Efficient Memory-Reference Checks for Real-time Java*

Angelo Corsaro and Ron K. Cytron
Department of Computer Science and Engineering
Washington University
St. Louis, MO, 63130, USA
{corsaro, cytron}@cse.wustl.edu

ABSTRACT

The scoped-memory feature is central to the Real-Time Specification for Java. It allows greater control over memory management, in particular the deallocation of objects without the use of a garbage collector. To preserve the safety of storage references associated with Java^{TM} since its inception, the use of scoped memory is constrained by a set of rules in the specification. While a program's adherence to the rules can be partially checked at compile-time, undecidability issues imply that some—perhaps, many—checks may be required at run-time. Poor implementations of those run-time checks could adversely affect overall performance and predictability, the latter being a founding principle of the specification.

In this paper we present efficient algorithms for managing scoped memories and the checks they impose on programs. Implementations and results published to date require time linear in the depth of scope nesting; our algorithms operate in constant time. We describe our approach and present experiments quantifying the gains in efficiency.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Construct and Features—*dynamic storage management, constraints*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Performance, Design

Keywords

Real-Time Java, Object Oriented Languages, Scoped Memory, Garbage Collection, Memory Management

*Sponsored by DARPA under contract F33615-00-C-1697

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LCTES'03, June 11–13, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-647-1/03/0006 ...$5.00.

1. INTRODUCTION

One of the main hurdles for making Java^{TM} [1] usable for the development of real-time systems has been the fact it is a garbage-collected language. While developers generally like the functionality provided by garbage collection, they object to those aspects of garbage collection that could introduce unpredictable delay into an application.

At one extreme, some collectors work in a *stop the world and collect* fashion, in which the garbage collector has to stop all the running threads before attempting to reclaim dead storage. Other types of garbage collectors take a less restrictive approach, but most of the precise garbage collectors known in literature [10] are not suitable for real-time systems.

Recently, garbage-collection algorithms have been described that reasonably bound the delay experienced by an application [5, 2]. However, those algorithms require an extra processor, extra heap space, or both—luxuries that may not be available on a cost competitive, embedded, real-time system. Thus, the expert group that designed the Real-Time Specification for Java (RTSJ) [4] decided to provide a memory management scheme that would allow programmers to circumvent the garbage collector, giving them control over the time at which memory is allocated and reclaimed, while at the same time retaining the storage-referencing safety properties of a garbage-collected language.

A programming language with safe storage references denies access to storage that is dead or that has been reallocated for other purposes. Languages like C++ allow explicit control of object allocation and deallocation, yet they provide no mechanism for checking safe storage references. If RTSJ had adopted the approach of C++, then unsafe storage references would go unchecked, violating a principle design force for Java^{TM}.

The RTSJ, as described in Section 2, extends the Java memory model by providing several kinds of memory areas other than the traditional, garbage-collected heap. These memory areas have different properties in term of the lifetime of objects allocated within them, and in term of the timing guarantees for object allocation and deallocation.

In particular, the *scoped memory* area provides a heap-like area for object allocation. However, garbage collection does not determine when objects in a scoped memory area are dead. Instead, the entire scope can be collected when all threads that have *entered* the area (by invoking a particular method on the area) have apparently abandoned the area (by exiting the method invoked to gain entry). Unfortunately, a thread that has left a scoped memory area could yet possess references into that area. Threads could also develop references to scoped memory areas for which they have never formally declared their intention to use.

Thus, the undisciplined use of the RTSJ could lead to dangling

references that would never occur in standard Java^{TM}. To avoid this problem, the RTSJ comes with a a set of rules that must be enforced—at runtime if necessary—by every compliant Java Virtual Machine (JVM). These rules substitute a runtime exception in place of an unsafe reference.

To summarize, scoped memory areas are a compromise for Java^{TM} and real-time developers, offering some control over storage deallocation while ensuring that unsafe references are checked and reported as such. At issue is the expense of implementing the checks necessary for compliance with the RTSJ.

This paper focuses on the data structure and the algorithms used to implement the RTSJ memory model and to enforce the *safety* rules. The main contribution of this paper is that of showing how all the necessary runtime checks can be performed in constant time, by either using more efficient data structures than those proposed by the RTSJ or by using analogies with problems that arise in type inclusion testing [12].

