Region-Based RTSJ Memory Management: State of the art

H. Hamza, S. Counsell*
Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK

ARTICLE INFO

Article history:
Received 25 October 2010
Received in revised form 11 January 2012
Accepted 11 January 2012
Available online 27 January 2012

Keywords:
RTSJ memory management
Scoped memory
Immortal memory
Memory regions
Real-time Java
Benchmarks
Programming languages

ABSTRACT

Developing a real-time system in Java requires awareness of memory behaviour in addition to software functional requirements. The Real-Time Specification for Java (RTSJ) introduces a scoped memory model to avoid garbage collection delays in critical real-time applications which need to meet hard real-time constraints. Scoped memory management has certain advantages over garbage collection in terms of predictability. However, developing a real-time application using scoped memory areas (regions) may suffer from both design and runtime errors. Moreover, from a memory footprint perspective, the inability to determine precisely how many scoped memory areas should be used and which objects or threads should be allocated into these scoped memory areas makes using RTSJ problematic for developing real-time systems. In this paper, a survey of the current approaches to improve scoped memory management and new emerging challenges in RTSJ scoped memory management model are presented. Categorizing those problems and challenges provides a picture of the issues researchers have yet to investigate and to support solutions for an optimal scoped memory model. Current approaches and a set of benchmarks used to evaluate current solutions are presented and new research questions in developing real-time Java systems using a scoped memory model are proposed.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

By their nature, real-time systems are characterized by limited resources in terms of memory size, power and processor speed [1,2]. Real-time systems can be divided into two main categories: soft, real-time systems and hard, real-time systems. The former is tolerant of missed deadlines without generating an error condition, while the latter cannot afford to miss a deadline [3]. A fault in either type of system can cause catastrophic results or loss of human life and, at the very least, be a significant financial setback [4,5]. These faults can be the result of many factors such as an inadequately designed memory model, miscalculation of deadlines or unexpected power failures. A badly designed memory management model may, for example, reclaim objects that still need to be accessed again by the software, or, in the context of limited available space, keep unused objects for a long time in memory without being reclaimed.

Memory management in real-time Java systems is still an open research area. Developers have to ensure that the systems they design are predictable in terms of memory behaviour and also that they meet real-time event deadlines without being affected by memory reclamation techniques [6]. The Real-Time Specification for Java (RTSJ) outlines seven areas of enhancements for real-time applications. These are: thread scheduling with priority based techniques, new memory management based on scope techniques where garbage collection does not interfere, resource sharing management, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination and physical memory access (when the system is connected to specialized hardware) [3]. The new RTSJ programming model is based on semi-explicit memory management in which allocation of objects into memory areas is undertaken by the developer. This new
The concept of RTSJ memory areas is borrowed from the more general concept of memory regions first introduced by Tofte et al., [9]. The predictable behaviour of the new RTSJ memory model makes it suitable for hard, real-time systems where determinism is the first requirement needing to be satisfied [10].

Nevertheless, development of applications using a scoped memory management model is a difficult task and has spawned research to help developers design their application memory model [11, 12]. Research has found that scoped memory management has many drawbacks. First, there is the increased development complexity; such a model needs many additional classes for proper management and possibly application of specific design patterns (e.g. the multi-scoped object pattern and the handoff pattern [6]). Second, the memory model in one application cannot be adapted to other applications, since the design of a scoped memory model requires information about the object and thread lifetimes of that application which, in turn, differ from one application to another. Third, the model needs precise knowledge of object lifetimes to determine how many scoped memory areas are required and which objects reside in which scoped memory areas. Finally, any scoped memory model needs to ensure safe references among objects allocated in different memory areas; otherwise, the resulting model could introduce runtime errors [13–19], which in turn produces a burden on the developer. It also constrains the design of the application’s memory model to allocate application objects that have different lifetimes into specific scoped memory areas.

The extent to which real-time and embedded Java-based systems are becoming more prominent in real, industrial settings is evidenced by a number of examples. The autonomous navigation capabilities of the ScanEagle unmanned aerial vehicle developed by Boeing and Purdue University [20], a motion control system developed by Robertz et al., [21], IBM’s comprehensive battleship computing environment and commercial real-time trading systems described in [12] are four such systems. The versatility of real-time and embedded systems is generally accepted and, from that perspective alone, we see their role as becoming increasingly important. However, ensuring the robustness of the memory model used in these systems is one of the primary concerns of the verification process. Several issues in an RTSJ scoped memory model need to be categorized to provide a full awareness of the challenges in this area. In the ensuing sections, we present a detailed description of the state-of-the-art in the RTSJ scoped memory model. An overview is provided that gives a broad understanding of the different issues and highlights existing problems that need to be tackled. The benchmarks used in the literature to evaluate the implementation of RTSJ scoped memory are also presented. This overview of RTSJ benchmarks can help to identify current case studies and their benefits and also shed light on the need for future benchmarks that verify and demonstrate the functionality of a given scoped memory management model.

The remainder of this survey paper is structured as follows: Section 2 provides background and introduces the scoped memory management of RTSJ. Current problems using scoped memory in RTSJ and their existing solutions are then introduced (Section 3). Section 4 describes a set of benchmarks with which to evaluate the implementation of an RTSJ scoped memory model. New research directions and possible solutions are discussed in Section 5. Finally, we draw some conclusions in Section 6.

2. Background

Memory management in early programming languages such as Fortran was static. In other words, the location of variables was statically defined at compile time and fixed at runtime. Static memory management has many disadvantages. The most prominent of these is that the developer has to define (in advance) the size of all variables allocated in memory—a fixed size memory is reserved during execution of the application. Reclaiming memory is not permissible while the application is still running and defining dynamic data structures at runtime is not possible in programming languages that use only static memory management. This has motivated research efforts to introduce dynamic memory management models where data structures can be dynamically defined at runtime. Some of these dynamic memory models are manual, for example in programming languages such as C and Pascal. However, a manual dynamic memory management model is susceptible to dangling pointers and memory leaks due to programming pitfalls [22]; a ‘memory leak’ is said to occur when unclaimed dead objects no longer reachable by the application remain in memory for a relatively long time [23]. The alternative model of dynamic memory management is ‘automatic’ typified by the garbage collection technique employed by the Lisp and Java programming languages [24]. However, applications may still suffer from unexpected delays due to garbage collection interrupts during the memory reclamation process. Such delays are unacceptable in real-time and critical systems [25]. Consequently, new real-time garbage collection algorithms in Java have been proposed and implemented in commercial products for real-time systems, but there are still many research challenges in real-time garbage collection for decreasing pause times and space overheads [26]. Definition of application parameters is necessary to calibrate the real-time garbage collector. One such example is the maximum allocation rate (bytes per clock cycle) which specifies the intervals of time between which the garbage collection is invoked; this can be problematic with respect to achieving low time and space overheads in an application [27–29].

