An Implementation of Scoped Memory for Real-Time Java

William S. Beebee, Jr. and Martin Rinard

MIT Laboratory for Computer Science
Massachusetts Institute of Technology, Cambridge MA, 02139
wbeebee@alum.mit.edu, rinard@lcs.mit.edu

Abstract. This paper presents our experience implementing the memory management extensions in the Real-Time Specification for Java. These extensions are designed to given real-time programmers the control they need to obtain predictable memory system behavior while preserving Java's safe memory model. We describe our implementation of certain dynamic checks required by the Real-Time Java extensions. In particular, we describe how to perform these checks in a way that avoids harmful interactions between the garbage collector and the memory management system. We also found that extensive debugging support was necessary during the development of Real-Time Java programs. We therefore used a static analysis and a dynamic debugging package during the development of our benchmark applications.

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 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 Java specification uses dynamic access 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 Java specification uses dynamic heap 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 Java specification 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.

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 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.

2 Programming Model

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

2.1 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.

2.2 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 access 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 access 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.

2.3 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 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 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.

3 Example

We next present an example that illustrates some of the features of the Real-Time Specification for Java. Figure 1 presents a sample program written in Real-Time Java. This program is a version of the familiar "Hello World" program augmented to use the Real-Time Java features. It first creates a scoped memory with a worst-case Linear Time allocation scheme (`LTMemory`) with a size of 1000 bytes. It then runs the code of the `run` method in this new scope. The `run` method creates a new variable time allocation scoped memory (the `VTMemory` object) and a new `Worker NoHeapRealtimeThread`. Both of these objects are allocated in the `LTMemory` scoped memory. The `run` method then starts the `Worker` thread and executes its `join` method, which will return when the `Worker` finishes.

The `Worker` thread runs in the new `VTMemory`. The `Worker`'s `run` method allocates a new `String`[1] in `ImmortalMemory` and stores a reference to this string in the static `results` field of the `Main` class, which was previously initialized to `null`. The `Worker` then creates a new `String`, "Hello World!", to place in the array. The worker then finishes, and the implementation deallocates all of the objects allocated in the `VTMemory`. Back in the main thread, the `join` method returns, and the main thread returns back out of its `run` method. The implementation deallocates all of the objects allocated in the `LTMemory`. Finally, the main thread prints "Hello World", the first element of the `results` array, to the screen.

Note that the `LTMemory` and `VTMemory` constructors differ slightly from the constructors described in the Realtime Java specification. We implemented these constructors in addition to the specified constructors to provide additional flexibility and convenience for the programmer.

This Hello World program is a legal program using our system. However, any of the following changes would make it an illegal program:

```
class Worker extends NoHeapRealtimeThread {
  Worker(MemoryArea ma) { super(ma); }
  public void run() {
    ImmortalMemory im = ImmortalMemory.instance();
    try {
      Main.results = (String[])im.newArray(String.class, new int[] { 1 });
      Main.results[0] = (String)im.newInstance(String.class,
                                               new Class[] { String.class },
                                               new Object[] { ‘‘Hello World!’’ });
    } catch (Exception e) { System.exit(-1); }
  }
}
public class Main {
  public static String[] results = null;
  public static void main(String args[]) {
    LTMemory lt = new LTMemory(1000);
    lt.enter(new Runnable() {
      public void run() {
        Worker w = new Worker(new VTMemory());
        w.start();
        try { w.join(); } catch (Exception e) { System.out.println(e); }
      }
    });
    System.out.println(results[0]);
  }
}
```

Fig. 1. A Real-Time Java Example Program

1. Replace the `im.newInstance...` with `‘‘Hello World!’’` and there would be an illegal reference from an `ImmortalMemory` to a `ScopedMemory`.
2. Replace the `im.newArray...` with `new String[1]` and there would be an illegal static reference to a `ScopedMemory`.
3. Replace the `ImmortalMemory.instance()` with `HeapMemory.instance()`, and there would be an illegal heap reference in a `NoHeapRealtimeThread` (**READ**).
4. Replace the `null` with a `new String[1]` and the `NoHeapRealtimeThread` would be illegally destroying a heap reference by assigning `Main.results` (**WRITE**).
5. Place the `Worker w` in the `main` method and the assignment `w = new Worker...` would illegally create a reference from the heap to a `ScopedMemory`.
6. Place the `System.out` in the `NoHeapRealtimeThread` and the `NoHeapRealtimeThread` would be illegally reading from the heap. `System.out` is initialized in the initial `MemoryArea` at the start of the program, the