The remainder of the paper is organized as follows, Section 2 contains an in depth description of the RTSJ memory model and of the implementation techniques currently used; Section 3 presents our improvements, showing how the RTSJ memory model can be implemented efficiently by using better data structures and algorithms; Section 4 presents the results of performance analysis that compares the performances of the algorithms proposed in this paper with those suggested by the RTSJ and used in literature. Section 5 provides an outline of the state of the art and related work; finally Section 6 contains our concluding remarks.

2. THE RTSJ MEMORY SUBSYSTEM

The RTSJ extends the Java memory model by providing memory areas other than the heap. As shown in Figure 1, these memory areas are characterized by the anticipated lifetime of the contained objects (immortal, scoped) as well as the time taken for allocation (linear, variable). Objects allocated within the (singleton) *Immortal Memory* have the same lifetime as the application: they are never collected. Each *Scoped memory* area is equipped with a reference count of the number of threads active in its area. The lifetime of objects allocated in such an area is keyed to the reference count. Figure 1 includes the `LTPhysicalMemory` and `VTPhysicalMemory` areas, which afford raw access to specific locations in an address space. We mention the physical memory areas for completeness; it is their scoped nature that is relevant to the work presented in this paper.

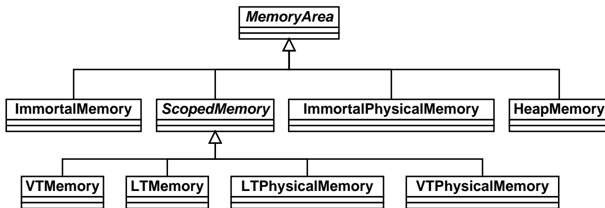

Figure 1: Hierarchy of Classes in the RTSJ Memory Model

Additionally, scoped memory areas provide bounds on the allocation time; currently, variable (`VTMemory`) and linear-time (`LTMemory`) allocators are accommodated. For linear allocation time, the RTSJ requires that the time needed to allocate the $n > 0$ bytes to hold the class instance must be bounded by a polynomial function $f(n) \leq Cn$ for some constant $C > 0$.[1]

[1]This bound does not include the time taken by an object's constructor or a class's static initializers.

For JVM and application developers alike, scoped memory is one of the more interesting features added to Java^{TM} by the RTSJ. Object allocated within a scoped memory are not garbage collected; instead, a reference-counting mechanism detects when all objects in a scope should be collected. Safety of scoped memory areas is ensured by reliance upon (1) a set of rules imposed on entrance of scoped memories, and (2) a set of rules that govern the legality of reference between objects allocated in different memory areas. The remainder of this section concerns memory areas and will provide an explanation of its mechanics.

2.1 Understanding the Scoped Memory Model

To understand the mechanics of scoped memory areas, it is important to understand the RTSJ rules that govern access to those areas. RTSJ assumes that Java^{TM} has its traditional threads, but adds two new real-time thread types: `RealtimeThread` and `NoHeapRealtimeThread`, with access rules as follows:

1. A traditional thread can allocate memory only on the traditional heap.

2. Real-time threads may allocate memory from a memory area other than the heap by making that area the current allocation context.

3. A new allocation context, or scope, is entered by calling the `MemoryArea.enter()` method or by starting a real-time thread whose constructor was given a reference to an instance of `MemoryArea`. Once a scope is entered, all subsequent uses of the `new` keyword, within the program logic, will allocate the memory from the current scope. When the scope is exited by returning from the `enter()` method, all subsequent uses of the `new` operation will allocate memory from the memory area associated with the enclosing scope.

4. A real-time thread is associated with a scope stack containing all the memory areas that the thread has entered but not yet exited.

On the other hand, the rules that govern the scoped memory behavior are the following:

1. Each instance of the class `ScopedMemory` must maintain a reference count of the number of threads active in that instance.

2. When the reference count for an instance of the class `ScopedMemory` is decremented from one to zero, all objects within that area are considered unreachable and are candidates for reclamation. The finalizers for each object in the memory associated with an instance of `ScopedMemory` are executed to completion before any statement in any thread attempts to access the memory area again.

3. Each `ScopedMemory` has at most one *parent*, defined as follows. For a `ScopedMemory` that has been pushed on a scope stack, *i.e.*, entered by at least one thread, its parent is the first instance of `ScopedMemory` below it on the scope stack, if there is one; otherwise, its parent is the *primordial scope*. For as scope not pushed on the scope stack, its parent is null.

