Optimizing Scoped and Immortal Memory Management in Real-Time Java

A Thesis submitted for the degree of Doctor of Philosophy

By

HAMZA HAMZA

Department of Information Systems and Computing,

Brunel University

October 2013

ABSTRACT

The Real-Time Specification for Java (RTSJ) introduces a new memory management model which avoids interfering with the garbage collection process and achieves better deterministic behaviour. In addition to the heap memory, two types of memory areas are provided - immortal and scoped. The research presented in this Thesis aims to optimize the use of the scoped and immortal memory model in RTSJ applications. Firstly, it provides an empirical study of the impact of scoped memory on execution time and memory consumption with different data objects allocated in scoped memory areas. It highlights different characteristics for the scoped memory model related to one of the RTSJ implementations (SUN RTS 2.2). Secondly, a new RTSJ case study which integrates scoped and immortal memory techniques to apply different memory models is presented. A simulation tool for a real-time Java application is developed which is the first in the literature that shows scoped memory and immortal memory consumption of an RTSJ application over a period of time. The simulation tool helps developers to choose the most appropriate scoped memory model by monitoring memory consumption and application execution time. The simulation demonstrates that a developer is able to compare and choose the most appropriate scoped memory design model that achieves the least memory footprint. Results showed that the memory design model with a higher number of scopes achieved the least memory footprint. However, the number of scopes *per se* does not always indicate a satisfactory memory footprint; choosing the right objects/threads to be allocated into scopes is an important factor to be considered. Recommendations and guidelines for developing RTSJ applications which use a scoped memory model are also provided. Finally, monitoring scoped and immortal memory at runtime may help in catching possible memory leaks. The case study with the simulation tool

developed showed a space overhead incurred by immortal memory. In this research, dynamic code slicing is also employed as a debugging technique to explore constant increases in immortal memory. Two programming design patterns are presented for decreasing immortal memory overheads generated by specific data structures. Experimental results showed a significant decrease in immortal memory consumption at runtime.

AKNOWLEDGMENTS

I would not have been able to provide and complete this Thesis without the sincere support and help of many people. Foremost, I would like to thank my supervisor Dr. Steve Counsell for his patience, motivation, advice, and continuous help and support. He made my PhD journey an excellent experience with his knowledge, kindness, thoughtfulness and encouragement. I would like to dedicate my deep thanks for my mother, for the encouragement she provided, her unlimited patience, prayers and the sacrifices she made to help me after my father passed away, which without, I would not be able to achieve my goals and survive difficult times. I am greatly indebted to my sincere wife Dalia and my daughter Julie, without their hopeful smiles, emotional support, patience, understanding and infinite love I would not have been able to stand during the difficult moments in my PhD. I would like to convey my sincerest gratitude to my uncle Wahid Hamza who has been supportive through all my life stages. His sympathy and understanding were enormous and significantly appreciated. I am very grateful to all people in the Department of Information Systems and Computing at Brunel University who helped me during my study and provided such a friendly and comfortable environment. Last, but by no means least, I would like to thank all my friends and colleagues who have been my other family in the UK.

LIST OF PUBLICATIONS

Journal Papers:

H. Hamza, S. Counsell, Region-Based RTSJ Memory Management: State of the art, Science of Computer Programming, Volume 77, Issue 5, 1 May 2012, Pages 644-659, (http://www.sciencedirect.com/science/article/pii/S0167642312000032)

H. Hamza, S. Counsell, Simulation of safety-critical, real-time Java: A case study of dynamic analysis of scoped memory consumption, Simulation Modelling Practice and Theory, Volume 25, June 2012, Pages 172-189. Cited by: (Singh et al., 2012) (Hamza, Counsell, 2013)

Conference Papers:

Hamza Hamza and Steve Counsell. 2013. Exploiting slicing and patterns for RTSJ immortal memory optimization. In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13). ACM, New York, NY, USA, pp. 159-164

Hamza, H. and Counsell, S., Simulation of a Railway Control System: Dynamic Analysis of Scoped Memory Consumption, the13th International Conference on Modelling and Simulation, Cambridge, Cambridgeshire United Kingdom, 2011. IEEE, pp. 287 - 292

Hamza, H. and Counsell, S., Using scoped memory in RTSJ applications: Dynamic analysis of memory consumption, 37th EUROMICRO Conference on Software Engineering and Advanced Applications, Proceedings of the 37th Euromicro Conference on SEAA 2011. IEEE, pp. 221-225

Hamza, H. and Counsell, S., The impact of varying memory region numbers and nesting on RTSJ execution time. Proceedings of the 3rd International Conference on Computer and Electrical Engineering (ICCEE 2010). pp. p.90-96

H. Hamza, S. Counsell, Improving the performance of scoped memory in RTSJ applications, work-in-progress session, in: SEAA Euromicro Lille, France,September 2010.

DECLARATION

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references. Some of the material presented in this thesis has previously been published as follows:

Chapter 2 is extension to the material previously published in Science of Computer Programming Journal, 2012. The name of the article is Region-Based RTSJ Memory Management: State of the art.

Chapter 3 is an extension on the work presented in 3rd International Conference on Computer and Electrical Engineering (ICCEE 2010). The name of the article is “The impact of varying memory region numbers and nesting on RTSJ execution time”.

Chapter 4 is based on the work published in the Simulation Modelling Practice and Theory Journal, 2012. The name of the article is: “Simulation of safety-critical, real-time Java: A case study of dynamic analysis of scoped memory consumption”.

Chapter 5 is based on the work published in proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13). The name of the article is: “Exploiting slicing and patterns for RTSJ immortal memory optimization”

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organizations.

Signed ……. . . . . (candidate)

Date . . . . . . ….. . . . .

