problems when running with no-heap real-time threads. The constructor for the `Thread` object uses `Integer.toString(int)` to form the thread name from its integer unique ID. `Integer.toString(int)` and `Integer.parseInt(String)` do not work in no-heap real-time threads since the `digit` method of `Character` loads a static reference to the heap, `Character.A`. `Double.toString(double)` does not work because `FloatingDecimal.long5pow` is a static reference to the heap. The greatest impediment to the early stages of debugging no-heap real-time thread support was the fact that `System.out` is a static reference to the heap, preventing `System.out.println` from working. In general, most problems with no-heap real-time threads resulted from the fact that static initializers are run in the initial memory area, `HeapMemory`. Unfortunately, running all static initializers in `ImmortalMemory`, in addition to violating the specification, may needlessly waste memory. A new real-time library usable from no-heap real-time threads would greatly facilitate development of large applications in Real-Time Java.

Chapter 6

Results

We implemented the Real-Time Java memory extensions in the MIT Flex compiler infrastructure. [1] Flex is an ahead-of-time compiler for Java that generates both native code and C; it can use a variety of garbage collectors. For these experiments, we generated C and used the Boehm-Demers-Weiser conservative garbage collector and a stop and copy collector.

We obtained several benchmark programs and used these programs to measure the overhead of the heap reference checks and assignment checks. Our benchmarks include Barnes, a hierarchical N-body solver, and Water, which simulates water molecules in the liquid state. Initially these benchmarks allocated all objects in the heap. We modified the benchmarks to use scoped memories whenever possible. We also present results for two synthetic benchmarks, Tree and Array, that use object field assignment heavily. These benchmarks are designed to obtain the maximum possible benefit from heap reference check and assignment check elimination. We modified Tree, Array, and Water to use no-heap real-time threads. We did not modify Barnes to use no-heap real-time threads. As a result, even though we report the performance of Barnes with no-heap real-time thread support, we do not expect Barnes to perform like a typical application designed for use with no-heap real-time threads.

We ran all benchmarks on a Dell Dimension 4100 with a Pentium III 866 MHz

[1] Available at `www.flexc.lcs.mit.edu`

Table 6.1: Number of Objects Allocated In Different Memory Areas

Benchmark	Version	Heap	Scoped	Immortal	Total
Array	heap	13	4	0	17
	no-heap	0	4	6	10
Tree	heap	13	65,534	0	65,547
	no-heap	0	65,534	6	65,540
Water	heap	406,895	3,345,711	0	3,752,606
	no-heap	0	3,328,086	405,647	3,733,733
Barnes	heap	16,058	4,681,708	0	4,697,766
	no-heap	16,058	4,681,708	0	4,697,766

Table 6.2: Number of Arrays Allocated In Different Memory Areas

Benchmark	Version	Heap	Scoped	Immortal	Total
Array	heap	36	4	0	40
	no-heap	0	4	56	60
Tree	heap	36	0	0	36
	no-heap	0	0	56	56
Water	heap	405,943	13,160,641	0	13,566,584
	no-heap	0	13,160,641	403,696	13,564,337
Barnes	heap	14,871	4,530,765	0	4,545,636
	no-heap	14,871	4,530,765	0	4,545,636

processor, a 256 KB CPU cache and 512 MB RAM running RedHat 7.0 (Linux 2.2.16-22smp kernel) at run-level 3. We compiled the benchmarks against the Sun JDK 1.3 class files using the FLEX compiler and gcc-2.95.3 with the -O9 option. Copies of all benchmarks used are available upon request.