2.1. RTSJ scope principles

In traditional Java, all objects are allocated from heap memory and are subject to garbage collection. Heap memory is “a pool of memory available for the allocation and de-allocation of arbitrary-sized blocks of memory in an arbitrary order”
Each block, a number of bytes known as single allocation unit used for heap memory stores application objects. In Java, the heap is the area of memory where the garbage collector searches for objects to free more space for future dynamic allocations. Failure to de-allocate dead objects (i.e. objects that will not be used again by the application) may eventually result in an out-of-memory space error for subsequent dynamic allocations.

The RTSJ provides, in addition to the heap memory, two other types of memory: (a) immortal memory which stores objects that remain alive until the application terminates and, (b) scoped memory which has a bounded lifetime and where objects of similar lifetime should reside. There is only one immortal memory instance and it is created when the real-time Java VM starts. Immortal memory and scoped memory areas are only entered by schedulable objects (real-time threads or asynchronous event handlers). Scoped memory can be assigned by parameters to specify the initial and maximum size of the scoped memory areas in bytes and optionally by the Runnable object that executes within the scope. Each scope can be entered by many schedulable objects which will allocate objects inside the scope. Objects in the scope cannot be reclaimed individually—the whole scope has to be freed at the same time, giving the application predictable timing behaviour. Scoped memory uses a reference counting technique to free its contents. For example, each time a schedulable object enters a scoped memory passing a Runnable object to be executed in that scoped memory, the reference count increases by one. Conversely, when the Runnable object finishes executing within the scope the reference count decreases by one. If the reference count reaches zero, objects are freed and the scope is marked for reuse.

The RTSJ also introduces new classes of real-time threads, RealtimeThread and NoHeapRealtimeThread. A RealtimeThread class has a more precise set of scheduling characteristics than a standard Thread class in Java. A NoHeapRealtimeThread or RealtimeThread instance can pre-empt garbage collection and access the heap memory area. For instance, the real-time garbage collector (RTGC) in Sun RTS 2.0 can be prompted by NoHeapRealtimeThreads and RealtimeThreads with priorities higher than the RTGC; however, the RTGC in Sun RTS 2.0 can boost its priority to a higher configurable-programmer level by the VM when the amount of free memory falls below a pre-defined threshold. However, if the garbage collector is running and the RealtimeThread starts, the latter has to wait for the garbage collector to reach a safe pre-emption point (when all scanned objects in the heap are marked as either alive or dead); at that point, the garbage collection can be pre-empted by the RealtimeThread without impacting the consistency of the heap. The NoHeapRealtimeThread is similar to RealtimeThread but does not access the heap and therefore does not interfere with the garbage collection process. However, in some cases, the developer is advised to avoid NoHeapRealtimeThread overwriting objects allocated in immortal memory to avoid unexpected interaction with the garbage collector. This occurs when object B (allocated in the heap) needs to be modified as a consequence of overwriting object A (allocated in the immortal memory) by the NoHeapRealtimeThread. Subsequently, the NoHeapRealtimeThread may be forced to wait for the garbage collection that runs in the heap to finish its cycle.

2.2. RTSJ Memory management APIs

The MemoryArea class is an abstract class from which different memory subclasses are inherited. One of its subclasses, ScopedMemory also has two subclasses: VTMemory and LTMemory. In LTMemory, allocation time is linear with respect to object size if the space used within the scope is less than the initial size, while allocation time varies in VTMemory depending on the memory allocation algorithm used in an RTSJ implementation. Scopes can also be nested in RTSJ. Nesting occurs when a schedulable object enters a scoped memory area; while executing in that scoped memory, the schedulable object enters another scoped memory area; the first scoped memory area becomes the parent of the second.

Fig. 1 shows an example of a RealtimeThread forming nesting scoped memory areas (A, B, and C). A stack of scoped memory areas is created for the thread to maintain the sequence where scoped memory areas have been entered. So the scope stack of each thread contains the list of all scoped memory areas entered by the thread in order. In other words, while executing code by a thread in the scope of memory 'A', an enter method for the scope of memory 'B' might be called. Henceforward, we will call 'A' the parent (outer scope) and 'B' the child (inner scope) since objects allocated in A, by definition, have a longer life than objects allocated in B. Since a scope can be entered by many threads at the same time, it can be a parent of many other scoped memory areas.

The key advantage of using nested scoped memory areas is the potential advantage of memory savings since the 'child' (inner scope) memory areas have shorter lifetimes than their (outer scope) parent. As a technique, nesting can be used when a schedulable object needs to allocate different objects that have different lifetimes into memory; the developer then allocates these objects into different nested scoped memory areas according to object lifetimes. Objects in the child scoped memory areas are de-allocated as soon as the schedulable object has finished executing in that child scope; dead objects in the child scope thus never wait for objects in the parent scope to die before being de-allocated themselves. The following is the list of methods to obtain information about the memory scope area:

- get_current_memory_area(): static method that returns the current allocation context.
- get_memory_area(): non-static method that returns the initial memory area used.
- get_memory_area_stack_depth(): returns the size of the current schedulable object's scope stack.
- get_outer_memory_area(index) returns a reference to the memory area at the stack at index given. Stack access is zero-based.
Fig. 1. A RealtimeThread forms nesting scopes, scope stack is created.

- **enter()**: to enter a memory scope where all new created objects in ‘run’ method of the Runnable object or the schedulable objects will be allocated inside this scope.
- **executeInArea()**: if code is executed in the child scope and some part of it needs to be executed in the parent code, executeInArea method can be used to change the current allocation context.
- **getReferenceCount()**: is used with ScopedMemory class and it returns the reference count of this scoped memory area.
- **memoryConsumed()**: returns the amount of memory consumed in bytes of the current memory area.
- **memoryRemaining()**: returns the amount of remaining memory of the current memory area.

### 2.3. Scoped memory reference semantics

Since many memory areas (scoped memory, immortal memory, heap memory) may exist in an application, there are limitations on how objects inside them may hold a reference to objects in different memory areas. The RTSJ rule is that a memory scope with a longer lifetime cannot hold a reference to an object allocated in a memory scope with a shorter lifetime, otherwise dangling references could occur at runtime (i.e., pointers to objects which are no longer considered alive). When an object holds a reference to another object, it implies that the first object calls the other object’s method or variables. For example, all objects, wherever they reside, can hold references to objects in immortal memory: such memory will never be reclaimed during the application’s execution time, so no dangling reference can occur. Similarly, objects in heap and immortal memory must never hold references to objects in scoped memory areas as these may be freed at any time (de-allocating objects in a scoped memory area is not subject to the garbage collection process and is technically independent of de-allocation of objects in other scoped memory areas).