TABLE OF CONTENTS

TABLE OF FIGURES ix

Chapter 1: Introduction 1

1.1 THESIS OVERVIEW 1

1.2 RESEARCH MOTIVATION 3

1.3 RESEARCH AIM AND OBJECTIVES 5

1.4 THESIS CONTRIBUTIONS 6

1.5 THESIS OUTLINE 7

Chapter 2: Literature Review 9

2.1 OVERVIEW 9

2.2 BACKGROUND 12

2.2.1 RTSJ SCOPE PRINCIPLES 14

2.2.2 RTSJ MEMORY MANAGEMENT APIs 16

2.2.3 SCOPED MEMORY REFERENCE SEMANTICS 18

2.2.4 SCOPED MEMORY IN NON-RTS JAVA VIRTUAL MACHINES 20

2.3 CURRENT PROBLEMS AND EXISTING SOLUTIONS 22

2.3.1 TIME OVERHEADS 22

2.3.2 SPACE OVERHEADS 24

2.3.3 DEVELOPMENT COMPLEXITY 27

2.3.3.1 ASSISTING TOOLS 27

2.3.3.2 SEPARATING MEMORY CONCERN FROM PROGRAM LOGIC 28

2.3.3.3 DESIGN PATTERNS AND COMPONENTS 31

2.3.3.4 ALLOCATION TIME 42

2.4 BENCHMARKS TO EVALUATE RTSJ SCOPED MEMORY 43

2.5 POTENTIAL RESEARCH DIRECTIONS 49

2.6 SUMMARY 52

Chapter 3: Empirical Data Using A Scoped Memory Model 54

3.1 OVERVIEW 54

3.2 EMPIRICAL DATA FOR SCOPED MEMORY AREA ALLOCATION 56

3.3 THE IMPACT OF CHANGING SCOPED MEMORY NUMBERS AND NESTING ON EXECUTION TIME. 60

3.3.1 EXPERIMENTAL CODE DESIGN 63

3.3.2 UN-NESTED SCOPED MEMORY AREAS 66

3.3.3 NESTED SCOPED MEMORY AREAS 70

3.4 THE ENTERING/EXITING TIME OVERHEADS OF SCOPED MEMORY AREAS. 75

3.5 SUMMARY 79

Chapter 4: A Case Study of Scoped Memory Consumption 81

4.1 OVERVIEW 81

4.2 SIMULATION MODEL 83

4.2.1 ASSUMPTIONS OF THE SIMULATOR 88

4.2.2 SCOPED MEMORY DESIGN MODELS 90

4.3 EXPERIMENTAL DESIGN 95

4.4 SIMULATION TOOL 98

4.5 SIMULATION ANALYSIS 104

4.6 GUIDELINES FOR USING SCOPED MEMORY IN RTSJ 112

4.7 CONCLUSIONS 114

Chapter 5: Slicing and Patterns for RTSJ Immortal Memory Optimization 115

5.1 OVERVIEW 115

5.2 METHODOLOGY 116

5.3 IMMORTAL MEMORY PATTERNS 122

5.3.1 HASHTABLE READING PATTERN 122

5.3.2 HASHTABLE MODIFYING PATTERN 125

5.4 DISCUSSION 128

5.5 SUMMARY 131

Chapter 6: Conclusions and Future Work 133

6.1 RESEARCH SUMMARY 134

6.2 RESEARCH OBJECTIVES RE-VISITED 137

6.3 SUMMARY OF RESEARCH CONTRIBUTIONS 138

6.4 RESEARCH LIMITATIONS 139
6.5 FUTURE WORK 141

TABLE OF FIGURES

Figure 2.1: A RealTimeThread forms nesting scopes, scope stack is created. 17
Figure 2.2: Scope stack (Dawson, 2007) 36
Figure 3.1: Execution Times of 5/10 scoped memory areas application for different data types (1000 objects example) 59
Figure 3.2: Scoped Memory Consumptions of different data types when 1000 objects are created in 5/10 scoped memory areas application 60
Figure 3.3 Creating objects in un-nested scoped memory areas sample..... 64
Figure 3.4: Execution time for un-nested scoped memory areas 66
Figure 3.5: 5 scoped memory area data (2500 objects) 67
Figure 3.6: 10 scoped memory area data (2500 objects) 68
Figure 3.7: 15 scoped memory area data (2500 objects) 68
Figure 3.8: 20 scoped memory area data (2500 objects) 69
Figure 3.9: 25 scoped memory area data (2500 objects) 69
Figure 3.10: Execution time for nested scoped memory areas 71
Figure 3.11: % in execution time increase (un-nested) scoped memory areas 73
Figure 3.12: % increase in execution time (nested scoped memory areas) 73
Figure 3.13: Differences in execution time (un-nested vs. nested) 74
Figure 3.14: Calculation of entering and exiting times in scoped memory area... 76
Figure 3.15: Entering Scoped Memory Execution Time 77
Figure 3.16: Exiting Scoped Memory Execution Time 78
Figure 4.1: Simulation Model for a Real-Time Java Scoped memory Model 84
Figure 4.2: The main objects and threads in the Simulator 87
Figure 4.3: Simulation GUI element at 140 seconds (Design 1) 100
Figure 4.4: Simulation GUI element at 299 seconds (Design 1) 101
Figure 4.5: Simulation GUI element at 142 seconds (Design 2) 102

Figure 4.6: Simulation GUI element at 300 seconds (Design 2) 103
Figure 4.7: Simulation Console element (Design 1) 104
Figure 4.8: Immortal memory consumption in Designs 1, 2, 3, 4 and 5............ 105
Figure 4.9: Scoped memory consumption in Design 1 107
Figure 4.10: Scoped memory consumptions in Design 2 108
Figure 4.11: Scoped memory consumptions in Design 3 109
Figure 4.12: Scoped memory consumption in Design 4 110
Figure 4.13: Scoped memory consumption in Design 5 111
Figure 5.1: (a) An example program. (b) A slice of the program criterion (10, product). 118
Figure 5.2: The Slicing Methodology. 120
Figure 5.3: Design Pattern 1 (Reading Hashtable Values) 125
Figure 5. 4: Design Pattern 2 (Modifying Hashtable Values)............................ 128
Figure 5.5: Before/After Implementing Design Patterns 1 and 2 131
Figure 5. 6: Before/After Implementing Design Pattern 2................................. 131

LIST OF TABLES

Table 2.1: Assignment rules (Dibble, 2008, Bruno and Bollella, 2009)............... 19

Table 2.2: A list of common RTSJ-design patterns. 40

Table 2.3: Benchmarks to evaluate scoped memory in RTSJ applications 45

Table 3. 1: Execution Time and Memory Consumption for each scoped memory area (Integer and Float) 58

Table 3. 2: Execution Time and Memory Consumption for each scoped memory area (Hashtable and Vector)... 58

Table 3. 3: Summary data for un-nested scoped memory areas 66

Table 3. 4: Summary data for nested scoped memory areas................................ 71

Table 3. 5: Summary Data for Entering/exiting Scoped Memory 78

Table 4. 1: Initial and possible design memory models of the case study 94

Table 4. 2: The simulation platform... 98

Table 4. 3: Summary Data for Immortal consumption 106

Table 4. 4: Summary Data for Scope consumption 111

Table 5.1:. Before/After Implementing Design Patterns 1 and 2....................... 130

Chapter 1: Introduction

1.1 Thesis Overview

Programming languages have different approaches to managing application memory. For example, in Fortran, memory management is static where the location of a variable is statically defined at compile time and fixed at runtime. Other programming languages use dynamic memory management models where data structures can be dynamically defined at runtime. Some of these dynamic memory models are manual memory management models (e.g., C and Pascal) where allocation/de-allocation of objects is handled by the developer. However, manual approaches add more complexity to the application development (Robertz, 2003). The other model of dynamic memory management is ‘automatic’ such as the garbage collection technique employed by the Lisp and Java programming languages (Henriksson, 1998).

Java uses a garbage collection technique to manage memory automatically. The garbage collector interrupts the application on different occasions to reclaim objects no longer in use by the application. However, garbage collection, when running, delays the application and pauses its execution. This is not acceptable in real-time applications that have deterministic behaviour and strict timing requirements (Brosgol and Wellings, 2006). A “Real-time system is a system in which its correctness depends not only on the logical result of the computations it performs but also on time factors” (Stankovic and Ramamritham, 1989). A fault in these systems can cause loss of human life or a significant financial setback (Baker et al., 2006, Dvorak et al., 2004). These faults can occur because of a poor memory model that may cause a system execution delay or a systems’ memory to overflow. A number of examples of

using Java in real-time systems is evident in industry such as the autonomous navigation capabilities of the ScanEagle unmanned aerial vehicle developed by Boeing and Purdue University (Armbruster et al., 2007), a motion control system developed by Robertz et al., (Robertz et al., 2007), and IBM's comprehensive battleship computing environment and commercial real-time trading systems described in Pizlo and Vitek (2008).

New real-time garbage collection algorithms in Java have been proposed and implemented in commercial products for real-time systems (Dawson, 2008), but there are still many research challenges in real-time garbage collection for decreasing pause times and space overheads (Kalibera, 2009) (Plšek, 2009).

The Java Community Process (JCP), founded in 1998 and supported by IBM and Sun Microsystems, proposed the first Java Specification Request as JSR-1 for the real-time specification of Java (RTSJ). RTSJ introduced a new memory model a semi-automatic memory management model, which includes scoped memory and immortal memory. In addition to heap memory, there is only one immortal memory and one or more scoped memory areas in real-time Java applications according to the RTSJ model. Scoped and immortal memory areas are not subject to garbage collection and therefore no delays or interruptions by garbage collection occur. However, developing applications using a scoped and immortal memory management model is a difficult task and has many drawbacks (Higuera-Toledano, 2006, Pizlo and Vitek, 2008). First, it requires additional classes for proper management and possibly application of specific design patterns (Pizlo, 2004). Secondly, 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, the memory model in one application cannot be adapted to other applications. Thirdly, 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 (Kwon and Wellings, 2004, Magato and Hauser, 2005, Borg et al., 2006, Fridtjof, 2006, Pizlo and Vitek, 2006, Chang, 2007, Bacon, 2007).

The aforementioned themes play an important role in the Thesis chapters and contents. The next section summarizes the motivation for conducting this research which leads to the set of stated contributions (Section 1.4).

1.2 Research Motivation

Reviewing the literature of the new memory model in RTSJ, a set of observations motivating the research in this thesis can be made:

1. To evaluate the expressiveness of the new dynamic memory model presented in RTSJ, case studies that include persistent dynamic allocation over period of time are required. However, RTSJ case studies that include scoped and immortal memory use are still very rare.
2. To verify the memory model exceptions at runtime (such as OutOfMemoryError exception) and to monitor immortal memory consumption, the availability of assisting development tools is essential (Kalibera et al., 2010). Region memory profiling (the study of the program behaviour at runtime based on set of inputs to optimize the program code more efficiently by gathering information on the application at runtime (Gabbay and Mendelson, 1997) is promising method of locating and fixing space leaks (Tofte et al.,

2004). Since the developer decides on where the objects will be allocated in scoped and immortal memory, there is a possibility of memory leaks occurring according to misjudgment on the right allocation. Therefore, using dynamic analysis tools which visualize object allocations into scoped memory and measure the consumption over time may help in catching possible memory leaks

3. For safety-critical real-time systems, since rigorous verification of their functionalities, timings and memory consumption is required, simulating these systems before putting them into their real environment is an important practice for eliminating the cost of testing, reducing the risk of failure and ensuring high quality results (Rosenkranz, 2004).
4. Deciding on the number of scoped memory areas, their sizes and which objects to be allocated in these scoped memory areas are left to the developer to design. Consequently, different scoped memory design models can be created according to specific priorities such as a smaller execution time or memory footprint. The optimum criteria to allocate objects/threads in scoped memory areas in a way that leads to minimum consumption space and safe referencing is an open research area. Therefore, providing developers with guidelines to use this model may help to optimize the use and the design of the scope memory model and simplify the development process.
5. The decision that a developer has to make on scoped memory area numbers can have a significant impact on potential application efficiency and execution time. On the other hand, nested scoped memory areas have potential advantages of memory savings, since child memory areas have shorter lifetimes than their parents; the impact this has on application execution time and the inherent

trade-off with those memory savings is an open research question. An empirical study of this memory model that cover different characteristics is required to provide more information about the usage and characteristics of this model; eliminating space overhead is not currently discussed in the literature.

6. Immortal memory space may increase constantly at runtime which may end up as an overflow error. Defining new techniques to debug and eliminate constant increases in immortal memory is a critical task for developers.