Table 6.1 presents the number of objects we were able to allocate in each of the different kinds of memory areas for the no-heap real-time thread and real-time thread versions of each benchmark. The goal is to allocate as many objects as possible in scoped memory areas; the results show that we were able to modify the programs to allocate the vast majority in scoped memories. Java programs also allocate arrays;

Table 6.2 presents the number of arrays that we were able to allocate in scoped memories. As for arrays, we were able to allocate the vast majority in scoped memories. The no-heap real-time thread versions of Array, Tree, and Water allocate objects and arrays that were on the heap in immortal memory. The "no-heap real-time thread" version of Barnes has many heap allocated objects, since it is the same program as the "real-time thread" version. Note that these statistics do not account for objects and arrays generated by native code or by static initializers.

6.1 Assignment Checks

Storing a reference to an object with a shorter lifetime from an object with a longer lifetime may create dangling references. The Real-Time Specification for Java [3] forbids the creation of dangling references. Therefore, our compiler inserts dynamic checks on every assignment to ensure this invariant. Unfortunately, these dynamic checks incur a significant performance overhead.

Table 6.3 presents the number and type of assignment checks encountered during the execution of each benchmark. Recall that there is a check every time the program stores a reference. The different columns of the table break down the checks into categories depending on the target of the store and the memory area that the stored reference refers to. For example, from scoped to heap counts the number of times the program stored a reference to heap memory into an object or array allocated in a scoped memory. The no-heap real-time thread versions of Array, Tree, and Water assign to and from immortal memory where the real-time thread version assign to and from heap memory. The "no-heap real-time thread" version of Barnes is the same program as the real-time thread version. Note that these statistics do not account for objects and arrays generated by native code or by static initializers.

Table 6.4 presents the running times of the benchmarks with no-heap real-time support disabled. We report results for eighteen different versions of the program. The three rows for each benchmark indicate the use of the BDW garbage collector with heavy threads (Heavy), the BDW garbage collector with user threads (User), and

Table 6.3: Assignment Counts

		To		
Benchmark(Version)	From	Heap	Scoped	Immortal
Array(heap)	Heap	14	forbidden	8
	Scoped	0	400,040,000	0
	Immortal	0	forbidden	0
Array(no-heap)	Heap	0	forbidden	12
	Scoped	0	400,040,000	0
	Immortal	0	forbidden	23
Tree(heap)	Heap	14	forbidden	8
	Scoped	0	65,597,532	65,601,536
	Immortal	0	forbidden	0
Tree(no-heap)	Heap	0	forbidden	12
	Scoped	0	65,597,532	65,601,536
	Immortal	0	forbidden	23
Water(heap)	Heap	409,907	forbidden	0
	Scoped	17,836	9,890,211	844
	Immortal	3	0	1
Water(no-heap)	Heap	0	forbidden	6
	Scoped	0	9,890,211	18,680
	Immortal	1	forbidden	408,306
Barnes(heap)	Heap	90,856	forbidden	80,448
	Scoped	9,742	4,596,716	1328
	Immortal	0	forbidden	0
Barnes(no-heap)	Heap	90,856	forbidden	80,448
	Scoped	9,742	4,596,716	1328
	Immortal	0	forbidden	0

Table 6.4: Execution Times of Benchmark Programs

		With Checks			Without Checks		
Benchmark	Version	Heap	VT	LT	Heap	VT	LT
Array	Heavy	28.7	43.3	43.9	7.9	7.9	8.0
	User	28.1	43.0	43.1	7.7	7.7	8.0
	Copy	151.1	192.7	194.1	16.9	17.9	17.9
Tree	Heavy	13.4	16.9	16.9	7.0	7.0	7.0
	User	13.3	16.8	16.8	7.0	7.0	7.0
	Copy	49.4	59.2	55.2	12.2	16.7	12.3
Water	Heavy	177.6	173.7	163.7	133.6	125.7	115.2
	User	99.9	90.6	79.6	77.3	67.6	58.1
	Copy	96.8	109.5	95.1	75.2	85.6	72.0
Barnes	Heavy	51.3	41.4	35.0	44.5	34.5	27.7
	User	39.1	25.8	19.6	38.9	25.3	19.0
	Copy	29.7	36.1	29.9	22.5	28.3	22.3