A scoped memory area cannot hold a reference to an object allocated to an inner scope. Since scoped memory areas can be shared by different schedulable objects, a single parent rule should be applied to avoid scope cycling, which occurs when two or more schedulable objects enter a different number of scoped memory areas at the same time. For example, assume a real-time thread T1 enters scope A then B. If, at the same time, a T2 real-time thread tries to enter scope B then A, this is prohibited by the single parent rule which ensures each scoped memory has one parent scope. In other words, each scope has one parent and all schedulable objects should follow the same sequence of entering the scoped memory areas. Any wrong assignment by the developer results in a runtime error; equally, exceptions such as IllegalAssignmentError, ScopedCycleException are thrown on attempted violations of the memory access rules and the single parent rule [3]. Table 1 summarizes the assignment rules between memory areas to avoid dangling references at runtime.

### 2.4. Scoped memory in non-RTSJ Java virtual machines

Scoped memory management implemented in Java RTS virtual machines has some distinct features that make it different from region-based memory management implemented in non-RTSJ Java virtual machines. One of these features is that in
Table 1
Assignment rules [3,8].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Immortal</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Scoped</td>
<td>Yes</td>
<td>Yes</td>
<td>Only if objects reside in the same scoped memory areas or in the outer scoped memory</td>
</tr>
<tr>
<td>Local variables</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

RTSJ, scoped memory areas are created explicitly and objects allocated into scoped memory areas manually—de-allocation of the scoped memory areas and finalizing of objects is undertaken automatically by the virtual machine. Finalizer methods are used to clean up legacy code and temporary files. Object finalizer methods are discouraged in RTSJ because of their unpredictability and their impact on schedulability analysis [33]. In other standard Java virtual machines that (potentially) can include region-based memory management, both allocation and de-allocation are achieved manually or explicitly. For instance, Cherem and Rugina [34] transformed Java code into region annotation-based code that included creating, removing and passing regions as parameters and allocating objects into these regions. All regions were created in heap memory. The static analysis was used to define region and object lifetimes; significant free space was saved in some of the Java Olden benchmarks (such as power and tsp benchmarks). On the other hand, for bh, health, and Voronoibenchmarks, the GC system was an improvement in terms of memory saving as an indication of static analysis drawbacks. However, their approach for complex applications did not reclaim any memory as all objects were placed into only one immortal region; this could have caused a memory leak, since static analysis gives only approximations of object lifetimes. Another approach to developing Java virtual machines using scoped memory was that proposed by Garbervetsky et al. [35], where creation instructions are inserted at the beginning of each method, together with exit statements for that scope at the end of the method, as the following example illustrates:

```java
// This code is not RTSJ code, it is written for a non-RTSJ virtual machine
void m0(int k)
{
    ScopedMemory.enter(new Region("m0"));
    // define new objects to be allocated in the scoped memory
    ScopedMemory.determineAllocationSite(RegisterExample.m0_2);
    ScopedMemory.exit();
}
```

At the beginning of the method m0, a scoped memory is entered and all objects allocated by the method m0 are stored in that scoped memory area; in the last line of the method m0, an exit statement is inserted to exit the scoped memory area. To decrease the impact of fragmentation in scoped memory (i.e., holes in memory resulting from freeing blocks randomly [30]), run time analysis was undertaken in [35] to allocate objects into either the scoped memory related to the current method they were created in, or to the parent scoped memory belonging to the methods in the call stack of the current method. Their approach eliminated runtime reference checks between scoped memory areas; runtime analysis was used to minimize fragmentation. Objects were allocated into one of the available candidate scoped memory areas according to a given performance criteria (e.g., minimizing memory, fragmentation). The approach required the logging of non-trivial amounts of runtime information about scoped memory areas’ remaining sizes and non-fragmented spaces in them. A prototype of the tool to automate the transformation of the application was developed, but it lacked the manipulation of both multithreading and recursion and, in our opinion, requires evaluation on different real-time case studies.

3. Current problems and existing solutions

Many problems with using scoped memory management have been described in the literature. For example, Beebee and Rinard [36] claim that real-time Java programs often need the help of other debugging tools and static code analysis to avoid convoluted errors occurring; examples include reference check errors and memory leaks. In this section, we categorize these problems to understand the different obstacles in the use of scoped memory in RTSJ.

3.1. Time overheads

Time overheads result when the virtual machine checks for every assignment between two objects obj1.v1 = obj2.v2 allocated into two scoped memory areas, and for every attempt to enter a memory area by a schedulable object to preserve the single parent rule among scoped memory areas. In Hamza and Counsell [37], the features of scoped memory in RTSJ were explored for large numbers of objects and investigated the effects of varying numbers of allocated objects in the context...
of nested scoped memory areas when compared with un-nested. Results showed that more scoped memory areas led to increases in execution time and when nested scoped memory areas were used, execution times increased proportionately. This indicated that the SUN RTS 2.2 virtual machine scans the scope stack, regardless of its depth, to perform memory reference checks.

There are two aspects that need to be considered to overcome time overheads. The first is to improve assignment rule implementation and reduce time checking at runtime. The second is to eliminate the use of reference checks by using either static analysis [38] that statically allocates referenced objects in the same scoped memory or by improving the performance of the application through preloading of some classes at compile time [3]. One of these solutions was introduced by Corsaro et al., [38] who improved the implementation of the single parent rule algorithm (a scoped area has exactly zero or one parent) and the reference checks algorithm by using different data structures that make the necessary runtime checks in constant, rather than linear time. In their proposed solution, checking the validity of references does not require the whole scope stack to be scanned but rather it uses an additional data structure to maintain ancestor information for each scope and creates a parenthood tree that represents the scoped memory model of the application with depth value for each scoped memory. The algorithm checks this information to help justify the legality of the references. They implemented their new approach in jRate (an open source RTSJ implementation) and tested its performance by using RTJPerf benchmarks. Results showed that their proposed algorithms gave a constant time overhead regardless of the depth of the scope stack. A more compact and faster access check was introduced by [39] through a subtype test algorithm to provide constant-time RTSJ memory access checks; a write barrier was needed to modify the virtual machine to achieve constant time checks.