All the above issues motivated this research to provide philosophical and practical knowledge of this memory model and to provide solutions that help in developing scoped and immortal memory applications in specific programming situations.

1.3 Research Aim and Objectives

Considering the research motivation discussed in Section 1.2, the aim of this research is thus: *To explore optimization in the context of the scoped and immortal memory of real-time Java applications.*

To fulfill this aim, a number of objectives are necessary:

Objective 1: to describe state of art issues in the use of scoped memory in real-time Java and discuss the current solutions and challenges to generate a set of research questions.

Objective 2: to provide an empirical study on some aspects of the scoped and immortal memory model and its impact on memory space and execution time of the application when different types of objects are allocated. This helps an understanding of different overheads and considering appropriate design of the memory model.

Objective 3: To develop a real-time Java case study which uses a scoped and immortal memory model in a multi-threaded environment where dynamic allocation of objects takes place constantly. Implementing and comparing different scoped memory models provides guideline for creating the appropriate scoped and immortal memory model.

Objective 4: To provide debugging techniques which help in decreasing the overheads of using a scoped and immortal memory model by implementing programming design patterns and evaluating their outcomes.

1.4 Thesis Contributions

The main contributions of this thesis are:

1. A survey of state of art issues of the new memory model introduced by RTSJ; this provided an overview of the problems, challenges, solutions, benchmarks and potential research directions in the scoped and immortal memory model.
2. A detailed study of the impact of using scoped memory on the execution time and memory space of the application when different data types are allocated in scoped memory areas and when different scoped memory numbers and nesting are used. A comparison between entering and exiting times of an active and non-active scoped memory area.
3. Provision of an additional RTSJ case study which integrates scoped and immortal memory techniques to apply different memory models.
4. Development of a simulation tool for a real-time Java application (the first that we know of) which shows scoped memory and immortal memory consumption of an RTSJ application over a period of time. The tool helps

developers to choose the most appropriate scoped memory model by monitoring memory consumption and application execution time.

5. Implementation of a dynamic slicing technique to debug RTSJ code and to define the objects that specifically affect immortal memory increases at runtime.
6. Proposition and validation of two programming design patterns to decrease immortal memory consumption when Hashtable data structures are manipulated inside immortal memory.

1.5 Thesis Outline

The remainder of the Thesis is structured as follows.

Chapter 2 presents a state of art literature review of using scoped memory in real-time Java (RTSJ). An overview of different issues related to the development of applications using a scoped memory model is provided. The benchmarks used to evaluate the implementation of RTSJ scoped memory are also presented and these can help to identify current case studies and their benefits. The chapter emphasizes the need for future benchmarks that verify and demonstrate the functionality of a given scoped memory management model. An overview of all current solutions, approaches and design patterns related to scoped memory applications are presented.

Chapter 3 enriches the empirical study on using a scoped memory model from different aspects in an RTSJ implementation: the Sun Java RTS 2.2. It provides empirical data on allocating different data types into scoped memory areas. Float, Hashtable and Vector data types were tested to measure the execution time and memory consumption for each when created inside scoped memory areas. It also

analyses the impact of changing scoped memory numbers and nesting on execution time. A comparison of the entering and exiting times of an active and non-active scoped memory area at runtime is also presented in this chapter.

Chapter 4 provides a new RTSJ case study, namely a railway control system implemented as a multi-threading system in the SUN RTS 2.0 virtual machine. The case study employs a scoped and immortal memory model to allocate different types of objects. Five possible scoped memory models are discussed. A simulation tool is developed to measure and show scoped and immortal memory consumption of the case study for each memory design model over a period of time along with the execution time of the case study. The tool enables developers to decide on the most appropriate scoped memory model by monitoring memory consumption and application execution time at runtime. Recommendations and guidelines for developing RTSJ applications which use a scoped and immortal memory model are also presented in this chapter.

Chapter 5 proposes a dynamic code slicing approach as a debugging technique to explore constant increases in the immortal memory of the case study. Objects and methods which cause immortal memory to increase constantly are defined. Two programming design patterns are presented for decreasing immortal memory overheads generated by specific data structures. Runtime data is also provided which consolidates the validity and importance of the approach to decreasing immortal memory consumption at runtime.

Chapter 6 summarizes the Thesis main contributions and findings. Finally, the chapter describes the limitations of this study and opportunities for future work.

Chapter 2: Literature Review

2.1 Overview

A real-time system is any system in which responding to external changes in a specific period of time is as important as satisfying the system's functionalities (Burns and Wellings, 2001). 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 (Bruno and Bollella, 2009). 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 (Dvorak et al., 2004, Baker et al., 2006). These faults can be the result of many factors such as miscalculation of deadlines, unexpected power failures, or an inadequately designed memory model which may delay the response time and cause a system's memory to overflow. Therefore, programming these systems requires precise design and implementation.

Java, as an object oriented programming language introduced by Sun Microsystems in 1995, is widely adopted in many sectors because of its code reliability, portability, maintainability and automatic memory management. Recent studies have showed how Java has increased in popularity against other programming languages such as C, C++ and Ada. Although Java embraces a multi-threading environment, it lacks some of the important characteristics that make it suitable for real-time systems such as non-deterministic timing behaviour due to automatic memory management and an unpredictable threads scheduling order. This has motivated the research since 1996 towards making Java suitable for real-time systems (Higuera-Toledano, 2012, Kelvin, 2012). The Java community Process (JCP), founded in 1998 and supported by IBM

and Sun Microsystems, proposed the first Java Specification Request as JSR-1 for the real-time specification of Java. 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) (Bruno and Bollella, 2009).

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 (Pizlo, 2004). 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 memory model is not subject to garbage collection either through time pauses or the collection of individual objects (Bollella et al., 2000, Dibble, 2008). The concept of RTSJ memory areas is borrowed from the more general concept of memory regions first introduced by Tofte et al., (Tofte and Talpin, 1997). 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 (Nilsen, 2006).

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 (Higuera-Toledano, 2006, Pizlo and Vitek, 2008). 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 (Pizlo, 2004)). Second, since the design of a scoped memory model requires information about the object and thread lifetimes of that application which, in turn, differs from one application to another, the memory model in one application cannot be adapted to other applications. 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 (Kwon and Wellings, 2004, Magato and Hauser, 2005, Borg et al., 2006, Fridtjof, 2006, Pizlo and Vitek, 2006, Bacon, 2007, Chang, 2007); this, 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 (Armbruster et al., 2007), a motion control system developed by Robertz et al., (Robertz et al., 2007), IBM's comprehensive battleship computing environment and commercial real-time trading systems described in (Pizlo and Vitek, 2008) 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.

This chapter presents a detailed description of the state-of-the-art in the RTSJ scoped memory model. An overview is provided which gives a broad understanding of the different issues and highlights existing problems that still 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 chapter is structured as follows: Section 2.2 provides background and introduces the scoped memory management of RTSJ. Current problems using scoped memory in RTSJ and their existing solutions are then introduced in Section 2.3. Section 2.4 describes a set of benchmarks with which to evaluate the implementation of an RTSJ scoped memory model. New research directions and possible solutions to use scoped memory in RTSJ are discussed in Section 2.5. Finally, Section 2.6 concludes the chapter.

2.2 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 which 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 (Robertz, 2003); a 'memory leak' is said to occur when unclaimed dead objects no longer reachable by an application remain in memory for a relatively long time (Jump and McKinley, 2013). The alternative model of dynamic memory management is 'automatic' typified by the garbage collection technique employed by the Lisp and Java programming languages (Henriksson, 1998). 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 (Brosgol and Wellings, 2006). 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 (Kalibera, 2009). 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 (Nakhli et al., 2006, Jones, 2007, Salagnac et al., 2007).

2.2.1 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" (Wilson et al., 1995). Each block in the heap memory contains a number of bytes known as single allocation unit to store the application objects (Kim and Hsu, 2000). 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 never 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 (Bruno and Bollella, 2009).

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. For instance, the real-time garbage collector (RTGC) in Sun RTS 2.0 can be pre-empted 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 (Robertz et al., 2007). 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 process 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 (Bruno and Bollella, 2009). 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 (Auerbach et al., 2007). 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.

2.2.2 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 (Bruno and Bollella, 2009). 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.

Figure 2.1 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.

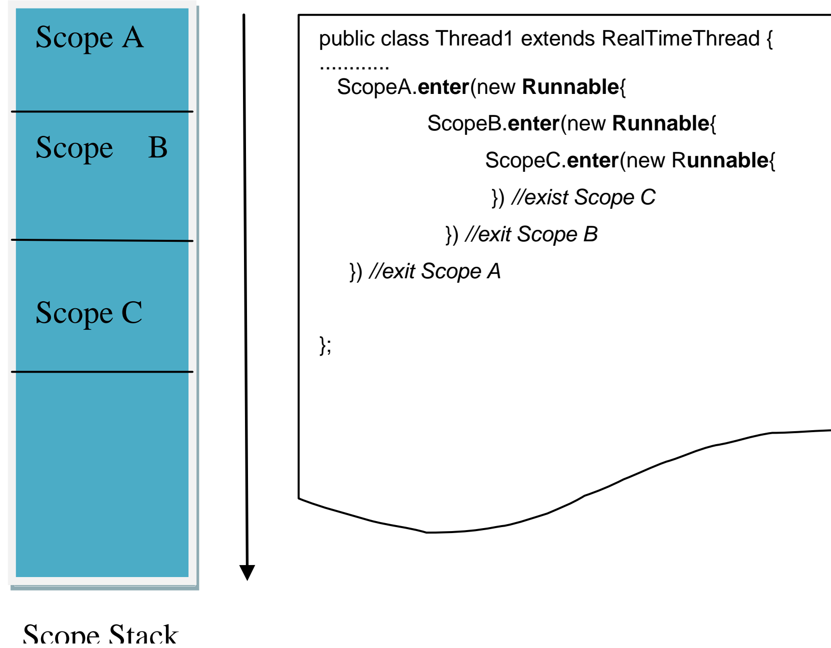

Figure 2.1: A RealTimeThread forms nesting scopes, scope stack is created.