the stop-and-copy garbage collector with user threads (Copy). The first three columns all have assignment checks, and vary in the memory area they use for objects that we were able to allocate in scoped memory. The Heap version allocates all objects in the heap. The VT version allocates scoped-memory objects in instances of `VTMemory` (which use `malloc`-based allocation); the LT version allocates scoped-memory objects in instances of `LTMemory` (which use stack-based allocation). The next three versions use the same allocation strategy, but the compiler generates code that omits the assignment checks. For our benchmarks, our static analysis is able to verify that none of the assignment checks will fail, enabling the compiler to eliminate all of these checks [9].

These results show that assignment checks add significant overhead for all benchmarks. But the use of scoped memories produces significant performance gains for Barnes and Water when running with user threads and the BDW garbage collector. In the end, the use of scoped memories without checks significantly increases the overall performance of the program using the BDW garbage collector. To investigate the causes of the performance differences, we instrumented the run-time system to measure the garbage collection pause times. Based on these measurements, we attribute most of the performance differences between the versions of Water and Barnes with and without scoped memories to garbage collection overheads. Specifically, the use of scoped memories improved every aspect of the BDW garbage collector: it reduced the total garbage collection overhead, increased the time between collections, and significantly reduced the pause times for each collection.

BDW is a conservative mark-and-sweep garbage collector for C. The BDW garbage collector conservatively scans all memory for possible pointers into the garbage-collected heap. BDW stops all threads, uses heuristics to identify pointers during a scan, then a generational mark-and-sweep algorithm to collect garbage. Since C does not tag pointers separately from numbers in memory, BDW must conservatively guess whether a given 32-bit integer in memory is actually a pointer to garbage-collected memory. Since the BDW collector stops all threads, scans all memory, and examines headers preceding allocated memory during collection, it cannot currently support concurrent running of no-heap real-time threads.

Therefore, we used another garbage collector based on a simple version of the stop and copy algorithm to support no-heap real-time threads. Our stop-and-copy garbage collector stops all real-time threads (but not no-heap real-time threads) by setting a flag which all threads check periodically. It waits for all threads to notice the flag and stop before proceeding with garbage collection. During the garbage collection scan, the stop-and-copy garbage collector calls a function in the Real-Time Java implementation to add all heap references from scoped and immortal memory areas to the rootset.

The use of linear-time allocation memory areas with the stop and copy garbage collector did not show significant performance improvements, since the current implementation incurs significant overhead for scanning for roots into garbage-collected space from linear-time memory areas. Optimization of the scanning routine may improve performance. Optimizing the linear-time allocator to remove unnecessary rezeroing of memory in the presence of precise garbage collection may also lead to performance increases.

Notice that the heavy (POSIX) thread versions of Water and Barnes use significantly more time than their user thread counterparts. The user thread manager only switches threads when a thread attempts to grab a lock and must wait for another thread to release it, or when the thread blocks for I/O. Since Water and Barnes do not use significant I/O after the benchmarks load initial parameters, and each thread rarely waits for a lock, fewer switches between threads occur when using user threads than heavy threads. Since threads share memory areas in both benchmarks, reducing thread switching can also reduce memory usage. When the reference count of a memory area reaches zero, the memory area finalizes and deallocates all contained objects. The memory area resets and the program can reuse the memory associated with the memory area. Thus, running threads serially reduces memory use. Also, heavy thread switching involves the overhead of operating system calls whereas user thread switching does not. Unfortunately, real-time performance constraints may require a real-time thread manager to switch threads even when no thread is blocked on I/O or waiting for a lock.

For Array and Tree, there is almost no garbage collection for any of the versions and the versions without checks all exhibit basically the same performance. With checks, the versions that allocate all objects in the heap run faster than the versions that allocate objects in scoped memories. We attribute this performance difference to the fact that heap to heap assignment checks are faster than scope to scope assignment checks.