Figure 2 depicts three scoped memory areas, A, B, and C, and two real-time thread T_1, and $T2$. In Figure 2, T_1 enters A, B, and then tries to enter C, while T_2 enters A, C, and then tries to enter

B. In Figure 2, circles represents scoped memories while arrows point from a child scope to its parent scope.

If T_1, as shown in Figure 2, tries to enter C after T_2 has entered it, than a compliant RTSJ JVM will detect a violation of the single parent rule and throw an exception. This violation, as visible in Figure 2 by the contents of the scope stack of T_1 and T_2, can be detected by a JVM inspecting the scope stack and checking that no single-parent rule violation happens.

Why is this single-parent rule necessary? The single parent rule guarantees that once a thread has entered a set of scoped memory in a given order, any other thread will have to enter them in the same order, up to the point at which the reference count for all these memory drops to zero. At that point, a new nesting will be possible. This requirement guarantees that a parent scope will have a lifetime that is at least that of any of its child scopes, making it safe for objects in a descendant scope to reference objects in an ancestor scope.

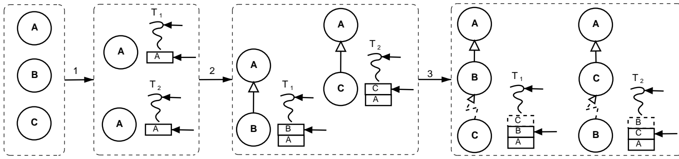

Figure 2: The scope stack and the single parent rule.

Figure 3 extends the example of Figure 2, showing a potential scope tree of an RTSJ application; all nodes of that tree represent scoped memory areas. An object in node x of such a tree can reference an object in node y only if y is an ancestor of x. Thus, the curved arrows in Figure 3 show some (not all) legal references, while the zig-zag arrows represent some (not all) illegal references.

With the single-parent rule, the ancestor relationship described above guarantees that legal accesses occur only from a scope to another at least as long-lived as the former. In other words, no legal references are "dangling."

To enforce the rules, a compliant JVM has to check every attempt to enter a memory area by a thread, to ensure that the single parent rule is not violated, and it has also to check the creation of reference between objects belonging to different memory areas. Since object references occur frequently in Java^{TM} programs, it is important to implement them efficiently and predictably.

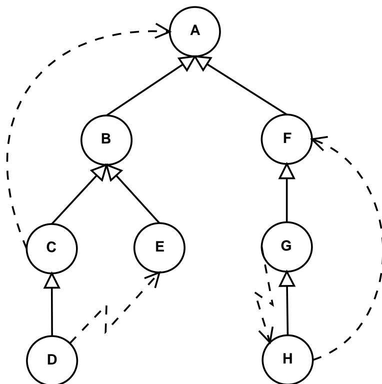

Figure 3: Scope Tree and Scoped Memory Reference Checking Sample.

Algorithm 1: checkSingleParentRule

Input: $MemoryArea$ ma, $ScopeStack$ scopeStack
Output: **boolean** isSingleParentRuleOK
begin
  isSingleParentRuleOK ← true;
  if *ma* ***instanceOf*** $ScopedMemory$ **then**
    parent = `findFirstScope` (scopeStack);
    if *ma.parent* = ***nil or*** *ma.parent* = *parent* **then**
      ma.parent ← parent ;
      scopeStack.push (ma);
      ma.refCount ← ma.refCount + 1;
    else
      isSingleParentRuleOK ← **false** ;
end

Algorithm 2: findFirstScope

Input: $ScopeStack$ scopeStack
Output: $ScopedMemory$ firstScope
begin
  firstScope ← PrimordialScope ;
  for i ← *scopeStack.size()*-1 ***downto*** 0 **do**
    if *scopeStack*$[i]$ ***instanceOf*** $ScopedMemory$ **then**
      firstScope ← scopeStack $[i]$;
      break ;
end

2.2 RTSJ Suggested Runtime Check Implementation

We next present the current state of algorithms for scope and reference checks, as suggested by the RTSJ and its current implementations [11, 3]. we provide examples that explain why this approach is not satisfactory for real-time applications.

2.2.1 Data Structures

The RTSJ assumes that (1) there is a scope stack associated (at least logically) with each real-time thread, (2) each memory area keeps a reference to its parent, (3) scoped memories keep track of their reference count, and (4) for any object, it is possible to obtain a reference to the memory area that contains it. The algorithms used to enforce the single-parent rule and the assignment rules are based on these data structures.

