

Asymptotic Analysis for Real-time Java Scoped-memory Areas*

Delvin C. Defoe Rob LeGrand Ron K. Cytron
{dcd2, legrand, cytron}@cse.wustl.edu
Department of Computer Science and Engineering
Washington University
St. Louis, Missouri

Abstract

Java has recently joined C and C++ as a development platform for real-time and embedded applications. Java's garbage collection, while generally a useful feature, can be problematic for these applications: garbage collection occurs at unpredictable times and its latency is typically unbounded. This can compromise necessary real-time guarantees.

To overcome these limitations, the Real-Time for Java Expert Group (RTJEG) proposed the Real-Time Specification for Java (RTSJ), which introduced new memory models and new threads to utilize those models. One such memory model uses scoped-memory areas, which work best in the context of a *NoHeapRealtimeThread* (NHRT). Although much work has been done with scoped-memory areas and NHRTs, there is no system-independent analysis of their costs. In this article we present an asymptotic analysis for RTSJ scoped-memory areas and NHRTs.

Keywords: Scoped Memory, Real-Time Java, Memory Management, Programming Languages, and Performance Analysis

1 INTRODUCTION

Real-time applications require bounded time memory-management performance. Many real-time applications are written in C and assembly language. These languages do not offer garbage-collection capabilities by default; as such, real-time applications written in them do not suffer from the overhead associated with garbage collection. Since the advent of Java in 1995 [?, ?], programmers have been exploring ways to use Java for real-time programming.

The Real-Time for Java Expert Group (RTJEG) has advanced the use of Java as a real-time programming language by issuing the Real-Time Specification for Java [?] (RTSJ). The RTSJ provides extensions to Java in support of real-time programming [?] that do not change the basic structure of the language. Instead, they added new class libraries and Java Virtual Machine (JVM) extensions to the language. Compilers that are not optimized to take advantage of those extensions are not affected by their addition. Although the RTSJ impacts many areas of the Java programming language, this article focuses on storage management.

In addition to the heap where dynamic storage allocation occurs, the RTSJ specifies other memory areas for dynamic storage allocation. Those memory spaces are called *immortal memory* and *scoped-memory* areas [?]. Objects allocated in those memory areas are never subjected to garbage collection although the garbage collector may scan immortal memory. Objects in immortal memory survive the execution of the program, whether or not there are references to them. Objects in a scoped-memory area, on the other hand, are collected *en masse* when every thread that has entered the scope has exited the scope.

Use of scoped-memory areas is not without additional cost or burden. We investigate the issue of providing asymptotic bounds for scoped memory use by exploring a selected class of abstract data types (ADTs). Asymptotic bounds will give an idea of how expensive it is to employ scoped-memory areas in real-time and embedded environments.

The rest of the paper is organized as follows. Section 2 discusses RTSJ's influence on memory management. Sections 3, 4 and 5 provide analysis of scoped-memory areas. We conclude in section 6.

*This work is sponsored in part by DARPA under contract F33615-00-C-1697, by AFRL under contract PC Z40779, and by Raytheon under subcontract 4400134526.

Objects in	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Allowed	Allowed	Not allowed
Immortal	Allowed	Allowed	Not allowed
Scoped	Allowed	Allowed	Allowed if same, outer, or shared scope

Table 1: References between storage areas [?] courtesy of [?]. Objects in the heap are allowed to reference objects in immortal memory, but not objects in scoped memory.

2 RTSJ’S MEMORY MANAGEMENT

One of the most interesting features of the RTSJ is the new memory management model based on *scoped-memory areas* (or scopes for short) [?]. This model ensures programmers of timely reclamation of memory and predictable performance. This comes at the cost of learning an unfamiliar programming model—a restrictive model that relies on the use of scopes. These new memory areas were designed to meet two very important requirements [?]: providing predictable allocation and deallocation performance, and ensuring that real-time threads do not block when memory is reclaimed by the virtual machine.