Notice that, while running the stop-and-copy garbage collector, the versions of Array and Tree with assignment checks run much slower than the versions without assignment checks. The stop-and-copy garbage collector stops threads by setting a flag which the program threads check periodically. The compiler inserts flag checks at the beginning of every method, inside every loop, after a method call, and when the method returns. In summary, it inserts flag checks at all program points with outside edges in the control flow graph. Our implementation uses Java code including several method calls and a loop for assignment checks. Therefore, the assignment checks are a target for the compiler pass that inserts garbage collector flag checks and each assignment check may involve several garbage collector flag checks. Since every assignment requires an assignment check, the program checks the garbage collector many times during the execution of the program. This is an easy problem to fix: simply do not emit garbage collector flag checks within assignment checks. However, this problem illustrates an important point. Without analyzing the interactions between checks, the effect on performance of adding a new type of check is multiplicative.

6.2 Heap Reference Checks

No-heap real-time threads run asynchronously with the garbage collector and cannot manipulate references to the heap, since the heap may be in an inconsistent state during a garbage collection. The Real-Time Specification for Java [3] does not permit no-heap realtime threads to manipulate reference to the heap. Therefore, our compiler inserts dynamic checks on every heap reference to ensure this invariant. There are five ways a thread could illegally obtain or manipulate a reference to the heap:

- A method call to a native method could return a heap reference (**CALL**).
- A call to the runtime could return a heap reference (**NATIVECALL**).
- Arguments to a method may point to the heap (**METHOD**).
- A read of a field may return a heap reference (**READ**).
- A write to a field may overwrite a heap reference (**WRITE**).

Unfortunately, heap reference checks incur a significant performance overhead.

Table 6.5 presents the number and type of heap reference checks encountered during the execution of each benchmark. Note that most heap reference checks occur during a read of a field. Array and Tree copy data structures, so each encounters as many reads as writes. This is not true for Water, which is more similar to typical programs. Note that Tree checks many arguments to methods. Every call to construct a new tree element object involves passing in multiple arguments. Also note the total

Table 6.5: Heap Check Counts

Benchmark	**CALL**	**METHOD**	**NATIVECALL**
Array	101	560	33
Tree	131,171	87,773,758	33
Water	50,582,452	1,739,663	71,934
Barnes	5,081,428	58,921,280	340,642

Benchmark	**READ**	**WRITE**	Total
Array	400,000,359	400,040,120	800,041,173
Tree	174,756,342	131,199,188	393,860,492
Water	353,089,248	10,642,997	416,126,294
Barnes	103,855,249	6,310,800	174,509,399

number of checks encountered ranges from 300–800 million for the three benchmarks listed.

Table 6.6 presents the number and type of heap references encountered during the execution of each benchmark. Since the main thread is always a real-time thread which starts execution in the heap, the number of heap references for no-heap real-time thread enabled benchmarks is nonzero. Note that Array, Tree, and Water only encounter 500–1,300 actual references to the heap. Most heap checks encountered in both real-time threads and no-heap real-time threads are for references to non-heap allocated objects in programs designed to use no-heap real-time threads extensively. Therefore, our implementation of heap checks, which tests the reference to see whether it points to the heap before testing whether the current thread is a no-heap real-time thread, makes sense. Most checks will just check the low bit of the pointer and move on.

Notice that the "no-heap real-time thread" version of Barnes (which is the same as the real-time thread version) has 85 million heap references. The large number of heap references in an unmodified program and the small number of heap references in a modified program illustrate the changes that occur when a programmer modifies a program to use no-heap real-time threads extensively.