Another solution was presented by Higuera-Toledano [40,41] who proposed changing the single parent assignment rule logic. When scoped memory areas are created, their parents are specified at the time of creation and not at the time they are ‘used’ by schedulable objects. They also allowed (in their proposed algorithm) bi-directional references at the cost of longer lifetimes for scoped memory areas. Their new algorithm still however needs to be evaluated after implementing it in the Java virtual machine. In other work, Higuera-Toledano [41] suggested a new algorithm to allow cyclic references among scoped memory areas by replacing the single parent rule relationship with a bit-map table. For each scope in the system, information about which scoped memory areas should be collected is saved in a bit-map table. According to this information, a scoped memory area will not be collected until two conditions are satisfied: first, the scope reference count has fallen to zero and second, there is no ‘collection before’ relationship in the bit-map table for that scope. However, this technique increases scoped memory area lifetimes and produces an overhead on the execution time provided by extra checks.

3.2. Space overheads

Objects created in scoped memory areas cannot be de-allocated individually—the whole scope will be de-allocated when no active threads run inside that scope [12]. Therefore, defining similar object lifetimes and assigning them into associated scoped memory areas is important for saving memory space and reducing the number of dead objects waiting for all objects in the same scope to die. That said, allocating objects in different scoped memory areas manually according to their lifetimes is a complex task for developers, since it requires knowledge of the lifetimes of all objects in the application; that becomes more difficult when the application has a large number of different object types. Different approaches have been developed to identify object lifetimes and their associated scoped memory areas in Java. All current approaches in the literature have investigated scoped memory allocation in sequential programs only and they do not cover multithreaded applications and the sharing of objects among many threads. For instance, Deters and Cytron [34,42] present an algorithm based on dynamic analysis and object referencing behaviour that satisfies RTSJ referencing rules. One scope is assigned to each method in the application—a method call stack is created when a method A calls method B and method B calls method C. The call stack of the method A will follow from bottom to top the following sequence: A, B and C. Objects created in a method A, for instance, might become collectable when method C finishes executing its code—those objects will be de-allocated when method C terminates. The algorithm was implemented on Sun’s JVM version 1.1.8 and benchmarks from Java SPEC suite were used to measure the lifetime of objects. Results showed that many objects do not become collectable for a long time due to the reference rule constraints of the RTSJ. These state that objects that reference other objects should reside in the same memory area to avoid reference violations between memory areas. However, in general, using dynamic traces fails to cover all program behaviours when there is a possibility of applying different sets of inputs. Dynamic analysis results change according to the data set inputs and therefore different behaviours of the application arise. Their approach produced too many regions and needs to consider multi-threading behaviour of real-time applications.

Kwon and Wellings [18] proposed an approach for building a new memory model to map one memory area for each method. In their approach, memory areas cannot be multi-threaded. If each method has one scoped memory, the application will have excessive numbers of scoped memory areas (when there are many methods). Consequently, that increases the execution time of the application as reported by Hamza and Counsell [37]. A semi-automated, static analysis tool was developed by Salagnac et al., [27] to allow a compiler to determine object lifetimes based on a hypothesis which states that connected objects are likely to have similar lifetimes. An allocation policy was developed to automatically allocate objects into different regions in memory at runtime. The static algorithm computed approximations to the connectivity of heap objects. A static analysis tool gave feedback to the developer about the areas of code where objects (or classes) leaked so that they could improve or amend their code. The study did not use one of the RTSJ implementations but ran experiments
on the JITS architecture providing J2SE compliant Java API and virtual machine. They evaluated their approach using JOlden benchmarks and measured memory occupancy during two executions, one with GC and the second with regions.

Results showed that most of the benchmark’s applications used less heap space when using regions as opposed to garbage collection. On the other hand, some of the applications suffered from memory leaks and showed that garbage collection out-performed regions in terms of memory space since static analysis did not give precise information about application behaviour in general. Borg and Wellings [43] also investigated how time and space overheads of the region-based memory model could be reduced when information on region lifetimes was available to the application at runtime. The conclusion was that the more information obtained about program semantics and flow, the less time and space overhead occurred. They considered region lifetimes to be expressed in the application instead of an object graph but this was only possible if the information was implicitly observable in the application, e.g., task flow in a control system.

All current approaches that have tried to allocate objects into regions/scoped memory areas still suffer from memory leaks since static analyses often give an over approximation to object lifetimes. All current approaches in the literature also fail to consider object allocation in multithreaded applications.

### 3.3. Development complexity

#### 3.3.1. Assisting tools

Using scoped memory management complicates the development of applications in real-time Java [15]. The developer needs to be aware of memory concepts and object allocation to ensure memory safety and avoid runtime errors caused by illegal references between memory areas; specifying memory requirements during the execution of the application is a non-trivial task [44] and can be made simpler/less onerous through the use of tools. Garbervetsky et al., [44] proposed a prototype model consisting of many tools for (a) specifying required region sizes (b) measuring the memory requirement of the source code and (c) transforming the Java code into region-based code. Static analysis was also used to capture information in object lifetimes. They evaluated their prototype on two real-time benchmarks, namely CDx and a Banking case study to show how this chain of tools helped developers in managing memory for different Java virtual machines. For the CDx benchmarks, 5 regions were created and for the Banking case study, 18 regions were created. The number of regions in the transformed code was equal to the number of methods that included a new statement (creation sites). Object lifetimes were identified by using static analysis that defined both objects created in the method and those that were either still alive or still to be collected after the method had finished execution. However, their approach still requires some development to measure performance of the region-based code and comparison with the GC-based code. Currently, their approach only works with simple data structures such as arrays and integers and needs to be developed to handle more complex data structures and specific programming aspects such as recursive methods. Allocation made by native methods also needs to be considered in the future (native methods are chunks of code written by other programming languages such as C to be imported into Java programs [45]).