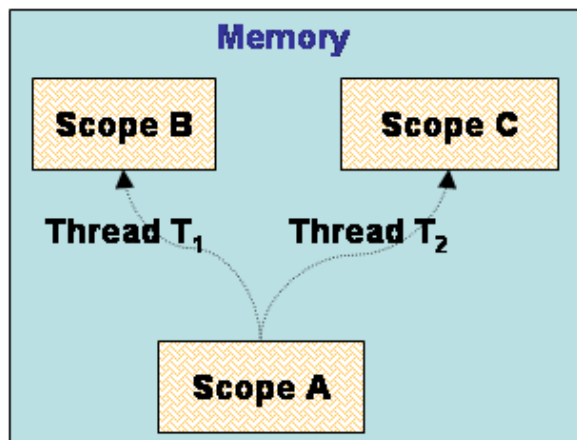


Figure 1: Scoped-memory single-parent rule. *A* is the parent of both *B* and *C*

To meet these requirements, the RTSJ ensures that objects in a scope are not deallocated individually. Instead, the entire scope is collected *en masse* when all thread that have entered the scope have exited the scope. A scope is a pool of memory from which objects are allocated. Each scope can be entered by multiple threads. These threads can allocate objects in the memory pool and communicate with each other by shared variables.

The RTSJ allows the nesting of scopes, but the nesting

is controlled by the order of threads entering the scopes—see Figure 1. A scope can become the parent of multiple scopes but no scope is allowed to have multiple parents.

To take advantage of scopes, the RTSJ defined a new type of thread called *NoHeapRealtimeThread* (NHRT). NHRTs cannot allocate objects in the garbage collected heap and they cannot reference objects in the heap. These constraints were added to prevent NHRTs from experiencing unbounded delay due to the locking of heap objects during garbage collection [?]. NHRTs have the highest priority among all threads and can preempt even the garbage collector.

Table 1 details how objects in certain memory areas are not allowed to reference objects in other memory areas. This constraint does not apply to objects only, but also to threads so that real-time threads do not block when the JVM reclaims objects.

3 SCOPED MEMORY ANALYSIS

This section focuses on computing asymptotic bounds for the RTSJ scoped memory model when NHRTs are used. These bounds will give an idea of how expensive it is to execute applications in a scoped memory environment. They will also facilitate comparison of execution in scoped memory environments with execution in other memory environments (*e.g.*, the heap).

Asymptotic analysis of scoped memory

To the best of our knowledge, there is no record in the literature of system-independent cost analysis for scoped memory regions with NHRTs. The goal of this article is to provide a framework for comparing programming with RTSJ scoped memory to programming with other memory models, *e.g.*, the heap. While there are several approaches that can be used to do this comparison, we propose a model that use asymptotic analysis. The steps of this model are listed below.

1. Select an abstract data type (ADT) that can hold an arbitrary number of elements.

2. Define the fundamental operations of the ADT and provide an interface.
3. Propose at least one implementation for each operation.
4. Adopt methods from Cormen *et al.* [?] to compute the worst-case running time for each operation.
5. Perform step 4 for an intermixed sequence of n operations. This denotes the ADT problem (*e.g.*, the stack problem).
6. Repeat step 1.

The input size n for each operation is characterized by the number of elements in the collection immediately before the operation is run. Should there be a need to use a different characterization for the input size of an operation, one will be provided. The pseudocode provided for selected operations follow the pseudocode conventions from Cormen *et al.* [?]. We apply this model to solve the problem at hand.

4 STACK ANALYSIS

The stack is the first abstract data type we analyze. A *stack* is an ADT that operates on the *Last In First Out* (LIFO) principle. One end of a stack called the top-of-stack is used for each stack operation. The fundamental operations are:

1. IS-EMPTY(S) - an operation that returns the binary value **TRUE** if stack S is empty, **FALSE** otherwise.
2. PUSH(S, x) - an operation that puts element x at the *top* of stack S .
3. POP(S) - an operation that removes the element at the top of the stack S and returns it. If the stack is empty the special pointer value *NULL* is returned. *NULL* is used to signify that a pointer has no target.

Heap implementation

Several data structures, including the singly linked list, can be used to implement a stack in the heap. The IS-EMPTY operation checks whether the top-of-stack points to *NULL*. The PUSH operation adds a new element to the top-of-stack, and the POP operation updates the top-of-stack and returns the topmost element. Each of these fundamental operations requires $T(n) = O(1)$ time.