Table 6.6: Heap Reference Counts

Benchmark	**CALL**	**METHOD**	**NATIVECALL**
Array	25	340	9
Tree	25	336	9
Water	27	631	7
Barnes	3,016,797	28,942,950	333

Benchmark	**READ**	**WRITE**	Total
Array	188	6	568
Tree	171	6	547
Water	565	7	1,237
Barnes	53,410,543	234,267	85,604,890

Table 6.7: Overhead of Heap Reference Checks

		With		Without		With	
	Thread	Heap Checks		Heap Checks		All Checks	
Benchmark	Type	VT	LT	VT	LT	VT	LT
Array	Heavy	38.3	38.4	32.0	32.0	560.9	567.3
	User	38.2	38.5	32.0	32.0	244.6	249.2
Tree	Heavy	44.5	28.3	29.1	25.3	174.6	159.9
	User	24.0	16.6	18.1	14.3	78.8	70.7
Water	Heavy	170.7	147.9	162.7	137.8	229.2	201.7
	User	97.1	86.3	88.6	78.3	128.3	116.3

Table 6.7 presents the overhead of heap reference checks for each of the three benchmarks which use no-heap real-time threads. The first two columns present performance with heap reference checks. The second two columns present performance of the benchmarks compiled without emitting heap reference checks. The third two columns present the performance of adding assignment checks and heap reference checks. The first of each pair of columns lists performance numbers for benchmarks using variable-time scoped memory areas. The second of each pair of columns lists performance numbers for benchmarks using linear-time scoped memory areas. The second column lists the type of the threads used in each benchmark, user thread or heavy (POSIX) threads.

Heavy threads incur a significant performance overhead for both Tree and Water. The memory usage of Tree doubled when using heavy threads instead of user threads. Since Tree uses no blocking locks, and the memory allocation system does not use blocking locks, the user thread manager only switches between the two threads when the first thread finishes. Since two threads share a single memory area in Tree, switching threads can use twice as much memory as running on thread serially. When one thread runs after another, the memory area resets and can reuse the same memory. Unfortunately real-time constraints may require a real-time thread manager to switch threads often.

Overall, performance improved with the use of linear-time allocation memory areas over variable-time allocation memory areas, as expected. In our implementation, linear-time allocation memory areas bump a stack pointer to allocate memory, whereas variable-time allocation memory areas call `malloc` to allocate memory.

Note the significant overhead when running with both assignment checks and heap reference checks enabled. The last two columns of table 6.7 demonstrate the multiplicative effect of the overhead of adding different types of checks.

Chapter 7

Conclusion

The Real-Time Specification for Java [3] promises to bring the benefits of Java to programmers building real-time systems. One of the key aspects of the specification is extending the Java memory model to give the programmer more control over memory management. Since Real-Time Java is a strict superset of ordinary Java, performance-critical sections of the program can run with extensions which offer real-time performance guarantees concurrently with non-critical sections which run in ordinary Java. Use of the extensions offers predictable performance.

We have implemented these extensions. We found that the primary implementation complication was ensuring a lack of interference between the garbage collector and no-heap real-time threads, which execute asynchronously. Since the use of blocking locks in the allocation system can introduce complications because of interactions between the garbage collector, no-heap real-time threads, and real-time threads, the implementation used various non-blocking synchronization techniques.

A second implementation complication was determining how to manage the memory for the memory management and general implementation data structures. The initial candidates were the memory areas, but for some data structures, no memory area described in the specification seemed appropriate. In particular, the Real-Time Specification for Java [3] seems to have no memory area which any type of thread can access after entering any type of memory area and can deallocate memory before the program terminates. We invented a new memory area to solve this problem.

Programs requiring performance guarantees often also require safety guarantees. Static program analysis can prove the safety of programs and identify all potential illegal accesses or assignments. Unfortunately, static correctness analyses can sometimes constrain the program in more ways than dynamic checks. If a program's correctness cannot be proven with the static analysis tools at hand, then dynamic checks and debugging tools should provide detailed information to locate quickly the source of errors. Unfortunately, dynamic checks and debugging tools cannot offer the safety guarantees that static analysis can.