2.2.2 Single Parent Rule

The single parent rule is enforced at the point a real-time thread tries to enter a scope s. At that time, if s has no parent, then entry is allowed. Otherwise, the thread entering s must have entered every proper ancestor of s in the scope tree. Algorithm 1 and Algorithm 2 contain the pseudocode that performs this test.

Examination of these algorithms reveals a time complexity of $O(n)$ where n represents the depth of the stack.

2.2.3 Memory Reference Checks

The rules that govern the validity of references across memory areas can be summarized as follows:

1. A reference to an object allocated in a `ScopedMemory` can never be stored in an object allocated in the Java heap or in the immortal memory.

Algorithm 3: checkReferenceValidity

```
Input: MemoryArea from, MemoryArea to, ScopeStack scopeStack
Output: boolean validReference
begin
    validReference ←true;
    if from ≠ to then
        if to instanceOf ScopedMemory then
            if from instanceOf ScopedMemory then
                toDepth = depth (to, scopeStack );
                if toDepth = inf then
                    validReference ←false;
                else
                    fromDepth = depth (from, scopeStack );
                    deltaDepth = toDepth − fromDepth ;
                    if not (0 <deltaDepth < inf) then
                        validReference ←false;
            else
                validReference ←false;
end
```

Algorithm 4: depth

```
Input: MemoryArea ma, ScopeStack scopeStack
Output: int depth
begin
    depth ← inf;
    index ← scopeStack.size() - 1;
    while index > 0 and scopeStack[index] ≠ ma do
        index ← index −1;
    if scopeStack[index] = ma then
        depth ← scopeStack.size()−index −1;
end
```

2. A reference to an object allocated in a `ScopedMemory` m can only be stored in objects allocated in a `ScopedMemory` p only if p is a descendant of m; note that the case $p = m$ is thus allowed.

The RTSJ specification does not explicitly provide an algorithm for checking the legality of a memory reference, but two approaches could potentially be taken. In one approach [11, 3], essentially following the advice given in the RTSJ specification, a thread's scope stack can be scanned to ensure that the memory area from which we are creating a reference was pushed later than the memory area of the reference's target. This approach is described by the Algorithm 3 and Algorithm 4. By inspection of the pseudocode, this check has time complexity $O(n)$ where n is the depth of the scope stack.

We present a slightly more efficient variation in Algorithm 5. This algorithm, based on the observation that reference from scoped memory to heap or immortal memory are always legal, and the opposite is always illegal, avoids scanning the scope stack, and simply follows a scope's parent link to discover if the target reference is in an ancestor scope of the source.

It is well known that for real-time applications, it is important to be able to put bounds on the execution of operations within some piece of code, and for code within a whole application. Real-

Algorithm 5: checkReferenceValidity2

```
Input: MemoryArea from, MemoryArea to
Output: boolean validReference
begin
    validReference ←true;
    if from ≠ to then
        if to instanceOf ScopedMemory then
            if from instanceOf ScopedMemory then
                ancestor ← from.getParent();
                while to ≠ ancestor and ancestor ≠ nil do
                    ancestor ← ancestor.getParent();
                if to ≠ ancestor then
                    validReference ←false;
            else
                validReference ←false;
end
```

time applications inevitably contain code fragments whose execution time must be statically known. The scope stacks for an application—in particular, their depth—are not necessarily decidable at compile-time. Thus, linear-time algorithms for checking the single-parent rule and the memory references may incur unpredictable overhead. As a result, the application must either overprovision, thus wasting resources, or else underprovision, and perhaps miss a critical deadline.

Moreover, the time spent checking a given memory reference using the above algorithms will vary depending on the particular scope stack in place when the check is performed. Even a simple store instruction will thus take varying amounts of time depending on the scope stack. This in prevents analysis of the code's timing in any particular context, but makes it necessary to analyze its timing in *all* the different contexts in which it could be executed. In our experience in developing an Object Request Broker (ORB) for RTSJ, such unpredictability makes it impossible to bound reasonably the time for servicing clients' requests.

3. OPTIMIZING THE RTSJ MEMORY SUBSYSTEM

We have thus far described the checks required by a compliant RTSJ implementation, and we have explained why the algorithms currently present in literature [4, 3, 11] are not ideal for real-time applications. We next show how to extend the data structures used by the RTSJ to perform all required checks in constant time. Our approach is inspired and based upon type-inclusion tests for single-inheritance, object-oriented languages [12].

We have implemented the algorithms described here in the jRate [8] memory subsystem; for convenience, we describe our approach in the context of jRate.