HeapMemory (**READ**) As a consequence, the NoHeapRealtimeThread cannot System.out.println the message from the exception.

7. Place the entire Worker w = new Worker(new VTMemory()); outside the LTMemory scope, and the this pointer of the NoHeapRealtimeThread would illegally point to the heap (**METHOD**).

4 Implementation

Our discussion of the implementation focuses on three aspects: implementing the heap and access checks, implementing the additional scoped immortal memory functionality, and ensuring the absence of interactions between no-heap real-time threads and the garbage collector.

4.1 Heap 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 2 presents the code that the compiler emits for each heap check; Figure 3 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).

READ	**WRITE**	**CALL**

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

NATIVECALL	**METHOD**

`refExp = nativecall(args);`
 	
becomes:
 	
`heapRef = nativecall(args);`
`if (heapRef&1)`
`  heapCheck(heapRef);`
`refExp = heapRef;`
	`method(args) { body }`
	
	becomes:
	
	`method(args) {`
	`  for arg in args:`
	`    if (arg&1)`
	`      heapCheck(arg);`
	`body }`

Fig. 2. Emitted Code For Heap Checks

4.2 Access Check Implementation

The access 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 access 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 presents the emitted code for the access checks, while Figure 5 presents some of the run-time code that this emitted code invokes.

```
#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 */
    }
  }
```