We found debugging tools necessary for the effective development of programs that use the Real-Time Java memory management extensions. We used both a static analysis and a dynamic debugging system to help locate the source of incorrect uses of these extensions. Incremental debugging and the ability to selectively enable or disable parts of the extensions was useful for debugging both applications and the implementation. Allocation site traces, pointer identification through association with allocation sites, allocation site memory maps, and detailed information from dynamic checks were essential to timely application development. Maintaining allocation site information separately from the objects allowed queries to the debugging system concerning pointers to corrupted objects to yield allocation site information for the object originally allocated at that location. Use of no-heap real-time threads with the Sun JDK 1.3 is not recommended. A real-time library usable from no-heap real-time threads would speed application development.

A library with real-time guarantees which uses performance-bounded memory allocation (linear-time allocation memory areas) from Real-Time Java has to specify an initial size for the performance-bounded region of a linear-time allocation memory area. Since this initial size is platform-specific, real-time libraries which use performance-bounded memory allocation may not be portable. The development of a large portable library with real-time performance guarantees to facilitate real-time application development may necessitate a solution to the memory area initial size problem.

Overall, the Real-Time Specification for Java [3] offers a good starting point as a set of language extensions to support initial real-time application development in Java. Solutions to all problems encountered during real-time application development in Java may require more research and specification development. Based on our experience, however, we feel that the specification and/or the standard Java execution environments may undergo significant additional development before they are suitable for widespread use.

Chapter 8

Future Work

The Real-Time Specification for Java [3] and our implementation provides a base for many future research projects. Unfortunately, our implementation currently includes support for only the memory management aspects of the Real-Time Specification for Java. [3] We have yet to explore aspects related to the real-time thread scheduler. Future work may include projects related to analysis, application development, and enhancements to the implementation.

Analysis projects include:

- escape analysis to eliminate heap reference checks,
- use of memory area type information to eliminate more checks,
- static cycle detection to ensure the safety of using reference count memory areas in applications,
- statically determining the sizes of memory areas in terms of variables available to the constructor of the memory area using implementation specific information,
- proving that some programs never run out of memory,
- and static analysis to infer regions and automatically convert Java programs into Real-Time Java programs using scoped memories for speed improvements.

Application development projects include:

- a general-purpose real-time library usable from no-heap real-time threads,
- specific real-time libraries to interface with real-world hardware, such as a video library to process images from a camera, or a library to control commands given to a robot,
- develop a standard set of benchmarks that can test the performance and compliance of different Real-Time Java implementations,
- and, after developing many applications using Real-Time Java, develop a new set of requirements for the Real-Time Java specification and ideas to enhance the application development environment and debugging system.

Implementation enhancements include:

- alternatives to the reference count memory areas,
- better memory allocation strategies for linear-time and variable-time memory areas which may deallocate objects early,
- finish implementing the entire Real-Time Specification for Java [3],
- and work on the integration of many different types of garbage collectors with no-heap real-time threads.

Bibliography

[1] A. Aiken and D. Gay. Memory management with explicit regions. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, SIGPLAN, pages 313–323, Montreal, June 1998. ACM.

[2] A. Aiken and D. Gay. Language support for regions. In PLDI [8].

[3] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, Reading, Massachusetts, June 2000.

[4] D. Cannarozzi, M. Plezbert, and R. Cytron. Contaminated garbage collection. In PLDI [7].

[5] M. Christiansen and P. Velschrow. Region-based memory management in Java. Master's thesis, DIKU, May 1998.

[6] T. Domani, E. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. In PLDI [7].

[7] *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, SIGPLAN, Vancouver, June 2000. ACM.

[8] *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, SIGPLAN, Snowbird, Utah, June 2001. ACM.

[9] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In PLDI [8].

[10] M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[11] D. Walker and K. Watkins. On regions and linear types. To appear in the ACM SIGPLAN International Conference on Functional Programming, September 2001.