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 (Baker et al., 2006). 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 the real-time thread and memory area class methods to obtain information about a memory scope area:

- getCurrentMemoryArea(): static method which returns the current allocation context.
- getMemoryArea(): non-static method which returns the initial memory area used.

- getMemoryAreaStackDepth(): returns the size of the current schedulable object's scope stack.
- getOuterMemoryArea(index) returns a reference to the memory area at the stack at index given. Stack access is zero-based.
- 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, the executeInArea method can be used to change the current allocation context.
- getReferenceCount(): is used with ScopedMemory class and 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.

2.2.3 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 references 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 (Bruno and Bollella, 2009). Table 2.1 summarizes the assignment rules between memory areas to avoid dangling references at runtime. Local variables are collected automatically when methods exit.

Object Stored In	Can Reference Heap?	Can Reference Immortal?	Can Reference Scope?
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Only if objects reside in the same scoped memory areas or in the outer scoped memory
Local variables	Yes	Yes	Yes

Table 2.1: Assignment rules (Dibble, 2008, Bruno and Bollella, 2009)

2.2.4 Scoped memory in non-RTS 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-RTS Java virtual machines. One of these features is that in 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 (Bøgholm et al.). 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 (Cherem and Rugina, 2004) transformed Java code into region annotation-based code which included the creation, removal and passing of regions as parameters and allocating objects into these regions. All regions were created in heap memory. 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 some benchmarks such as bh, health, and voronoi, the garbage collection version was an improvement over the region-based version in terms of memory saving which is an indication of static analysis drawbacks. Static analysis gives only approximations of object lifetimes and may allocate all objects into only one immortal region and consequently a memory leak occurs (Cherem and Rugina, 2004). Another approach to developing Java virtual machines using scoped memory was that proposed by Garbervetsky et al., (Garbervetsky et al., (2005), 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:

```
// This code is not an 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 (Wilson et al., 1995)), run time analysis was undertaken in (Garbervetsky et al., 2005) 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 and 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 multi-threading and recursion and, in our opinion, requires evaluation on different real-time case studies.

2.3 Current problems and existing solutions

Many problems with using scoped memory management have been described in the literature. For example, Beebee and Rinard (Beebee and Rinard, 2001) 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.

2.3.1 Time overheads

Time overheads result when the virtual machine checks for every assignment between two objects obj1.v1=obj2.v2 allocated into two different scoped memory areas and for every attempt to enter a memory area by a schedulable object to ensure the single parent rule among scoped memory areas. Defoe et al., (2007) provided asymptotic time-complexity analysis of abstract data types such as stack and queue when RTSJ scoped-memory areas and NHRTs (No Heap Real-time Threads) were used . Results concluded that a linear complexity is associated with a scoped memory model and complexity will increase when a nesting scoped memory model is used. However, the authors did not test any RTSJ implementation. In Hamza and Counsell (Hamza and Counsell, 2010), the features of scoped memory in RTSJ implementation SUN RTS 2.0 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. Empirical 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 (Corsaro and Cytron, 2003) which 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 (Bruno and Bollella, 2009). One of these solutions was introduced by Corsaro et al., (Corsaro and Cytron, 2003) 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 did not require the whole scope stack to be scanned but rather to use an additional data structure to maintain ancestor information for each scope; a parenthood tree was created representing 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 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 (Palacz and Vitek, 2003) 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 (Higuera-Toledano, 2008b, Higuera-Toledano, 2008a) 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 needs to be evaluated after implementing it in the Java virtual machine. Higuera-Toledano (Higuera-Toledano, 2008a) 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, in the bit-map table for that scope there is inner-reference (a reference from another scoped memory area). However, this technique increases scoped memory area lifetimes and produces an overhead in terms of the execution time provided by extra checks.

2.3.2 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 (Pizlo and Vitek, 2008). 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; this 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 multi-threaded applications and the sharing of objects among many threads. For instance, Deters and Cytron (Deters and Cytron, 2002) 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 (Kwon and Wellings, 2004) 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 (Hamza and Counsell, 2010). Previous work on garbage collection algorithms by Hirzel, et al., (2003) showed that there was a strong correlation between connectivity and the lifetime characteristics of objects. They introduced a new garbage collection process which allocated objects into partitions based on their connectivity and de-allocated (at each collection) specific partitions using their connectivity information. A semi-automated, static analysis tool was developed by Salagnac et al., (2007) to allow a compiler to determine object lifetimes based on the theory of connected objects correlations with their 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 (Just In Time Scheduling) architecture providing a 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, (2006) 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. On the other hand, all current approaches in the literature fail to consider object allocation in multi-threaded applications.

2.3.3 Development complexity

2.3.3.1 Assisting Tools

Using scoped memory management complicates the development of applications in real-time Java (Magato and Hauser, 2005). 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 (Garbervetsky et al., 2009) and can be made simpler/less onerous through the use of tools. Garbervetsky et al., (2009) 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 allocation sites (program locations that create a new object (Singer et al., 2008)). Object lifetimes were identified by using static analysis which defined both objects created in the method and those that were either still alive or still 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 and multi-threading behaviour. 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 (Liang, 1999)).

2.3.3.2 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 (Borg and Wellings, 2006, Andreae et al., 2007). Ideally, the onus of memory management should be devolved as far as possible to the system rather than the developer. Andreae et al., (2007) 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 program's 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 the 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 (Baker et al., 2006). 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, scope types 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., (2003). 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 (Andreae et al., 2007). 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., (2008), 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 which 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.

2.3.3.3 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 (Benowitz and Niessner, 2003, Bollella et al., 2003, Otani et al., 2007, Alrahmawy and Wellings, 2009). In theory, application of design patterns in any sphere of software development should result in code that is efficient and highly maintainable. A patterns catalogue was introduced by (Benowitz and Niessner, 2003) and 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. The Factory pattern avoids violation of the single parent rule.

- Memory Pools introduced by (Dibble, 2002) 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 unusable 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 (Gamma et al., 1994) but they have been updated to work with RTSJ rules. For example, Meersman et al.,

(2004) 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 that 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 (Pizlo, 2004):

- 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 pattern does not allow referencing from any

code outside the loop and therefore a different pattern should be used (such as the multi-scoped pattern).

- The Encapsulated Method Pattern; this pattern executes a method body in a scoped memory area and this can be used for methods which include newly created objects not to be used after the method finishes its execution. An example is a computational method which uses temporary allocation during its task of calculating a specific formula.
- The Multi-Scoped Object Pattern: is an instance of a class that can be spanned over different scoped memory areas. This occurs when the class creates different object lifetimes and it is important to allocate them into different scoped memory areas according to their lifetimes.
- Portal Object Idioms: portal object is an object created in the scoped memory and can be shared by different threads that enter the scope. The developer has to define the portal object. The downside of this pattern is that threads have to access this scope to modify the portal object. Using this pattern requires synchronization among threads sharing this object.
- The Wedge Thread Pattern: is a thread that enters a scope and does nothing. It is used to make the scope live longer until the specific condition is satisfied. This pattern can be used when a thread modifies a scoped memory's portal object and it needs to exit that scope before another thread enters. It is then necessary to keep the scope alive until the other thread enters and reads or modifies the portal object. This pattern is therefore considered as a method to communicate and pass objects among threads.

- The Handoff Pattern: This pattern is used when two sibling scoped memory areas need to pass objects between each other. One sibling will store the object in the parent scope (the reference is allowed from the child scope to the parent scope); the other sibling scope will then read that object from the same parent scope.

Based on grouping similar lifetime objects perspective, The Lifecycle Memory Managed Periodic Worker Threads pattern was introduced in Dawson (2007) to simplify developing real-time applications using scoped memory, the rule for this pattern is to group similar lifetime objects in one scoped memory. When periodic threads run together to accomplish a specific task, four main categories of object lifetimes can be defined as follows (see Figure 2.2):

- **Retain Forever**: Objects with this lifetime are alive until the application terminates and are accessible to all threads.
- **Retain Thread Group**: Objects with this lifetime will not be reclaimed until all the threads that share these objects have terminated. These objects are accessible only by threads within the group of threads.
- **Retain Thread**: Objects with this lifetime will be created by a specific thread and are not accessible by other threads.
- **Retain Iteration**: Objects with this lifetime are created during the iteration and will not be used outside of the iteration.

The limitation of that approach is that it scarifies the granularity of the memory management model and may consume more space than required; nevertheless, the

developer has to decide in advance which objects will be allocated in which regions according to the four categories mentioned before.

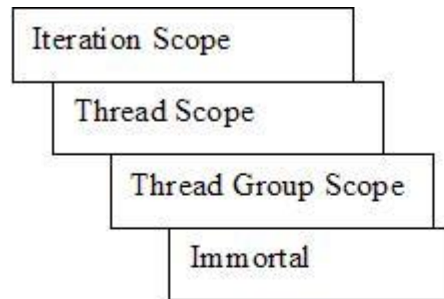

Figure 2.2: Scope stack (Dawson, 2007)