Fig. 3. 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;
```

Access check:

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

becomes:

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

Fig. 4. Emitted Code for Access Checks

In `MemoryArea`:

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

In `RealtimeThread`:

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

Fig. 5. Code for performing access checks

4.3 Operations on Memory Areas

The implementation needs to perform three basic operations on scoped and immortal memory areas: allocate an object in the area, deallocate all objects in the area, and provide the garbage collector with the set of all heap references stored in the memory area. 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.

Each memory area may have its own object allocation algorithm. Because the same code may execute in different memory areas at different times, our implementation is set up to dynamically determine the allocation algorithm to use based on the current memory area. Whenever a thread allocates an object, it looks up a data structure associated with the memory area. A field in this structure contains a pointer to the allocation function to invoke. This structure also contains a pointer to a function that retrieves all of the heap references from the area, and a function that deallocates all of the objects allocated in the area.

4.4 Memory Area Reference Counts

As described in the Real-Time Java Specification, each memory area maintains a count of the number of threads currently operating within that region. These counts are (atomically) updated when threads enter or exit the region. When the count becomes zero, the implementation deallocates all objects in the area.

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.

4.5 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.

There is a complication associated with this implementation. Note that multiple threads can attempt to concurrently allocate memory from the same stack allocator. The implementation must therefore use some mechanism to ensure that the allocations take place atomically. Note that the use of lock synchronization could cause an unfortunate coupling between real-time threads, no-heap real-time threads, and the garbage collector. 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. 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.

Our current implementation avoids this problem by using 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.

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, `VTMemory`s use the linked-list allocator, while `LTMemory`s use the stack-allocator.

4.6 Garbage Collector Interactions

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. This scan is coded to operate correctly in the presence of concurrent updates from no-heap real-time threads. The garbage collector uses the extracted heap references as part of its root set.

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 checks, this property ensures that the garbage collector and no-heap real-time threads do not inappropriately interfere.

5 Debugging 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 and access 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 and access 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 [5]. 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 and access 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 in which the object is allocated. Given any arbitrary object reference, a debugging function can retrieve the debugging information for the object. 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 be instrumental in quickly determining the exact cause of the error and the objects responsible. Allocation site information can also be displayed 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.

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.

We obtained several benchmark programs and used these programs to measure the overhead of the heap checks and access 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 and access check elimination.

Table 1 presents the number of objects we were able to allocate in each of the different kinds of memory areas. The goal is to allocate as many objects as possible in scoped memory areas; the results show that we were able to modify

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

Table 1. Number of Objects Allocated In Different Memory Areas

Benchmark	Heap	Scoped	Immortal	Total
Array	13	4	0	17
Tree	13	65,534	0	65,547
Water	406,895	3,345,711	0	3,752,606
Barnes	16,058	4,681,708	0	4,697,766

Table 2. Number of Arrays Allocated In Different Memory Areas

Benchmark	Heap	Scoped	Immortal	Total
Array	36	4	0	40
Tree	36	0	0	36
Water	405,943	13,160,641	0	13,566,584
Barnes	14,871	4,530,765	0	4,545,636

the programs to allocate the vast majority of their objects in scoped memories. Java programs also allocate arrays; Table 2 presents the number of arrays that we were able to allocate in scoped memories. As for objects, we were able to allocate the vast majority of arrays in scoped memories.

Table 3 presents the number and type of access checks for 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, the Scoped to Heap column counts the number of times the program stored a reference to heap memory into an object or array allocated in a scoped memory.

Table 3. Access Check Counts

Benchmark	Heap to Heap	Heap to Immortal	Scoped to Heap	Scoped to Scoped	Scoped to Immortal	Immortal to Heap	Immortal to Immortal
Array	14	8	0	400,040,000	0	0	0
Tree	14	8	0	65,597,532	65,601,536	0	0
Water	409,907	0	17,836	9,890,211	844	3	1
Barnes	90,856	80,448	9,742	4,596,716	1328	0	0

Table 4 presents the running times of the benchmarks. We report results for six different versions of the program. The first three versions all have both heap

Table 4. Execution Times of Benchmark Programs

	With Checks			Without Checks		
Benchmark	Heap	VT	LT	Heap	VT	LT
Array	28.1	43.2	43.1	7.8	7.7	8.0
Tree	13.2	16.6	16.6	6.9	6.9	6.9
Water	58.2	47.4	37.8	52.3	40.2	30.2
Barnes	38.3	22.3	17.2	34.7	19.5	14.4

and access 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 all of the checks. For our benchmarks, our static analysis is able to verify that none of the checks will fail, enabling the compiler to eliminate all of these checks [5].

These results show that checks add significant overhead for all benchmarks. But the use of scoped memories produces significant performance gains for Barnes and Water. In the end, the use of scoped memories without checks significantly increases the overall performance of the program. 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 garbage collector: it reduced the total garbage collection overhead, increased the time between collections, and significantly reduced the pause times for each collection.

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 access checks are faster than scope to scope access checks.

7 Related Work

Christiansen and Velschow suggested a region-based approach to memory management in Java; they called their system RegJava[4]. 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 [6]. 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.

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.

8 Conclusion

The Real-Time Java Specification 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 the memory management. 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 with respect to the design. We also 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.

Acknowledgements

This research was supported in part by DARPA/AFRL Contract F33615-00-C-1692 as part of the Program Composition for Embedded Systems program. The authors 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 access violations. Brian Demsky implemented a user-threads package for the Flex compiler which improved the performance of some benchmarks.

References

1. Aiken, A., Gay, D.: Memory Management with Explicit Regions. Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation. Montreal, Canada, June 1998.
2. Aiken, A., Gay, D.: Language Support for Regions. Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation. Snowbird, Utah, June 2001.
3. Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M.: The Real-Time Specification for Java. Addison-Wesley, Reading, Massachusetts, 2000.
4. Christiansen, M., Velschow, P.: Region-Based Memory Management in Java. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, May 1998.
5. Salcianu, A., Rinard, M.: Pointer and Escape Analysis for Multithreaded Programs. Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Snowbird, Utah, June 2001.
6. Tofte, M., Talpin, J.: Region-Based Memory Management. Information and Computation, 132(2):109–176 (1997)