Scoped memory implementation

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

1. Each application A that manages a stack S is fully compliant with the RTSJ.
2. A has a single thread T_a , which is an instance of RTSJ NHRT.
3. A executes on an RTSJ compliant JVM.
4. T_a can legally access S and the elements managed by S .
5. Before an element, x , is pushed on stack S , a new scope s must first be instantiated to store x , and T_a must enter s .

Assumption 5 is relevant for the purpose of complexity analysis. Although we do not suggest a scope/element for an actual implementation, here we are concerned about worst-case analysis. Storing a single element in a scope simplifies analysis and yields the smallest amount of unnecessarily live storage in scoped memories for this particular data structure. Pseudocode and analysis for the fundamental stack operations follow.

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

```
IS-EMPTY( $S$ )
1 return  $TOS = S$ 
```

Figure 2: Procedure to test if the stack is empty—scoped memory implementation

PUSH: The PUSH operation depicted in Figure 3 is equivalent to the following sequence of basic operations performed by the application thread T_a . From the current scope T_a instantiates a new scope sm . T_a enters sm then sets the *TOS* field in sm to point to element x .

Assuming each line i in PUSH requires c_i time for execution, the worst case execution time for PUSH is given by $T(n) = c_1 + c_2 + c_3 = O(1)$. The correctness of this result is based on the fact that each line is executed once per invocation. Because a scope has a limited lifetime dictated by the reference count of threads executing therein,

```

PUSH( $S, x$ )
1  $scope \leftarrow new\ ScopedMemory(m)$ 
2  $enter(scope, T_a)$ 
3  $TOS \leftarrow x$ 

```

Figure 3: Procedure to push an element onto the stack—scoped memory implementation. $m \geq |x| + |TOS|$.

T_a is not allowed to exit sm . To ensure that T_a keeps sm alive T_a does not return from the $enter()$ method of line 2, Figure 3. Should T_a return from the $enter()$ method, the thread reference-count of sm would drop to zero, sm would be collected, and the PUSH operation would fail.

POP: The POP operation returns the TOS element if one exists, $NULL$ otherwise. Assuming each line i of the POP operation (Figure 4) requires c_i time to execute, the worst-case execution time for the POP operation is given as $T(n) = O(1)$.

```

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

```

Figure 4: Procedure to pop the topmost element off the stack—scoped memory implementation

After popping the stack, T_a must return from the $enter()$ method of line 2, Figure 3. We assume for all practical purposes that returning from the $enter()$ method takes $O(1)$ time. The new top-of-stack becomes the TOS element of the parent scope *i.e.*, the parent of the scope that contained the popped element.

The stack problem

The stack problem is defined as the problem of running an intermixed sequence of n PUSH and POP operations on a stack instance. We analyze the stack problem for a singly linked list implementation in the heap and for a scoped memory implementation of stack. Let n denote the total number of operations and let m denote the number of PUSH operations. The number of POP operations is thus given by $n - m$ where $n - m \leq m \leq n$. The worst-case running time for the singly linked list implementation of the stack problem is computed as

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

For a scoped memory implementation the running time for PUSH or POP is $O(1)$. Thus, the running time for the stack problem in the context of a scoped memory implementation is given by $T(n) = O(n)$.

Discussion

The linked list implementation in the heap yields $T(n) = O(1)$ worst-case execution time for each stack operation. The scoped memory implementation also yields $T(n) = O(1)$ worst-case execution time for each operation. The stack problem, *i.e.*, the problem of running an intermixed sequence of n PUSH and POP operations yields a worst-case running time of $T(n) = O(n)$ for each implementation, as expected. Given a particular program that uses a stack, the programmer can thus choose among these two implementations.

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

A scoped memory implementation, while good in real-time environments, comes at the cost of learning a new, restrictive programming model. Real-time programmers, however, can benefit from the timing guarantees promised by the RTSJ.