The Real-Time Specification for Java (RTSJ) is the first Community Process' Java Specification Request (JSR-1). After finding some faults in the implementation and according to improvements requested based on the experience of using RTSJ version 1.0.1 and 1.0.2 (developed in 2004 and 2006 sequentially), the Java Community Process' Java proposed the Java Specification Request (JSR 282) as a modified version of RTSJ to introduce RTSJ 1.1 with new promising features. However, the implementation is not yet complete and some alpha versions are available on http://www.timesys.com/java/. One feature of RTSJ 1.1 related to scoped memory usage is the concept of "scope pinning" which replaces the need for wedge-threads and enables the scope to be alive even though there are no schedulable objects running within it (Dibble and Wellings, 2009).

A component model has been introduced by many studies to be implemented in RTSJ as a means of facilitating design, implementation and maintenance (Alrahmawy and Wellings, 2009). A component is "a software entity interacting with its environment via a well-defined interface, making it ready for composition and reuse" (Etienne et al., 2006). Etienne et al., (2006) described the applicability of Component-Based

Software Engineering (CBSE). RTSJ was investigated to increase the abstract level representation of real-time applications. Each component was allocated into one scope to provide flexibility of component management and to ensure reference rules were not violated; this increased execution time of the application, but, on the other hand, did not express the real-time memory concerns separately from the business architecture. RTSJ concerns should be specified at early stages of architectural design to simplify the implementation process (Plsek et al., 2008). The component model proposed in (Plsek et al., 2008) shows different steps of design: a business view of the real functionalities of the application, a memory management view and a thread management view. Assigning scoped memory areas to tasks is left for the developer to decide.

RTZen is a Real-Time Java Object Request Broker (ORB) available on http://doc.ece.uci.edu/rtzen/ (Potanin et al., 2005, Raman et al., 2005b) and is considered as 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).

- 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 exception handlers.

- Immortal Facade: is a pattern which 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 (Raman et al., 2005a) 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, memory consumption was not specified in their experiments. An Extended Portal Pattern was proposed by (Pablo et al., 2006) to enable referencing portal objects from outside its current scope. However, this approach needs to modify the virtual

machine; it also adds extra overheads since it forces a thread that needs 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 RTSJ-design patterns is summarized in Table 2.2.

RTSJ-specific patterns	**Reference**
Scoped Memory Entry per Real-Time Thread	(Benowitz and Niessner, 2003)
Factory Pattern with Memory Area	(Benowitz and Niessner, 2003)
Memory Pools	(Benowitz and Niessner, 2003) (Dibble, 2002)
Memory Blocks	(Benowitz and Niessner, 2003)
Singleton, Factory, and Leader-Follow Patterns	(Meersman et al., 2004)
Memory Tunnel	(Meersman et al., 2004)
Handle Exceptions Locally	(Meersman et al., 2004)
Scoped Run Loop Pattern	(Pizlo, 2004)
Encapsulated Method Pattern	(Pizlo, 2004)
Multi-Scoped Object Pattern	(Pizlo, 2004)
Portal Object Idioms	(Pizlo, 2004)
Wedge Thread Pattern	(Pizlo, 2004)
Handoff Pattern	(Pizlo, 2004)
Scope Pinning	(Dibble and Wellings, 2009)
The JSR-302 Safety Critical Java specification (SCJ)	(Henties et al., 2009) (Bøgholm et al., 2009)
Component-Based Software Engineering (CBSE)	(Etienne et al., 2006)
Component Model	(Plsek et al., 2008)
Separation of Creation and Initialization	(Potanin et al., 2005)

Cross-Scope Invocation Immortal Exception Pattern Immortal Facade	(Raman et al., 2005b)
An Extended Portal Pattern	(Pablo et al., 2006)

Table 2.2: A list of common RTSJ-design patterns.

(Kwon et al., 2002) have proposed a profile for real-time Java for high-integrity real-time systems. The profile adopts architecture with an Initialization Phase and Mission Phase and restricts automatic garbage collection to ensure the predictability of system operation. For safer real-time systems, the JSR-302 Safety Critical Java specification (SCJ) (Henties et al., 2009) is proposed which is based on the Real-Time Specification for Java to provide a safer profile for safety-critical systems. Safety-critical systems are those systems that cannot afford any incorrect or delayed response and therefore need rigorous verification techniques. The SCJ has no heap memory and the scoped memory has been further restricted. An SCJ compliant application consists of one or more missions and a mission may consist of a limited set of schedulable objects such as periodic event handlers and NoHeapRealtimeThread instances. Each mission has its own memory area in which temporary objects created in initialization mode will be allocated. When a mission's initialization has completed, mission mode is entered. When a schedulable object is started, its initial memory area is a scoped memory area entered when the schedulable object is released and exited when the release is terminated. This scoped memory area is not shared with other schedulable objects and therefore a ScopedCycleException cannot occur (Henties et al., 2009).

A safety critical profile developed in (Henties et al., 2009) and predictable profile developed in (Bøgholm et al., 2009) (more generalized profile based on RTSJ) feature a simplified scope based memory management structure where scoped memory is

implicitly created for each periodic task and cleared after execution of the task while it waits for the next periodic release. Design patterns were introduced to simplify the development of SCJ applications (Rios et al., 2012) such as "Execute with Primitive Return Value" pattern which is used when a piece of code needs to run in a scoped memory but a primitive value will be returned once exiting from that scope, and "Returning a Newly Allocated Object" pattern; the key point here is that objects created while executing in an inner scope need to be created in an outer scope. The authors suggested modifying some of the SCJ APIs to such as executeInArea() by executeInOuter() and to modify some of Java library classes such as HashMap, Stack, and Vector to be used safely in scoped memory areas and to reduce any possible memory leak.

SCJ case studies are rare, the cardiac pacemaker case study (Singh et al., 2012) has no dynamic load, it was proposed to evaluate the concurrency and timing models of two programming language subsets that target safety-critical systems development: Safety-Critical Java (SCJ), a subset of the Real-Time Specification for Java, and Ravenscar Ada, a subset of the real-time features provided by (Ada 2005). The main purpose of those profiles is to eliminate constructs with a high overhead or non-deterministic behaviour while retaining those constructs which ensure safe real-time systems. Results showed that extra timing procedures are required for the SCJ; on the other hand, a redundant task is required for an Ada solution to prevent premature termination of the system. A Desktop 3D Printer in Safety-Critical Java case study was developed by (Strøm and Schoeberl, 2012) as the first real SCJ-based application controlling a robot to evaluate the specification and its usability for developers of safety-critical systems. Results showed the need for tools to analyse Worst Case Execution Time (WCET) and maximum memory usage of the applications. A full

knowledge of the library code is required to prevent creating objects in wrong scopes and producing dangling references as a consequence.

2.3.3.4 Allocation time

Corsaro and Schmidt (2002) 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 (Corsaro and Schmidt, 2003) 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, the entering time of a scoped memory area in a TimeSys implementation varied slightly with changing scoped memory size from 4Kbytes to 1Mbytes, 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., (2007) two different implementations of the RTSJ were compared, namely Jamaica VM from Aicas and Sun's RTSJ 1.0.0. Their study analyzed 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., (2009) 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).

2.4 Benchmarks to evaluate RTSJ scoped memory

Table 2.3 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 (Salagnac et al., 2007) 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 and use of dynamic memory) which must be supported in a Java real-time environment. Results in Salagnac et al., (2007) showed that most of the benchmark applications used less heap space when using regions than garbage collection. However, some of the benchmark's applications such as Voronoi showed that garbage collection out-performed 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 (Cherem and Rugina, 2004) 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.

Notable benchmarks used in evaluating real-time Java implementations. Benchmark	**Where used?**	**Why used?**
JOlden (Salagnac et al., 2007)	(Cherem and Rugina, 2004, Salagnac et al., 2007)	To compare memory occupancy obtained during execution of different memory models.
CDx	(Pizlo and Vitek, 2006, Andreae et al., 2007, Garbervetsky et al., 2009, Kalibera et al., 2009)	To compare the performance of running in new RTGC to using scoped memory areas.
RTJPerf	(Corsaro and Schmidt, 2002, Corsaro and Cytron, 2003,)	• To compare different memory-reference checking schemes. • To measure the allocation time regarding different size of allocated objects. • To measure the entering/exiting times of scoped memory with respect to its scoped memory size.
JScoper	(Ferrari et al., 2005)	To enable automatic and semi-automatic tools to translate heap-based Java programs into scope-based ones, by leveraging GUI features for navigation, specification and debugging.

Two micro benchmarks (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 sever).	(Beebee and Rinard, 2001, Boyapati et al., 2003)	To measure the execution times of these programs both with and without scoped memory dynamic checks specified in the Real-Time Specification for Java.
Java SPEC suite (SPEC-Corporation, 1999)	(Deters and Cytron, 2002)	Allocate objects into scoped memory areas.

Table 2.3: Benchmarks to evaluate scoped memory in RTSJ applications

RTJPerf (Corsaro and Schmidt, 2002, Corsaro and Cytron, 2003) is an open-source benchmarking suite used to measure criteria of real-time Java systems and to apply different tests such as Timer tests, Threads scheduling tests and Asynchronous Event Handler Dispatch Delay tests. In Corsaro and Cytron (2003) 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 Corsaro and Schmidt (2002) 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 Corsaro and Schmidt (2003) 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 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 (Kalibera et al., 2009) is an open-source, real-time benchmark 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 an aircraft collision detection based on simulated radar frames. The input is a complex simulation involving over 200 aircraft. In (Pizlo and Vitek, 2006) the latency of processing one input frame was recorded when real-time garbage collection and a scoped memory management model were used. Results showed that scoped memory experienced better performance than real-time garbage collection. The OVM virtual machine was used in their study. In Garbervetsky et al., (2009) CDx was used to implement a transformation algorithm from plain Java code to a region-based Java code and five regions were created. In Andreae et al., (2007) CDx was used to evaluate a programming model known as STARS (the Scoped Types and Aspects for Real-time Systems) implemented 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 Deters and Cytron (2002) 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 was 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 which 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 Boyapati et al., (2003) and Beebee and Rinard, (2001), 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 a 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 which 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 Ferrari et al., (2005) 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 as presented in Garbervetsky et al., (2005) where object lifetimes are identified by using the call graph of available methods which include object creation sites. The tool enables the developer to visualize the transformation process, to create additional scoped memory areas and to delete or edit scoped memory areas. However, JScoper needs to be compatible with RTSJ applications. Moreover, its debugging approach for the memory model are highly recommended for future work (Ferrari et al., 2005), 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., (2010) 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 and ScopeCycleException,
- 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.

2.5 Potential Research Directions

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