3.1 Optimizing the Singe Parent Rule Check

jRate's scope-stack implementation uses data structures that allow all scope stack operations to be performed in constant time. As shown in Figure 4, jRate augments the scope-stack data structure suggested by RTSJ, linking those slots that represent scoped memory areas. An index is also maintained to the topmost scoped memory, so that the next scoped memory area pushed onto the stack can be linked with the others.

This design allows a constant-time implementation of $findFirst-Scope()$, while at the same time maintaining constant-time bounds

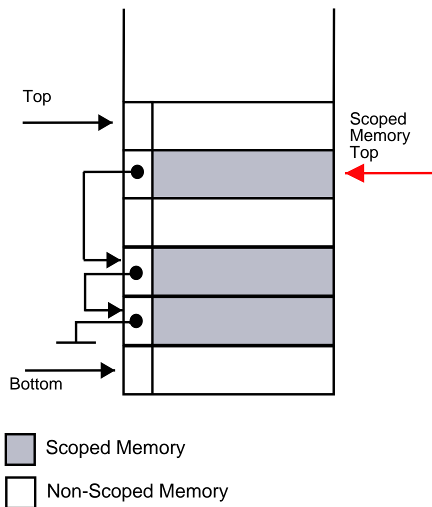

Figure 4: The jRate Scope Stack structure.

Algorithm 6: findFirstScope

Input: $ScopeStack$ ss
Output: $ScopedMemory$ firstScope
begin
  firstScope $\leftarrow$ $primordialScope$;
  if *ss.lastSMIndex* $\neq$ $STACK_END$ **then**
    firstScope $\leftarrow$ ss [ss.lastSMIndex].ma ;
end

for $push()$ and $pop()$.

Algorithm 6 provides the pseudocode for the constant-time implementation of $findFirstScope()$, while Algorithm 7 and Algorithm 8 provide the pseudocode for the $push$ and pop operations. On inspection it can be seen that all of these operation are performed in constant time. As compared with the RTSJ-inspired implementations, no scanning of the scope stack is required.

The actions required for processing memory areas require knowing what kind of memory area is at-hand (immortal, scoped, etc). While such information could be determined by Java^{TM}'s `instanceof` test, that test may require time linear in the depth of the program's class hierarchy. Instead, jRate] optimizes such tests by tagging memory area objects to allow a constant-time test.

This enhancement of the scope stack structure makes it possible to know exactly which entries of the scope stack are scoped memories and which are not. This knowledge enables it to elide a test on the type of memory area, and avoids blindly searching for scoped memories on the stack to decrement the reference count when destroying the scope stack. As a final note, the size of a jRate real-time thread's scope stack can be fixed at thread-creation time. This design makes jRate more space efficient by avoiding the use of pointers to implement the linked list.

3.2 Optimizing the Memory Area Reference Check

To understand the main idea behind our technique for implementing the reference checks in constant time, consider a generic scope stack tree, such as the one in Figure 5. In this example, some of the memory areas are scoped and some are not. The scope stack of any running real-time thread can be represented as a path from the root down to some interior or leaf note. As stated in Sec-

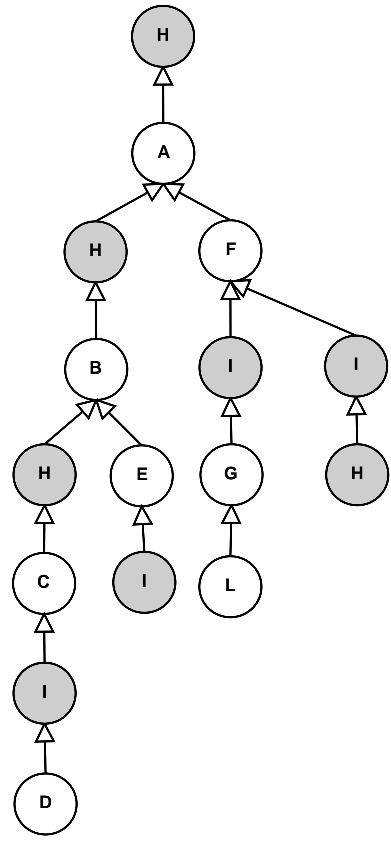

Figure 5: A sample Scope Tree structure (Grey nodes represent the Heap(H), and Immortal (I) Memory).