#### 3.3.2. Separating memory concern from program logic

Simplifying the development process through the separation of memory concerns from program logic has been considered a new research direction in region-based/scoped memory management [46,43]. Ideally, the onus of memory management should be devolved as far as possible to the system rather than the developer. Andreae, et al., [46] introduced the ‘Scoped Types and Aspects for Real-Time Systems (STARS)’ model to reduce the burden on developers through the use of scoped types and aspects. Scoped types are based on simple Java concepts (packages, classes, and objects) and give programmers a clear model of their programs’ memory use by creating packages that group classes allocated into one scope. Each package equates to one scope. The main package is the immortal package that has sub-packages to model nested scoped memory areas. Scoped types ensure that the allocation context of any object is obvious from the program text. Developers have to decide on the packaging structure according to the functionality of the application and class coupling. Aspect-oriented programming was used to separate real-time and memory behaviour of the application from its functional aspects (the application logic). After the program had been statically verified, aspects weaved necessary elements of the RTSJ API into the system to define scoped entering using the declarative specification of scoped types. In their approach, reference checks between scoped memory areas were avoided at runtime due to checks on the scoped type system at compile time. These checks ensure that allocating objects in scoped memory areas conforms to the hierarchical structure of the application. They evaluated their prototype model by implementing the STARS in the OVM framework, a tool that assists in creating real-time Java virtual machines [4]. They measured the performance of three versions of the CDx benchmark: (a) with an RTSJ version, (b) with a real-time garbage collection version and, (c) with the STARS version. Results showed that STARS worked 28% faster than programs run on RTSJ or Java with real-time garbage collection. However, the approach required modification of the virtual machine to add functionality provided by scoped types and aspects. On the other hand, the approach did not manipulate array types and required involvement of the developer to decide on the package names and structures in the nesting of memory as well as definition of classes belonging to a specific scope.

A more abstract level approach to STARS is the ownership types by Boyapati, et al. [47]. Each object owns other objects and references to objects are only allowed through their owners. Such an approach guarantees the safety of scoped memory area references by implementing hierarchical regions in ownership types. The ownership relationship between objects is defined
by the developer and is used as criteria for grouping objects into scoped memory areas instead of using object lifetimes. The ownership types still needed some changes to the Java syntax and explicit type annotations [46]. Moreover, their approach exposed programming overheads as the evaluation results showed more lines of code were added to micro-benchmarks used in the evaluation. Zhao et al., [48], defined implicit ownership rather than explicit ownership. The purpose was to decrease the burden on the developer in assigning explicit parameters to classes to define ownership or region information in the program. The allocation contexts of the classes in implicit ownership are defined by their position in the nested class definition hierarchy which, in turn, shapes their instances' position in the dynamic nested scoped memory areas. They presented 'ScopeJ', a simple multi-threaded object calculus with scoped memory management, supported by a type system that ensured safety of object de-allocation. They applied a 'handoff' pattern to transfer data between sibling scoped memory areas without the need to use a copying objects mechanism. Temporary references should be released at an appropriate time to avoid dangling references. The goal of ScopeJ was to offer an alternative to the memory model of the RTSJ.

3.3.3. Design patterns and components

Design patterns can be defined as solutions to commonly-encountered design problems and have been introduced to simplify and solve programming issues related to scoped memory management and real-time threads [49–51]. In theory, application of design patterns to any sphere of software development should result in code that is efficient and highly maintainable. A patterns catalogue was introduced by [49] that included programming designs to solve scoped memory management issues such as:

- Scoped Memory Entry per Real-Time Thread: in this pattern, each real-time thread runs in one scoped memory to avoid interference with the garbage collection that runs only in the heap. However, the pattern does not allow sharing data between threads. If there is data that has a longer lifetime than its specified thread, then this data should be copied from the current scoped memory to either immortal memory or to the heap. If data is copied onto the heap, it will be subject to garbage collection. On the other hand, if data is copied into immortal memory it will remain there indefinitely and consequently, immortal memory size will increase.
- Factory Pattern with Memory Area: A Factory pattern is used when there is a need to create different objects implementing different interfaces, without the need to reveal the implementation class. The Factory class should be placed in immortal memory since it is a singleton (the instantiation of a class is only to one object). When using a Factory pattern with scoped memory areas, each object creation method within the Factory has a memory area parameter which defines where to create the object. In this case, the immortal memory area will be the parent of all created scoped memory areas and therefore the Factory pattern avoids violation of the single parent rule.
- Memory Pools introduced in [52] reduce the footprint of immortal memory by using a pool of already created objects from a specific class. When the application needs to create a new object it will ask the pool to release an unused object. When the application finishes using this object, it will be returned to the pool and made usable for subsequent use. Although this pattern is a way of recycling objects in immortal memory, it has disadvantages. First, it is a manual de-allocation approach where each pool of fixed number of objects can be created only for a specific class. Second, it may cause a memory leak since it reserves memory for a pre-allocated fixed number of objects which may not all be used by the application.
- Memory Blocks overcome the problem of having a pool of fixed number of objects of a specific class. It uses a block of bytes as a unit to store an object that could be instantiated from a different class. When the object is allocated into immortal memory it is serialized in the block; when the object is no longer used it will be de-serialized from it. When de-serializing finishes, the block will be available for further allocation. However, this method is a low-level programming technique and it has costs in terms of serializing, de-serializing, and input/output operations.

Some of the introduced design patterns are already included in [53] but they have been updated to work with RTSJ rules. For example, Meersman et al., [54] gives guidelines for implementing Singleton, Factory, and Leader-Follow patterns for RTSJ applications. The Singleton instance should be allocated in immortal memory to make all threads access it. The Leader-Follow pattern is used to manage concurrent requests to a server and give different threads different priorities when they are activated; all threads are NoHeapRealtimeThreads and will be allocated in one scoped memory. Moreover, each of these threads is associated with another scoped memory to execute code that handles specific events. The Memory Tunnel is a new pattern that enables different schedulable objects running in different scoped memory areas to communicate with each other; the 'tunnel' is a temporary memory queue that should be allocated into a non-scoped memory area. The Memory Tunnel requires deep copying of objects; for example, if real-time thread A wants to pass an object to another real-time thread B, then thread A copies the object into the tunnel memory. The real-time thread B will retrieve that object from the tunnel memory by copying it to its scoped memory. The tunnel queue must be allocated either in the heap or in immortal memory and both have strict referencing rules in RTSJ. The Handle Exceptions Locally pattern is a new pattern which ensures when exceptions are raised, they are executed in the same memory area where they have been raised (or in one of current memory area's ancestors to avoid reference violation errors).

More design patterns are also introduced by [6].

- The Scoped Run Loop Pattern: frees memory space allocated for temporary objects by the loop code and will not be used for the next iteration of the loop. Hence this pattern will reclaim objects each time the loop finishes its iteration. This
The efficiency of memory management is increased through the use of design patterns that allocate long-lived components into parent scoped memory areas and short-lived components into child scoped memory areas according to their lifetimes. Each component has a scoped memory, and a hierarchy of scoped memory areas is created to ensure safety for distributed systems. There is no heap memory used in this architecture, and the model consists of various components.