First, there is no precise way to find out the lifetimes of objects to help developers in grouping objects into specific scoped memory areas. Research in this area can benefit from the research undertaken into finding similar lifetimes of objects in non–RTSJ implementations (Guyer and McKinley, 2004). For example, connected objects (objects that directly or indirectly call other objects methods or modify the 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 (Salagnac, 2008). 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 memory leaks 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 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 measure the allocation overheads of scoped memory areas. Real-time GUI tools which 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:

- What are the optimum criteria to allocate objects/threads in scoped memory areas in a way that leads to minimum consumption space and safe referencing? 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., Section 2.3.2, Section 2.3.3.2 and Section 2.3.3.3).

- How effective is using dynamic analysis tools that visualize object allocations into the scoped memory and measure the consumption over time in catching possible memory leaks? (c.f., Section 2.3.3.1).

- Can the application adapt different scoped memory models where one of them will be relied on according to specific priorities such as shorter execution time or smaller memory footprint? (c.f., Section 2.3.1 and Section 2.3.2).

- How effective are the aforementioned design patterns in simplifying the development process and avoiding both memory leaks and dangling references (c.f., Section 2.3.3.3)?

- How effective is scoped memory if it is applied to commercial real-time Java applications? This needs a thorough evaluation of the scoped memory model against a garbage collection model in these applications using benchmarks (c.f., Section 2.4).

2.6 Summary

The state-of-the-art in RTSJ memory management highlights 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 chapter 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) variety of 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 chapter. Consequently, there is a necessity to develop real-time Java case studies and benchmarks to help answer different research questions and provide guidelines and solutions for building the appropriate design of the memory model. Providing an empirical study for an RTSJ to understand different aspects and overheads of the scoped and immortal memory model is essential.

Chapter 3: Empirical Data Using A Scoped Memory Model

3.1 Overview

In order to propose guidelines and solutions for the scoped and immortal memory in RTSJ applications, an empirical study of the different aspects of this memory model when different types of objects are allocated is essential. This helps to specify the impact of using a scoped and immortal memory model on memory consumption and execution time of the application and consideration of an appropriate design of the memory model.

Prior data analysis using a scoped memory model has been limited. Most of the work has been done on measuring the allocation time of scoped memory at runtime (the time needed to allocate an array of bytes that comprise the object). For example, (Corsaro and Schmidt, 2002) showed that scoped memory allocation times were linear with allocated object sizes in a Timesys implementation, while in jRate the allocation times were independent of the allocated object sizes. In (Corsaro and Schmidt, 2003), the creation time (the time required for a scoped memory object to be declared and initialized), entering time and exiting time of the scoped memory area were measured with respect to scoped memory size. Results showed that creation time relied on the scope size for both implementations. On the other hand, the entering time of a scoped memory area in the TimeSys implementation varied slightly by changing scoped memory size (from 4Kbytes to 1Mbytes); in a jRate implementation on the other hand, 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. Enery at al., 2007

(Enery et al., 2007) compared two different implementations of the RTSJ, namely Jamaica VM from Aicas and Sun's RTSJ 1.0.0. Results showed that the creation times for scoped memory were again linear with scoped memory sizes. Object allocation times were also linear with object sizes. Schommer et al., (Schommer et al., 2009) evaluated the Sun RTS2.1 from different perspectives; the relationship between allocation time and object size allocated into memory areas was explored – and the relationship was again shown to be linear. It was concluded that allocation to immortal memory seemed, in general, to take longer than allocation to both scoped memory types (LTMemory and VTMemory).

The goal of this chapter is to enrich the empirical study of a scoped memory model from different aspects in an RTSJ implementation: the Sun Java RTS 2.2. Different data types in scoped memory may have different impact on the execution time and memory space. Therefore, Float, Hashtable and Vectors were tested to measure the execution time and memory consumption for each type when created inside scoped memory areas. The impact of increasing scoped memory numbers on execution time is investigated. Furthermore, an empirical study measuring the entering and exiting times of an active and non-active scoped memory area at runtime is presented. (The active scoped memory area is scoped memory that has one or more threads executing inside. A non-active scoped memory area is the scoped memory that has no threads running inside it.)

The contributions of this chapter are therefore:

1- Empirical data on allocating different data types into scoped memory areas.
2- Empirical analysis on the impact of changing scoped memory numbers and nesting on execution time.

3- Comparing the entering and exiting times of an active and non-active scoped memory area.

All code was run using the Sun Java RTS 2.2 implementation of RTSJ, the real-time operating system - Solaris 10 and on a stand-alone computer with Intel Pentium Dual Core Processor speed 2.8 GHZ, RAM, capacity 2GB and Hard disk size of 40GB. For all experiments in this thesis and to get precise results, the experiments were repeated 50 times and average execution times calculated. To avoid jitter (i.e., fluctuation in the execution time that may happen while loading and initializing classes at runtime), initialization time compilation mode (ITC) was used to compile and initialize classes at the virtual machine startup time and the real-time garbage collection disabled to prevent any interference that may occur in the heap memory.

The remainder of the chapter is organized as follows. Section 3.2 introduces empirical data on allocating different object types in scoped memory areas. The empirical analysis on the impact of changing scoped memory numbers and nesting on execution time is presented in Section 3.3. Section 3.4 highlights the overhead of entering and exiting active and non-active scoped memory areas. Finally, Section 3.5 concludes the work.

3.2 Empirical data for scoped memory area allocation

Before investigating the impact of increasing numbers of nested and un-nested scoped memory areas on the execution time of the application, it is important to study the impact of allocating different types of data objects in scoped memory areas. In this section, Integer, Float, Vectors and Hashtable data types are studied. We note that

Vectors are dynamic arrays and the elements of Vectors in the experiments are integer objects. Hashtables are data structures similar to arrays but are able to include different object types. In this study, an element of the Hashtable object is also a collection of integer objects. The execution time and memory consumption for each scoped memory area were measured.

RTS 2.2 syntactic code was run multiple times on Solaris 10, each time with a different object type and different number of objects (only one type is used in each iteration); this was done for two versions of the code, one with 5 scoped memory areas and the other one with 10 scoped memory areas. The two versions of code were used to allocate different numbers of objects in scoped memory areas to obtain valid and precise results. The number of objects was distributed equally across scoped memory areas. For example, with 5 scoped memory areas and 1000 integer objects, 200 integer objects are allocated into each scoped memory area; when Hashtable objects are used, each scoped memory area contains one Hashtable object which creates 200 integer objects. The same is true for Vector and Float types. On the other hand, in the case of 10 scoped memory areas and 1000 integer objects, 100 integer objects are allocated into each scoped memory area. Finally, when Hashtable objects are used, each scoped memory area contains one Hashtable object that creates 100 integer objects.

Table 3.1 and Table 3.2 show the results of these experiments for un-nested scoped memory areas. Nesting will be studied in Section 3.2.2 to measure its impact on execution time, regardless of what objects are allocated.

		Integer		**Float**	
	ObjectsNo	**Time (ms)**	**Memory (bytes)**	**Time (ms)**	**Memory (bytes)**
5 Scoped Memory Areas	**100**	6	752	15	2192
	500	9	2992	18	10192
	1000	11	5792	24	20192
10 Scoped Memory Areas	**100**	10	472	20	1192
	500	14	1592	22	5192
	1000	15	2992	28	10192

Table 3.1: Execution Time and Memory Consumption for each scoped memory area (Integer and Float)

		HashTable		**Vector**	
	ObjectsNo	**Time (ms)**	**Memory (bytes)**	**Time (ms)**	**Memory (bytes)**
5 Scoped Memory Areas	100	8	1720	7	848
	500	13	7384	11	3960
	1000	16	16264	13	7664
10 Scoped Memory Areas	100	13	952	12	504
	500	18	3800	16	2096
	1000	19	7384	17	3960

Table 3.2: Execution Time and Memory Consumption for each scoped memory area (Hashtable and Vector)

Results show that HashTable object type consumes more space in the scoped memory area and requires more execution time than the Vector object type. Float objects consume more space in the scoped memory area and impact the execution time more than the remaining objects types. When the number of scoped memory areas increases, the memory consumption for each scoped memory area decreases as the

number of objects allocated in each scoped memory area correspondingly decreases. However, execution time increases when the number of scoped memory areas increases. For example, with 5-scoped memory areas and 1000 integer objects, execution time is 11ms and the memory consumption for each scoped memory area 5792 bytes. When 10-scoped memory areas and 1000 integer objects are used, the execution time is 15ms and the memory consumption for each scoped memory area 2992 bytes. It is clear that Hashtable objects consume more memory than other object types.

Figures 3.1 and Figure 3.2 show a sample of the execution time and scoped memory area consumption, respectively for different data structures when 1000 objects are created in two versions of the application (5 and 10 scoped memory areas).

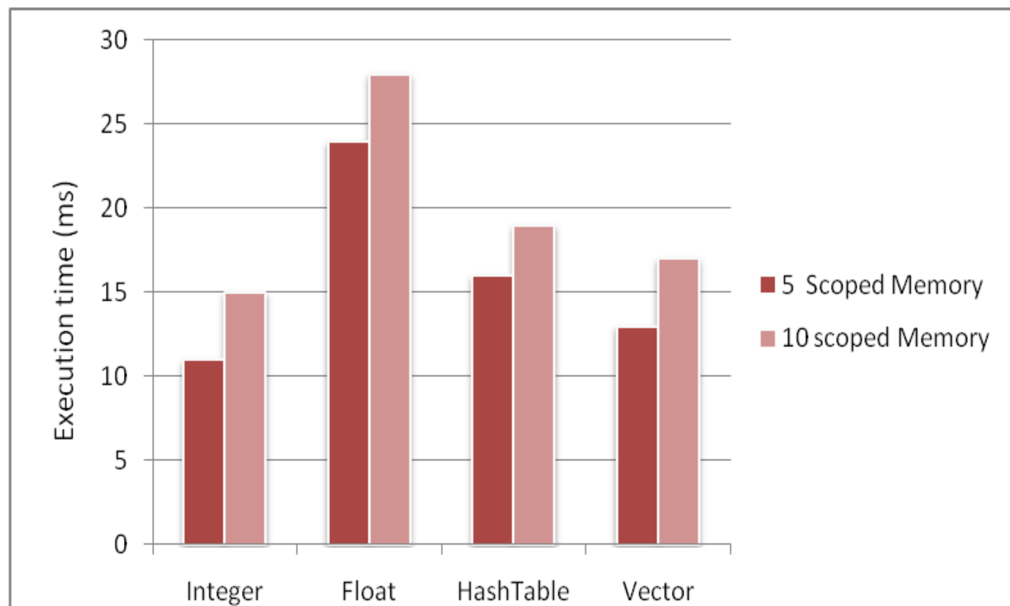

Figure 3. 1: Execution Times of 5/10 scoped memory areas application for different data types (1000 objects example)

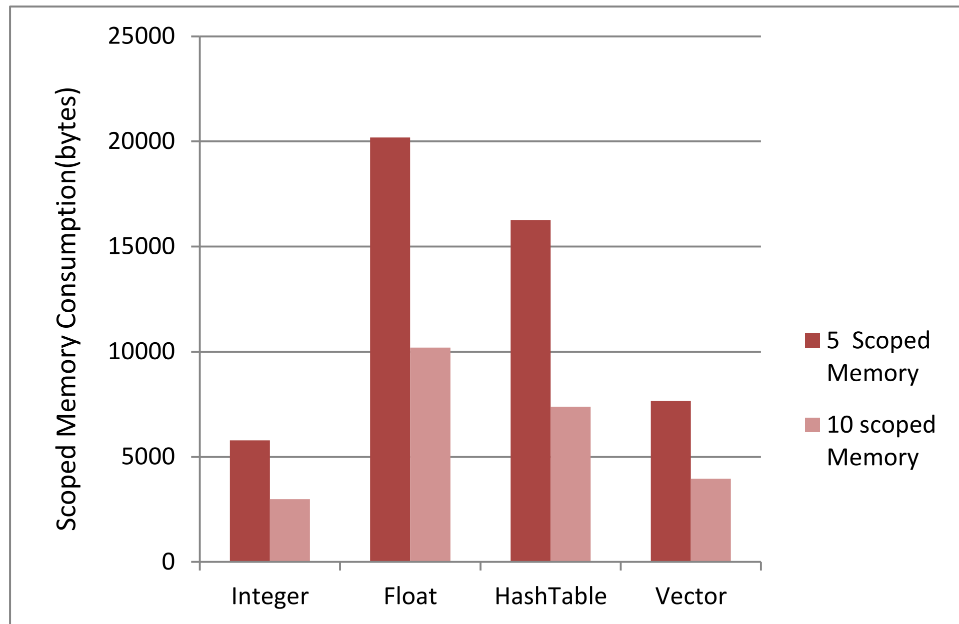

Figure 3. 2: Scoped Memory Consumptions of different data types when 1000 objects are created in 5/10 scoped memory areas application

Using scoped memory with different data objects has different impact on execution time and memory space; therefore, choosing the right data objects and the scoped memory size is likely to increase the efficiency of the scoped memory model.

3.3 The impact of changing scoped memory numbers and nesting on execution time.

The motivation for studying the impact of changing scoped memory numbers and nesting on execution time stems from two sources. It is the first study which assesses the relative merits of different numbers of scoped memory areas and the effect on execution times. Yet, the decision that a developer has to make on scoped memory area numbers can have a significant impact on potential application efficiency and execution time. Second, nested scoped memory areas have potential advantages of memory savings since child memory areas have shorter lifetimes than their parents;

the impact this has on application execution time and the inherent trade-off with those memory savings is an open research question. Nesting can be used, for example, when a thread needs to allocate different object lifetimes in memory; the thread then distributes these objects into different nested scopes according to their lifetimes (Andreae et al., 2007).

In this section, experiments were conducted to evaluate both un-nested and nested scoped memory area techniques to measure the impact of increasing levels of nesting over those scoped memory areas on execution times. In theory, higher numbers of scoped memory areas should lead to increased execution times (Deters and Cytron, 2002) since the memory management burden is naturally higher.

In all experiments, LTMemory object was used which guarantees linear-time allocation. Each memory scoped memory area is created by defining a new object memory area:

```
mem = new LTMemory(16*1024);
```

This creates a new LTMemory area with fixed size of 16K. The new object 'mem' then points to that scoped memory area of memory. To start using the block of memory referenced by 'mem', a 'Runnable' object should be used in the enter method of the 'mem' object; the Runnable interface is implemented by any class whose instances are intended to be executed by a thread. The same class must define a method of zero arguments called 'run'; all objects created inside the 'run' method of the 'Runnable' object will be allocated into the memory area referenced by 'mem'. The 'Runnable' object itself will be allocated to a different memory area - the memory area from which the 'enter' method of 'mem' object is called:

```
mem.enter(new  Runnable(){
  public void run(){
    // create new objects and run other tasks
  }
});
```

Memory scoped memory areas can also be nested in RTSJ. In other words, while executing code in the scope of memory ‘A’, an enter method for the scope of memory ‘B’ might be called. Henceforward, ‘A’ will be called 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. For example, in the following code, there is one nesting level, and two memory scoped memory areas are thus used:

```
memA.enter(new  Runnable(){
     public void run(){
         // create new objects and run other tasks
         memB.enter(new Runnable(){
             public void run(){
               // create new objects and run other tasks
               }
         });
         // create new objects and run other tasks
     }
});
```

In RTSJ, the outer scope is not permitted to reference any object in the inner scope, since the inner scope has shorter lifetime than the outer scope.

3.3.1 Experimental code design

Creating objects in the RTSJ code is facilitated through an array of objects (line 12 of Figure 3.3). The code can be updated with larger numbers of objects (from 100 to 2500, stepped by 100 objects upon each execution). Figure 3.3 includes a class definition for a simple, real-time thread (Example 1). In this thread, two new objects 'mem1' and 'mem2' are created to point to two scoped memory areas of memory (each of size 16K). All objects created in the 'run' method of the 'Runnable' object are allocated to that memory area. The array H of integer objects (50 objects) is created in mem1 and the array L of integer objects (50 objects) created in mem2 (lines 13 and 22 in Figure 3.3, respectively). Example 1 shows only 2 un-nested scoped memory areas allocating 100 objects in total. As an integral part of the analysis, the code was updated to include 5, 10, 15, 20 and 25 scoped memory areas. Example 1 was then updated to enable a re-run of the experiments using nested scoped memory areas.

All scoped memory areas have the same size (16K) and the number of objects distributed into each scoped memory area for each set of scoped memory area experiments is approximately equal. For example, for 5 scoped memory areas and allocation of 500 objects, each scoped memory area has 100 objects allocated to it. These objects are de-allocated when 'Runnable' objects finish executing their 'run' methods. A 'for' loop is used to execute the re-activation of the scoped memory areas multiple times according to the number of parameters entered. The type of parameter is thus Integer, and the values of these parameters are the values of the Integer objects allocated into the scoped memory areas. In the experiments presented, two Integer

parameters were used to execute the for-loop twice and execution time was measured using the Java clock method:

clock.getTime().