5 QUEUE ANALYSIS

A *queue* is an ADT that operates on the *First-In-First-Out* (FIFO) principle. The fundamental operations for a queue are:

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

Heap implementation

One possible implementation of the queue ADT in the heap uses a singly linked list data structure with two special pointers, *front* and *rear*. The ISQ-EMPTY operation checks whether *front* points to *NULL*. The ENQUEUE operation adds a new element to the rear end of the linked list and updates the *rear* pointer. The DEQUEUE operation updates the *front* pointer and returns the element that was at the front of the linked list. Each of these fundamental operations takes time $T(n) = O(1)$.

Scoped memory implementation

Consider execution of an application A that manages a queue instance in an RTSJ scoped memory environment. Efficient execution of A depends on proper management of memory, which is a limited resource. Assume A uses a stack of scoped memory instances to manage the queue. Assume also, for the purpose of worst-case analysis, that a queue element resides in its own scope when added to the queue. A service stack with its own NHRT T_1 is used to facilitate the ENQUEUE operation. See Figure 5 for a representation of a queue instance. If T_0 is the application thread, then T_0 is a NHRT. Detailed analysis of the fundamental queue operations follows.

ISQ-EMPTY: The current scope contains a *front* field that points to the front of the queue. An empty queue is a queue with no elements. Emptiness, in Figure 6, is illustrated by the *front* field of the current scope pointing to the queue object itself. Assuming that the running time of the lone line of ISQ-EMPTY is c_1 , the worst-case running time of ISQ-EMPTY is given as

$$T(n) = c_1 = O(1).$$

DEQUEUE: The DEQUEUE operation removes the element at the front of the queue and returns it if one exists. Otherwise, it returns *NULL*. A close examination of the DEQUEUE operation (Figure 7) reveals that it is

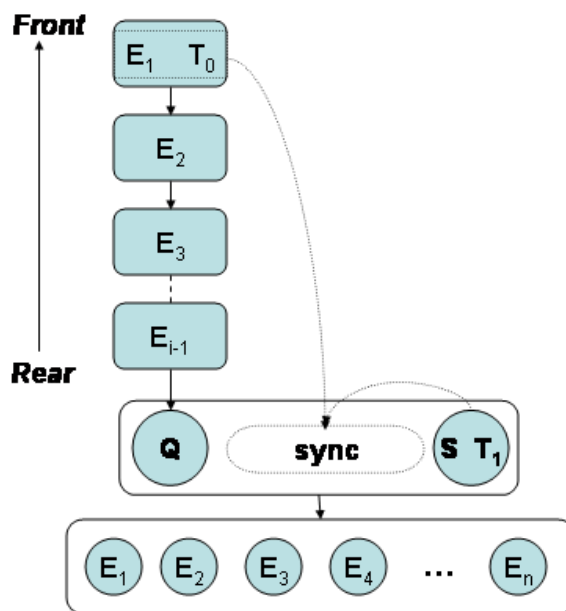


Figure 5: Representation of a queue instance in an RTSJ scoped memory environment. Rounded rectangles represent scoped memory instances and circles represent object instances. T_0 is the application thread and T_1 services the stack. The arrows pointing downward represent legal scope references. The *sync* field/object is a synchronization point for the T_0 and T_1 . E_i denotes element i . On the queue, E_i is a reference to element i .

```
ISQ-EMPTY( $Q$ )
1 return  $front = Q$ 
```

Figure 6: Procedure to test if the queue is empty—scoped memory implementation

similar to the POP operation of Figure 4. Hence, the worst-case running time for DEQUEUE is $T(n) = O(1)$ time.

ENQUEUE: The ENQUEUE operation is a relatively complex operation because of the referencing constraints imposed by RTSJ: objects in an ancestor scope cannot reference objects in a descendant scope because the descendant scope is reclaimed before the ancestor scope. As a consequence of these constraints, the elements already in a queue must first be stored somewhere before a new element can be enqueued. After the element is enqueued, all the stored elements are put back on the queue in their correct order. A stack is an ideal structure

```

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

```

Figure 7: Procedure to remove an element from the front of the queue—scoped memory implementation

to store the queue elements because it preserves their order for the queue. As illustrated in Figure 5, two threads are needed to facilitate the ENQUEUE operation: one for the queue and one to service the stack. The thread that services the queue is the application thread and is referred to as T_0 ; T_1 is the service thread for the stack. These two threads are synchronized by a parameter *sync*, which they use to share data between them. *sync* is referred to as y in Figure 8.