RTZen is an available Real-Time Java Object Request Broker (ORB) that is highly predictable, real-time Java middleware for distributed systems. It is designed to hide the complexities of RTSJ for distributed systems. There is no heap memory used in this architecture, and the model consists of various components. Each component is associated with a scoped memory, and a hierarchy of scoped memory areas is created to ensure safety of reference rules. Since the lifetimes of the components are explicit in the application, nesting scoped memory areas were used to allocate long-lived components into parent scoped memory areas and short-lived components into child scoped memory areas. Scoped memory exists on the server and client side, and design patterns are implemented in middleware to increase the efficiency of memory management. The design patterns used are:

- Separation of Creation and Initialization.
- Cross-Scope Invocation: to traverse the scoped memory areas hierarchy in order to pass data through a scoped memory that is a common ancestor of both objects (allocated into different scoped memory areas).
<table>
<thead>
<tr>
<th>RTSJ-specific patterns</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scoped memory entry per real-time thread</td>
<td>Benowitz and Niessner [49]</td>
</tr>
<tr>
<td>Factory Pattern with Memory Area</td>
<td></td>
</tr>
<tr>
<td>Memory blocks</td>
<td></td>
</tr>
<tr>
<td>Memory pools</td>
<td>Benowitz and Niessner [49]</td>
</tr>
<tr>
<td>Dibble [52]</td>
<td></td>
</tr>
<tr>
<td>Singleton, factory, and leader-follow patterns</td>
<td></td>
</tr>
<tr>
<td>Memory tunnel</td>
<td>Meersman et al., [54]</td>
</tr>
<tr>
<td>Handle exceptions locally</td>
<td></td>
</tr>
<tr>
<td>Scoped run loop pattern</td>
<td></td>
</tr>
<tr>
<td>Encapsulated method pattern</td>
<td>Pizlo [6]</td>
</tr>
<tr>
<td>Multi-scoped object pattern</td>
<td></td>
</tr>
<tr>
<td>Portal object idioms</td>
<td></td>
</tr>
<tr>
<td>Wedge thread pattern</td>
<td></td>
</tr>
<tr>
<td>Handoff pattern</td>
<td></td>
</tr>
<tr>
<td>Scope pinning</td>
<td>Dibble and Wellsings [55]</td>
</tr>
<tr>
<td>The JSR-302 safety critical Java specification (SCJ)</td>
<td>Henties et al., [56]</td>
</tr>
<tr>
<td>Bogholm et al., [57]</td>
<td></td>
</tr>
<tr>
<td>Component-based software engineering (CBSE)</td>
<td>Etienne et al., [58]</td>
</tr>
<tr>
<td>Component model</td>
<td>Plsek et al., [59]</td>
</tr>
<tr>
<td>Separation of creation and initialization</td>
<td></td>
</tr>
<tr>
<td>Cross-scope invocation</td>
<td>Potanin et al., [60]</td>
</tr>
<tr>
<td>Immortal exception pattern</td>
<td>Raman et al., [61]</td>
</tr>
<tr>
<td>Immortal facade</td>
<td></td>
</tr>
<tr>
<td>An extended portal pattern</td>
<td>Pablo et al., [63]</td>
</tr>
</tbody>
</table>

- **Immortal Exception Pattern**: A schedulable object running inside a scoped memory may raise an exception according to a runtime error and the exception handler may need to access and allocate objects in a different scoped memory area rather than the local scoped memory where it was raised. Therefore, to avoid violating RTSJ referencing rules among scoped memory areas, exception handler objects will be allocated in immortal memory where all objects, wherever they reside, can hold references to objects in immortal memory. Exception handler objects allocated in immortal memory will be reused for possible allocation by later exceptions handlers.

- **Immortal Facade**: Is a pattern that hides the complexity of scoped memory area hierarchies and simplifies the maintenance of large applications by encapsulating the logic that handles cross-scope invocation.

A runtime debugging tool IsoLeak was developed in [62] to visualize scoped hierarchies and find potential memory leaks by defining transient scoped memory areas; however, how the tool defines leaks is not obvious. RTZen was predictable compared to other Java applications that did not use RTSJ. That said, the memory consumption was not specified in their experiments. An Extended Portal Pattern was proposed by [63] to enable referencing portal objects from outside its current scope. However, this approach needed to modify the virtual machine; it also added extra overheads since it forced a thread that needed to reference the portal object to enter the creation context of the portal object itself (which might include nested scoped memory areas).

The three techniques discussed (i.e., software tools, separation of memory concerns from program logic and patterns) are three research directions that show promise in addressing the overheads and, more particularly, the complexity that arises when considering the use of scoped memory management. While the benefits of scoped memory management are relatively clear, the process of memory allocation in the same context is far from trivial.

A list of the RTSJ-design patterns is summarized in Table 2.

### 3.4. Allocation time

Corsaro and Schmidt [64] compared two RTSJ implementations of Timesys and jRate. They used an open-source benchmarking suite called RTJPerf to apply their tests. Their experimental results showed that scoped memory average allocation times (the time needed to allocate an array of bytes that comprise the object) were linear with allocated object sizes in Timesys implementation, while in jRate the allocation times were independent of the allocated object sizes. The same authors [65] extended their work to measure the creation time, entering time, and exiting time of the scoped memory area with respect of scoped memory size. Again, Timesys and jRate RTSJ implementations were studied. Results showed
that creation time relied on the scope size for both implementations. On the other hand, entering time of a scoped memory area in TimeSys implementation varied slightly with changing the scoped memory size (from 4 kbytes to 1 Mbytes), while in a jRate implementation, the entering time of a scoped memory is more dependent on the size of the scoped memory area. Exiting time however did not show any correlation with scoped memory size for both implementations. In another approach by Enery et al., [66] two different implementations of the RTSJ were compared, namely Jamaica VM from Aicas and Sun’s RTSJ 1.0.0. Their study analysed memory allocation, thread management, synchronization and asynchronous event handling. Results showed that the creation times for scoped memory (the time required for a scoped memory object to be declared and initialized) were again linear with scoped memory sizes. Object allocation times were also linear with object sizes. Recent work by Schommer et al., [67] evaluated the Sun RTS2.1 from different perspectives; the relationship between allocation time and object size allocated into memory areas was explored—the relationship was again shown to be linear. They concluded that allocation to immortal memory seemed in general to take longer than allocation to both scoped memory types (LTMemory and VTMemory).