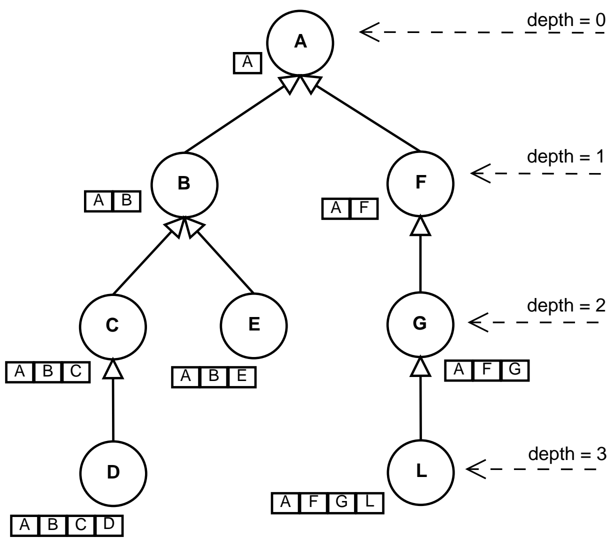

Figure 6: Transformed and Decorated Scope Tree structure.

Algorithm 7: Push

```
Input: MemoryArea ma, ScopeStack ss
begin
    if ma instanceOf ScopedMemory then
        if ma.getParent() = nil then
            ma.setParent(findFirstScope( ss ));
        else
            if ma.getParent() ≠ findFirstScope( ss )
            then
                throw ScopedCycleException;
        ss.top ← ss.top +1;
        ss [ss.top ].ma ← ma ;
        ss [ss.top ].prevSM ← ss.lastSMIndex ;
        ss.lastSMIndex ← ss.top ;
    else
        ss.top ← ss.top +1;
        ss [ss.top ].ma ← ma ;
end
```

Algorithm 8: Pop

```
Input: ScopeStack ss
begin
    if ss [ss.top ].ma instanceOf ScopedMemory then
        ss.lastSMIndex ← ScopedMemory[ss.top
        ].prevSM ;
    ss.top ← ss.top −1;
end
```

tion 2.2.3, references can always be made from objects in a scoped memory to object in the heap or immortal memory; the opposite is never allowed. Also, the ancestor relation among scope memory areas is defined by the nesting of the areas themselves; intervening entry into the heap or immortal memory areas do not affect the scopes. We call the tree implied by the scoped memory areas' ancestor relation the *parenthood tree*. For instance, collapsing all the nodes that represent either the heap or immortal memory in the scope tree of Figure 5, we obtain the tree depicted in Figure 6. In this tree, a direct edge from node B to node A, means that A is the parent of B.

The advantage of this formulation is that *subtype-testing* algorithms [12] can be applied to the parenthood tree to determine legal references.

Relying on this observation, we can use a technique based on displays, similar to that proposed by Cohen [12], to determine the validity of a memory reference in constant time. The *parenthood tree* can change with time, since the scopes' parent-child relation changes as the application runs and scoped memory areas are entered and exited. Thus, the associated type hierarchy is not fixed, but changes at well-defined points—when the reference count of a scoped memory goes from zero to one or from one to zero. At such points, the typing information associated with a scoped memory has to be created and destroyed, respectively.

To facilitate a constant-time memory-reference check, we augment the information associated with a scoped memory to include its depth in the *parenthood tree* and a display that contains the type identification codes of its ancestors in the parenthood tree.

The address of a scoped area serves nicely as a unique type identifier, thanks to the single-parent rule: once a scoped memory area is reclaimed, no memory area can store its address in its display. This avoids the need to map a storage area to a unique number and improves the efficiency of our algorithms.

For example, if we consider the scoped memory C in Figure 6, its depth will be 2 and its display will be (A, B, C). Notice that the depth is the same as the display length minus one, so an implementation won't necessarily need to store this information twice. Here we treat them separately only to simplify exposition of our algorithms. Algorithm 9 contains the pseudocode that shows how, with these extensions, it is possible to perform the memory reference check in constant time. In this algorithm it is assumed that both the heap and the immortal memory have a depth of -1.

Algorithm 9: checkReferenceValidity

```
Input: MemoryArea from, MemoryArea to
Output: boolean validReference
begin
    validReference ←false;
    if from.depth ≥ to.depth then
        if to.depth = −1 then
            validReference ←true;
        else
            if from.display [to.depth ] = to then
                validReference ←true;
end
```

Finally, we note that the management of displays does not add considerable complexity when entering or exiting a memory area. When a scoped memory is entered for the first time, or after its reference count has dropped to zero, setting up the display simply requires copying the parent's display and adding itself at the end. The only operation required when the last thread leaves the scoped memory is to invalidate the display.