ENQUEUE(Q, x)	<i>time cost</i>	<i>frequency</i>
1 while !ISQ-EMPTY(Q)	c_1	$n + 1$
2 do $y \leftarrow$ DEQUEUE(Q)	c_2	n
3 PUSH(S, y)	c_3	n
4 $S_c \leftarrow$ new <i>ScopedMemory</i> (m)	c_4	1
5 <i>enter</i> (S_c, T_0)	c_5	1
6 $front \leftarrow x$	c_6	1
7 while !IS-EMPTY(S)	c_7	$n + 1$
8 do $y \leftarrow$ POP(S)	c_8	n
9 PUSH-Q(Q, y)	c_9	n

Figure 8: Procedure to add an element to the rear of the queue—scoped memory implementation. Each c_i is a constant and $n = |Q| + |S|$. Initially stack S is empty.

```

PUSH-Q(S, x)
1   $scope \leftarrow$  new ScopedMemory( $m$ )
2  enter( $scope, T_a$ )
3   $front \leftarrow x$ 

```

Figure 9: Private helper method that puts an element at the front of the queue in the same manner that an element is pushed onto a stack—scoped memory implementation. $m \geq |x| + |front|$.

PUSH-Q is a private method that puts a stored element back on the queue in the way that the PUSH operation works for a stack. The worst-case running time for this method is $T(n) = O(1)$ time. This is the same running time for the PUSH operation of Figure 3.

Given the procedure in Figure 8 the worst-case running time for ENQUEUE is linear in the number of elements already in the queue:

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

The queue problem

The queue problem is defined as the problem of running an intermixed sequence of n ENQUEUE and DEQUEUE operations on a queue instance. We compute the theoretical running time for the queue problem by analyzing the worst-case running time of the sequence. Since we suggested two implementation contexts for the queue ADT, we compute the running time for each implementation. Suppose n denotes the number of operations in the sequence, m denotes the number of ENQUEUE operations, and s denotes the number of elements in the queue. The number of DEQUEUE operations is thus given as $n - m$ where $n - m \leq s \leq m \leq n$. This suggests that there are never fewer ENQUEUEs than DEQUEUEs in a subsequence of the first $i \leq n$ operations. The worst-case running time for the heap implementation follows.

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

This is identical to the linked-list analysis of the stack problem because the insertion operation and the deletion operation each executes in constant time. The scoped memory implementation is more complex for the ENQUEUE operation. The running time for the queue problem in that context is also more complex and more costly. We compute the worst-case running time as follows:

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

$s \in [s_1, s_2, \dots, s_m]$ is included as input to the computation of the running time for the ENQUEUE operations because the running time of each invocation of the ENQUEUE operation depends on the number of elements in the queue. The worst case running time for the sequence of operations occurs when the m ENQUEUE operations are executed one after another, with no intervening DEQUEUE operations. In this case the value of s is mono-

tonically increasing from 0 to $m - 1$; so for the computation of $T(n)$ given below, $s_i = i - 1$.

$$\begin{aligned}
T(n) &= T_{enq}(m, s) + T_{deq}(n - m) \\
&= \sum_{i=1}^m (a + s_i b) + T_{deq}(n - m) \\
&= \sum_{i=0}^{m-1} (a + ib) + T_{deq}(n - m) \\
&= (m - 1)a + \sum_{i=0}^{m-1} (ib) + T_{deq}(n - m) \\
&= (m - 1)a + b \sum_{i=0}^{m-1} i + T_{deq}(n - m) \\
&= (m - 1)a + \frac{m(m - 1)b}{2} + T_{deq}(n - m) \\
&= (m - 1)a + \frac{m^2 b}{2} - \frac{mb}{2} + T_{deq}(n - m) \\
&= \frac{m^2 b}{2} - \frac{m(b - 2a)}{2} - a + T_{deq}(n - m) \\
&= \frac{m^2 b}{2} - \frac{m(b - 2a)}{2} - a + (n - m)c_2 \\
&= \frac{m^2 b}{2} - \frac{m(b + 2c - 2a)}{2} - a + nc_2
\end{aligned}$$

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

Discussion

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

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

Improved scoped memory implementation

We presented thus far an implementation of a queue in an RTSJ scoped memory environment that turned out to have a worst-case running time of $O(n^2)$ for n consecutive ENQUEUE operations. Here, we present a modified queue implementation that has better worst-case time performance on the queue problem (see Figure 10).