4. Evaluating scoped memory model in RTSJ

Table 3 shows a list of notable benchmarks used in evaluating real-time Java implementations. In this section, we only discuss scoped memory features that the benchmarks evaluated. For example, to measure the memory occupancy during execution of different memory models, JOlden [27] was used to compare heap space growth when regions are created using static analysis. JOlden benchmarks are not real-time applications but they have typical Java programming patterns such as (polymorphism, recursion, use of dynamic memory) which must be supported in a Java real-time environment. Results in [27] showed that most of the benchmark applications used less heap space when using regions rather than garbage collection. However, some of the benchmark’s applications such as Voronoi, showed that garbage collection outperformed regions in terms of memory space. This in turn showed that static analysis did not always give precise information about object lifetimes. Similar results were obtained in [34] where significant free space was saved in some of the Java Olden benchmarks (such as power and tsp benchmarks) when regions were used. However, for bh, health, and Voronoi benchmarks, the GC system was better in terms of memory savings and that in turn demonstrated that static analysis had drawbacks. JOlden benchmarks are available on www.ali.cs.umass.edu/DaCapo/benchmarks.html.

<table>
<thead>
<tr>
<th>Table 3 Benchmarks to evaluate scoped memory in RTSJ applications.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notable benchmarks used in evaluating real-time Java implementations.</td>
</tr>
<tr>
<td>Benchmark</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>JOlden [27]</td>
</tr>
<tr>
<td>CDx</td>
</tr>
<tr>
<td>RTJPerf</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>JScoper</td>
</tr>
<tr>
<td>Two micro-benchmarks</td>
</tr>
<tr>
<td>(Array and Tree), two scientific computations (Water and Barnes), several components of an image recognition pipeline (load, cross, threshold, hysteresis, and thinning), and several simple servers (http, game, and phone, a database backed information server).</td>
</tr>
<tr>
<td>Java SPEC suite [70]</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
RTJPerf [38,64] is an open-source benchmarking suite used to measure different criteria of real-time Java systems and to apply different tests such as Timer test, Threads scheduling tests and Asynchronous Event Handler Dispatch Delay tests. In [38] RTJPerf was used to evaluate the implementation of the single parent rule algorithm and the memory area reference checks algorithm in jRate. Results showed that their proposed algorithms provided constant time overheads regardless of the depth of the scope stack. In [64] RTJPerf was used to evaluate two RTSj implementations of Timesys and jRate. Experimental results showed that scoped memory average allocation times were different in both implementations. For example, allocation times were linear with allocated object sizes in Timesys while in jRate the allocation times did not show any relation to allocated object sizes. In [65] the work was extended to measure creation time, entering time, and exiting time of the scoped memory area with respect to scoped memory size for Timesys and jRate. The RTJPerf benchmark was used and results showed that scoped memory creation time relied on the scope size for both implementations. On the other hand, the entering time of a scoped memory area showed different behaviour with respect to different scoped memory sizes in both implementations. For instance, in the TimeSys implementation there was a slight impact on the entering time when scoped memory size was changed but there was a more significant impact observed on jRate implementation. Exiting time however did not show any relation to the scoped memory size for both implementations. RTJPerf is a promising benchmark to test new, real-time Java virtual machines and measure scoped memory performance. The RTJPerf can be obtained freely at http://jrate.sourceforge.net/Download.php.

The CDx benchmark [68] is an open-source, real-time benchmark available and was used to evaluate the performance of applications that used scoped memory compared with the same version of applications that used real-time garbage collection. It included one periodic NoHeapRealtimeThread which implemented aircraft collision detection based on simulated radar frames. The input was a complex simulation involving over 200 aircraft. In [14] they recorded the latency of processing one input frame when real-time garbage collection and a scoped memory management model were used. The results showed that scoped memory experienced better performance than real-time garbage collection. The OVM virtual machine was used in their study. In [44] CDx was used to implement a transformation algorithm from plain Java code to a region-based Java code and five regions were created. In [46] CDx was used to evaluate their STARS approach (the scoped types and aspects for real-time Java) implementation in an OVM virtual machine. Results showed that STARS worked 28% faster than programs run on RTSJ or Java with real-time garbage collection since reference checks were achieved statically. The CDx can be downloaded from http://adam.lille.inria.fr/soleil/rcd/.

The Java SPEC suite was used in [42] to implement automated discovery of scoped memory regions for real-time Java based on a dynamic, trace-based analysis which observed object lifetimes and object referencing behaviour. Each method is instrumented with a region memory creation statement. An optimum scoped allocation algorithm was developed to allocate objects into the best stack frame (stack of pushed scoped memory area). The Java SPEC suite applications used were raytrace: renders an image, javac: the Java compiler from Sun’s JDK 1.0.2, mpegaudio: a computational benchmark that performs compression on sound files, and jess: an expert-system shell application that solves a puzzle in logic. Results showed that too many regions were created due to many creation sites (827 to 1239) included in each benchmark. The benchmarks comprised a large number of objects (raytrace has 559,287 objects)—a feature that makes it a reasonable example to study. The Java SPEC suite can be obtained from www.spec.org/benchmarks.html.

In [47,36], a variety of benchmarks were used to measure the overhead of heap checks and access checks after implementing region creation algorithm. These benchmarks include Barnes, a hierarchical N-body solver, and Water, which simulates water molecules in the liquid state. These benchmarks allocated all objects in the heap. Two synthetic benchmarks Tree and Array use object field assignment heavily. These benchmarks were designed to obtain the maximum possible benefit from heap and access check elimination. They implemented the real-time Java memory extensions in the MIT Flex compiler infrastructure. Flex is an ahead-of-time compiler for Java that generates both native code and C; it can use a variety of garbage collectors. Results show that reference checks add significant overhead for all benchmarks. However, using scoped memories rather than garbage collection improved the performance of Barnes and Water benchmarks from an execution time perspective.

The JScoper tool, an Eclipse plug-in is presented in [69] as a tool to transform standard Java applications into RTSJ-like applications with scoped memory management. The scoped memory areas creation approach is based on the same approach presented in [35] where object lifetimes are identified by using the call graph of available methods that include object creation sites. The tool enables the developer to visualize the transformation process, to create additional scoped memory areas and to delete or to edit scoped memory areas. However, JScoper needs to be compatible with RTSJ applications. Moreover, its debugging approach for the memory model is recommended as a topic for future work [69], such as visualization of both object lifetimes and active scoped memory areas, scope rules violation, and memory consumption of the scoped memory areas at runtime. JScoper can be downloaded from http://dependex.dc.uba.ar/jscoper/download.html.

Kalibera et al., [71] emphasize the shortage of real-world case studies and the need for tools and benchmarks for real-time applications. To verify memory concerns of the real-time application, tools and benchmarks should provide the following.