In summary we have described a constant-time algorithm that offers far greater predictability and asymptotic efficiency over the approach currently implemented for RTSJ. We next explore the practical efficiency of our approach.

4. PERFORMANCE EVALUATION

We next present the results of a performance comparison between the memory-reference checking scheme proposed in this paper and the one proposed by the RTSJ. Below we describe the testbed, the tests performed, and the results obtained.

4.1 Testbed

The platform used for running the test was a Pentium III, 733 MHz processor with 256 MB of RAM, running Linux RedHad 8.0 with the kernel v2.4.18–24.8.0. The latest version of jRate, which is available at `http://tao.doc.wustl.edu/~corsaro/jRate` was used in the performed test. jRate [8] is an open-source RTSJ-based real-time Java implementation currently under development at Washington University. jRate extends the open-source GNU Compiler for Java (GCJ) runtime system [9] to provide an ahead-of-time compiled middleware platform for developing RTSJ-compliant applications.

4.2 Timing Measurements

An issue that arises when conducting benchmarks concerns the timing mechanism that provides the results. To ensure precise measurements we used our own native timers, which, on all Pentium-

based systems, rely on the *read-time stamp counter* RDTSC[2] instruction [6] to obtain timing resolution in terms of the CPU clock period.

4.3 Test Description and Results

To compare the performance of our approach against that of other systems based on the RTSJ, we implemented both algorithms in jRate. Other RTSJ systems to-date follow Algorithm 3—scanning the scope stack to determine the validity of a reference check. But, if the $instanceof$ tests can be carried out in constant time[3], then Algorithm 5 is more efficient since there is no need to scan the scope stack. Instead, the scope hierarchy is consulted. We therefore implemented and compared the better Algorithm 5 with our display-based approach as described by Algorithm 9.

Both algorithms are implemented in C++, on the native side of jRate. To better estimate the performance of the two algorithms, and to avoid the overhead and interference of Cygnus Native Interface (CNI), we instrumented the native code directly to measure times for both approaches.

In order to compare the efficiency of the different memory reference algorithms, we extended the RTJPerf[4] [7] benchmarking suite, with a test that creates a reference from a scoped memory B to a scoped memory A; where A is the nth ancestor of B, *i.e.*, an ancestor that has a distance of n from B. The values of n considered were 0, 1, 2, 4, 8, 16, 32, and 64. The time for performing the reference check was computed as the average of 2000 samples.

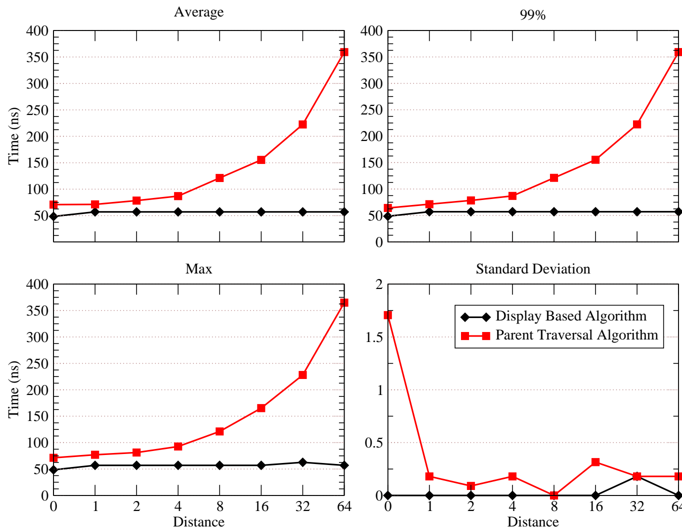

Figure 7: Performances comparison between Display based vs. Parent traversal algorithm.

Figure 7 shows the average, 99%, maximum, and standard deviation for the two algorithms. Figure 8 shows the average check time, the difference between the two averages, and the speedup.

As can be seen from both Figure 7, and Figure 8, the display-based memory-reference check outperforms the parent-traversal-based check in each of the test cases. Moreover, as claimed in Section 3.2, the experiments confirm that the checks execute in constant time, regardless of the depth of the scope stack. The display based algorithm, not only provides average constant time checks, but its worst case performance indexes, *i.e.*, 99% and max, are very closely bound—practically identical—to the average.