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

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

```

ENQUEUE(Q, x)
1  nenq ← nenq + 1
2  if nenq is some power of 2
3    then while !ISQ-EMPTY(Q)
4      do y ← DEQUEUE(Q)
5         PUSH(S, y)
6      for i = 1 to nenq
7        do Sc ← new ScopedMemory(m)
8           enter(Sc, T0)
9           front ← getOuterScope()
10     thread[next] ← front
11     front ← x
12     while !IS-EMPTY(S)
13       do y ← POP(S)
14          PUSH-Q(Q, y)
15     else temp ← thread[next][front]
16        thread[next][front] ← x
17        thread[next] ← temp

```

Figure 10: Procedure to add an element to the rear of the queue—scoped memory implementation.

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

Figure 11: Statistics for procedure in Figure fig:enq-queue-scoped-imp. Each c_i is a constant and $n = |Q| + |S|$. Initially stack S is empty and so $n = |Q|$.

The queue problem revisited: The worst-case running time for a single call of the ENQUEUE operation is $O(n)$, where n is the number of elements already on the queue, so the worst-case running time for n consecutive ENQUEUE calls might reasonably be expected to be $O(n^2)$. Fortunately, however, that turns out not to be the case. Consider beginning with an empty queue and performing a series of n ENQUEUE operations with no DEQUEUEs. During the i th ENQUEUE call, $n = i - 1$ (since n is the number of elements already on the queue) and $n_{\text{enq}} = i$ (after the shared-memory field n_{enq} is incremented as the first step of the ENQUEUE algorithm). Then it can be seen from Figure 10 that the i th ENQUEUE call takes $c_a + c_b i$ time if $i = 2^x$ for some integer x , where

$$c_a = c_1 + c_2 - c_4 - c_5 + c_6 + c_{10} + c_{11} - c_{13} - c_{14}$$

$$c_b = c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{12} + c_{13} + c_{14}$$

and c_c time otherwise, where

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

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

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

Therefore the improved ENQUEUE operation has a worst-case running time of $O(n)$ on the queue problem. However, because it overallocates when resizing, it relies at some point on having twice the number of cells allocated as are actually in use. Interestingly, a time/space tradeoff of this nature is endemic to the real-time collectors [?] as well.

6 CONCLUSIONS

There are many implementations of RTSJ including TymeSis, JRate, and Sun Microsystems' implementation.

Many in the real-time and academic communities are experimenting with RTSJ and scoped-memory areas. However, until now, there has not been an objective analysis of scoped-memory areas.

To fill this void we presented a model to perform asymptotic analysis for RTSJ scoped-memory areas. Using the model, we determined asymptotic bounds for RTSJ scoped-memory areas in the context of stacks and queues. Our analysis permits us to compare scoped memory with other memory models and to reason more thoroughly about different memory models.

Although in practice we do not suggest one element per scope, the analysis does not change if multiple elements are allowed per scope. Consider, for example $4k$ elements per scope. If $4k$ elements are enqueued on a queue and a $4k + 1^{st}$ element is to be enqueued (with no intervening dequeues), a new scope would have to be instantiated to accommodate that element. That enqueue operation suffers the cost discussed in Section 5.

In the near future, we plan to explore RTSJ scoped memory use with other abstract data types to gain a more complete understanding of how they impact the cost of scoped memory use. However, even in the case of a simple queue, using RTSJ scoped memories instead of a collector does not avoid the time/space tradeoffs experienced with the real-time collectors. Ours is the first work to point this out.

7 ACKNOWLEDGEMENTS

We thank the Chancellor's Graduate Fellowship Program at Washington University for its support of Delvin Defoe. We thank Morgan Deters for his insight, feedback, and support.