- Exception verifications: to ensure the absence of uncaught exceptions such as OutOfMemoryError exception, StackOverflowError exception, ScopedCycleException,
- Analysing memory requirements to define the maximum size each scope requires when different threads are running at the same time—a maximum bound for immortal memory is needed to avoid out of memory runtime errors.
A number of conclusions can be made from the preceding analysis of scope-based benchmarks. First, there is no generally and widely accepted set of benchmarks for evaluation of scopes, which is, in effect, an impediment to progress in the area. Until a generally accepted set of benchmarks evolve, evaluating the efficacy of scoped memory will continue to remain problematic. Second, in common with many empirical evaluations and studies of software, only limited attempts have been made to establish that set of benchmarks. Until a body of evidence has been compiled, that will remain the case. Finally, it is difficult to compare studies if they use disjoint sets of benchmarks; even if those benchmarks are similar, the value and effect of any comparison process can be compromised by minor differences.

5. Potential research directions

Through analysis in this survey, many important and open research questions on using scoped memory management model in real-time Java emerge.

First, there is no precise and simple way to find out the lifetimes of objects to help developers in grouping objects into specific scoped memory areas. The research in this area can benefit from the research undertaken into finding similar lifetimes of objects in non-RTSJ implementations [72]. For example, connected objects (objects that directly or indirectly call other objects methods or modify status of each other) should reside in one scoped memory as there is a correlation between connected objects and their lifetimes. On the other hand, unconnected objects should, in theory, be allocated into one memory area (i.e., immortal memory) since the lifetime of objects is largely unknown [73]. Allocating objects into immortal memory keeps objects alive until the application terminates, even though some objects in immortal memory die after a period of time with the consequent memory leak. Therefore, finding an algorithm to optimize allocation of unconnected objects is crucial to reducing memory leaks. New allocation algorithms should be developed to accurately predict similar object lifetimes in RTSJ. Criteria should be developed for grouping objects into regions/scoped memory areas to help the developer allocate objects into different scoped memory areas and decrease the impact of the memory leak caused by different lifetimes of objects.

Second, the shortage of real-time case studies limits research in finding optimized and precise criteria for allocating objects. Consequently, new real-time benchmarks for RTSJ applications should be provided. This emphasizes the necessity of having scoped memory areas created within these benchmarks (with a non-trivial allocation rate of objects over a period of time). Having these new benchmarks should enable testing different implementation of RTSJ to measure the memory consumption and execution time overheads.

Third, tools to implement the object allocation criteria and to simplify the development process are required. These tools could use static or dynamic analysis to allocate objects into different scoped memory areas; at the same time, it could verify memory requirements and measures the allocation overheads of scoped memory areas. Real-time GUI tools that provide memory visualization and analyses of memory consumption throughout the execution of the application as well as showing memory leaks are also required. Tools should enable the implementation of different scoped memory layouts according to different criteria. Moreover, the developer should be able to re-allocate objects according to memory consumption through comparison of multiple scoped memory layouts. The memory leak in this case can be eliminated.

The preceding analysis and discussion has highlighted a number of open issues in the field of scoped memory; it has also highlighted certain strengths and weaknesses in current approaches to the same area. As a summary of analytical discussions presented in this survey, a set of possible research questions is therefore proposed. Each question may represent a research study in its own right.

- How can we construct a real-time application with scoped memory model in simplified way? This will help the developer decide on the number of scopes and, equally relevant, which objects/threads to be allocated to these scopes (c.f., Sections 3.2, 3.3.2 and 3.3.3).
- Can the implementation of different scoped memory design models be automated? A tool assistant is necessary to convert Java code into scoped memory based RTSJ code, visualize objects allocations inside these scopes, measure the consumption over time and catch possible memory leaks (c.f., Section 3.3.1).
- How effective is scoped memory if it is applied to industrial real-time applications written in Java? This needs a thorough evaluation of the scoped memory model against a garbage collection model in these applications using benchmarks (c.f., Section 4).
- Is there any feasible method to improve reference semantics and the single parent rule implementation presented in RTSJ? Is it possible to allow references between non-sibling, scoped memory areas? This would simplify the use of scoped memory and break some of its design constraints (c.f., Section 3.1).
- Is it possible to create an abstract scoped memory model that can be adapted to many real-time applications? Can some scoped memory designs be adapted to different application with only minor changes? (c.f., Sections 3.3.2 and 3.3.3).
- Will a scoped memory approach survive or be replaced by more deterministic, real-time garbage collection techniques? As germane, how far has the current real-time garbage collection met the current state-of-the-art real-time application requirements?

6. Conclusions

This study of the state-of-the-art in RTSJ memory management highlights what we feel are important issues in scoped memory management for real-time Java. Research in this area has adopted many approaches to develop safety critical,
real-time systems. However, many drawbacks using this model still exist such as time overheads related to reference checks, space overheads due to allocating long lifetimes object in the same scoped memory with short lived objects and complexity of development. This survey discussed current approaches and methods to enhance scoped memory management in RTSJ. Most of the research in RTSJ scoped memory has focused on two important issues. First, decreasing the impact of reference checks and second, converting the application into a component-based application. A set of the most popular benchmarks in the area was introduced and illustrated the shortage of tools and benchmarks for evaluating different memory approaches.

New research directions were also proposed to guide the research towards different directions such as (a) finding the best allocation strategy for developing real-time Java applications using scoped memory mode, (b) new real-time benchmarks that cover more aspects of scoped memory model, and (c) tools to decrease the difficulty of developing real-time Java applications using a scoped memory model. A list of future research questions was also presented as a summary of analytical discussion through this survey of the issues in a scoped memory model.

It is worth noting that a scoped memory management model can be simply utilized and most of the problems will be avoided if it is used for very specific cases in hybrid, real-time systems which consist of soft/hard, real-time and non-real-time application logic. Hard, real-time components comprise the smallest part in most commercial systems [74]. In these systems, heap memory can be used for non-real-time tasks or even for soft, real-time tasks. Scoped memory can be used for hard, real-time tasks. Moreover, using scoped memory in hybrid systems will induce unnecessary complexity that could be avoided. To conclude, excessive usage of scoped memory without rigorous pre-analysis is not a strategy recommended for programmers to adopt.

The use of a real-time garbage collection memory model rather than a scoped memory model is suggested in [14] for applications that can tolerate allocation latency overhead. Although real-time garbage collection has been improved during the last decade, there is still a subset of hard real-time applications that never tolerate the low overheads of real-time garbage collection [20].

Acknowledgements

The authors would like to thank the anonymous reviewers for their incisive and useful comments which have helped to improve the paper significantly.

References