[2] The RDTSC is a 64-bit counter that can be read with the x86 assembly instruction RDTSC.

[3] jRate uses a custom encoding for those classes that require frequent `instanceof` tests. This allows `instanceof` to be replaced by an integer comparison or a bitwise-and operation.

[4] RTJPerf is currently the only available RTSJ benchmarking suite

	Display	Parent Traversal	Delta	Speedup
0	48.460 ns	70.830 ns	22.370 ns	1.461
1	57.011 ns	71.270 ns	14.259 ns	1.250
2	57.011 ns	78.394 ns	21.382 ns	1.375
4	57.011 ns	86.948 ns	29.937 ns	1.525
8	57.011 ns	121.149 ns	64.138 ns	2.125
16	57.011 ns	155.366 ns	98.355 ns	2.725
32	57.017 ns	222.351 ns	165.334 ns	3.899
64	57.011 ns	359.178 ns	302.166 ns	6.300

Figure 8: Average reference check time.

Note that the results provided by this test also predict the timing behavior of these algorithms for *invalid* memory references. For the display-based approach, the time taken to detect a valid or an invalid reference is the same, and does not depend on the structure of the parenthood tree. On the other hand, if we consider the parent-traversal algorithm, the time necessary for detecting an invalid reference (in the case of two scoped memories) requires traversing the parents path to the root. Thus, results shown in Figure 7, and Figure 8 essentially determine the time taken to check an invalid reference when the $from$ scoped memory has a distance of n from the root of the *parenthood tree*.

5. RELATED WORK

Most of the work available in literature, which addresses the implementation of the RTSJ memory subsystem, such as [11], and [3], use the same algorithms proposed by the RTSJ, by scanning the scope stack, in order to perform the memory reference checks. Thus, all these works provide $O(n)$ algorithms for checking both the single parent rule and the memory reference checks.

Some of the work present in literature, on compositional pointer and escape analysis, such as [14], could be used as compile time techniques to remove, in some cases, run-time checks. But undecidability issues may imply that some, perhaps many, checks may be required at run-time. These techniques could be combined with the algorithms proposed in this paper in order to provide a good speedup, and more predictability to RTSJ applications.

The concept of RTSJ memory areas, is borrowed from the more general concept of regions which was first introduced by Tofte et al., in [13]. While, the display based approach for constant time type-extension type test was first introduced by Cohen in [12]. The main difference between our case and the approach described in [12] is that our type hierarchy changes with time, so the type encoding cannot be fixed at compile time.

6. CONCLUDING REMARKS

We have introduced the concept of *parenthood tree*, and shown how the memory reference checking problem can be reformulated as a subtype-testing problem on a type hierarchy associated with the *parenthood tree*. We have presented algorithms that reduce the time necessary to support the RTSJ-mandated checks from linear to constant-time. The algorithms are implemented in jRate which is open source and available for experimentation. Our results confirm the predictability of our approach as well as its overall efficiency.

7. REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, Boston, 2000.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298. ACM Press, 2003.

[3] W. S. Beebee and M. Rinard. An implementation of scoped memory for real-time Java. *Lecture Notes in Computer Science*, 2211, 2001.

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

[5] P. Cheng and G. Belloch. A parallel, real-time garbage collector. *ACM SIGPLAN Notices*, 36(5):125–136, May 2001.

[6] I. Coorporation. Using the RDTSC Instruction for Performance Monitoring. Technical report, Intel Coorporation, 1997.

[7] A. Corsaro and D. C. Schmidt. Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems. In *Proceedings of the* 8th *IEEE Real-Time Technology and Applications Symposium*, San Jose, Sept. 2002. IEEE.

[8] A. Corsaro and D. C. Schmidt. The Design and Performance of the jRate Real-Time Java Implementation. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, pages 900–921, Berlin, 2002. Lecture Notes in Computer Science 2519, Springer Verlag.

[9] GNU is Not Unix. GCJ: The GNU Complier for Java. `http://gcc.gnu.org/java`, 2002.

[10] R. Jones and R. Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management*. Wiley & Sons, New York, 1996.

[11] M. A. d. M.-C. M. Teresa Higuera-Toledano. Dynamic Detection of Access Errors and Illegal References in RTSJ. In *Proceedings of the* 8th *IEEE Real-Time Technology and Applications Symposium*, San Jose, Sept. 2002. IEEE.

[12] Norman H. Cohen. Type-Extension Type Tests Can Be Performend In Constant Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

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

[14] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.