```
1.public class Example1with2scoped memory areas100objects extends RealtimeThread {
2.---------------
3. public void run(){
4.      mem1 = new LTMemory(16*1024);
5.      mem2 = new LTMemory(16*1024);
6.      for (int i = 0; i < this.args.length; ++i) {
7.            mem1.enter(new Runnable(){ //50 objects will be allocated
8.                         public void run()
9.            {
10.                   final int k = i;
11.                             Integer [] H= new Integer[50];
12.                             for( counter=0,  counter<50, ++counter){
13.                             H[counter]= Integer.valueOf(args[k]);
14.                         }
15.                      }});
16.           mem2.enter( new Runnable(){//50 objects will be allocated
17.                      public void run()
18.                           {
19.                    final int y = i;
20.            Integer [] L= new Integer[50];
21.                             for( counter=0,  counter<50, ++counter){
22.                             L[counter]= Integer.valueOf(args[y]);
23.                         }
24.                      }});
25.                    } //for loop
26.                 newTime= clock.getTime();
27.                 interval=newTime.subtract(oldTime);
28.                 System.out.println(interval);
29.      }; // for the run method
30. static public void main(String [] args)
         {  // main method of the class Example1with2scoped memory areas100objects
31.   RealtimeThread rt = new Example1with2scoped memory areas100objects(args);
32.  oldTime= clock.getTime();
33.   rt.start();
34.   try {
35.   rt.join();
36.  }
37. catch (Exception e) { };
38. }
39. }
```

Figure 3.3: Creating objects in un-nested scoped memory areas sample

3.3.2 Un-nested Scoped memory areas

Table 3.3 provides summary data (Mean, Median (Med.) and Standard Deviation (SD)) values for each of the set of un-nested scoped memory area experiments when allocating integer objects ranging from 100 to 2500 (integer objects). The widest variation in execution times is for 5 scoped memory areas (with an SD of 1.68) and the narrowest variation in execution time is for 25 scoped memory areas (SD of 1.41).

Number of Scoped Memory Areas	Mean	Med.	SD
5	10.68	11.47	1.68
10	14.99	15.65	1.51
15	18.66	19.20	1.67
20	20.90	21.37	1.48
25	25.16	25.52	1.41

Table 3.3: Summary data for un-nested scoped memory areas

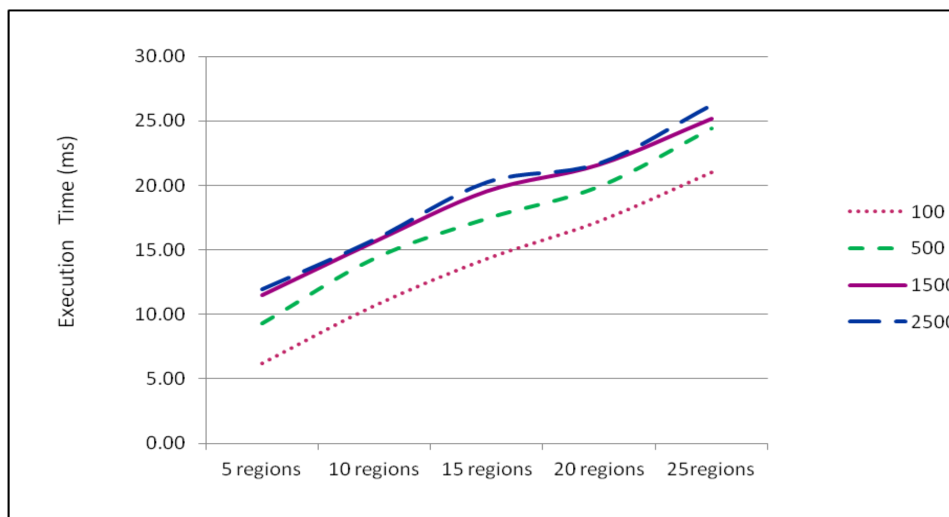

Figure 3.4: Execution time for un-nested scoped memory areas

Figure 3.4 shows how increasing the number of scoped memory areas increases execution time when the same number of objects is used. For clarity, only variations in time for 100, 500, 1500 and 2500 objects were shown. For instance, when 100 objects are distributed across 5, 10, 15, 20 and then 25 un-nested scoped memory areas the execution time of the application ranges from 6ms to 21ms. On the other hand, when 2500 objects are distributed across many scoped memory areas, the execution times are higher, ranging from 12ms to 26ms. It is interesting that for a period, the execution time for 2500 objects is close to the execution time of 1500 objects. Clearly, there are gains and losses to be made depending on the choice of number of scoped memory areas the developer has to make.

Figures 3.5 and 3.6 show the impact of increasing the number of the allocated integer objects on execution time for 5 and 10 un-nested scoped memory areas, respectively, with 100-2500 objects, stepped by 100, giving 25 data points for each figure. The R2 (correlation coefficient) value for 5 scoped memory areas (Figure 3.5) is equal to that for 10 scoped memory areas (Figure 3.6), with value 0.79 which means a strong relationship between the number of objects allocated in the regions and the execution time of the application.

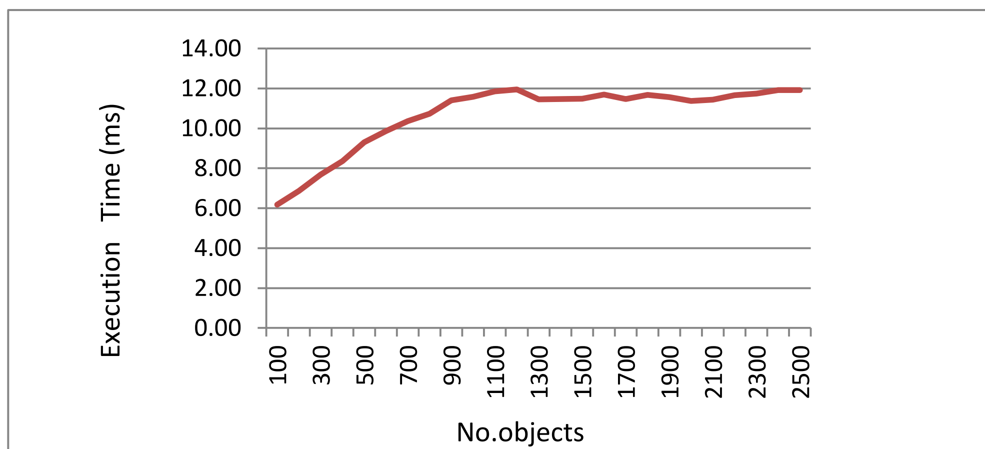

Figure 3.5: 5 scoped memory area data (2500 objects)

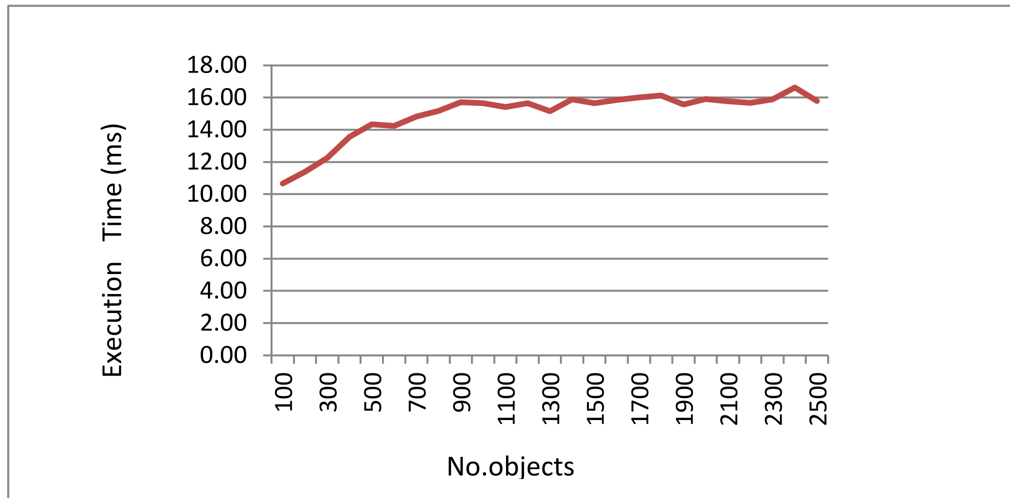

Figure 3.6: 10 scoped memory area data (2500 objects)

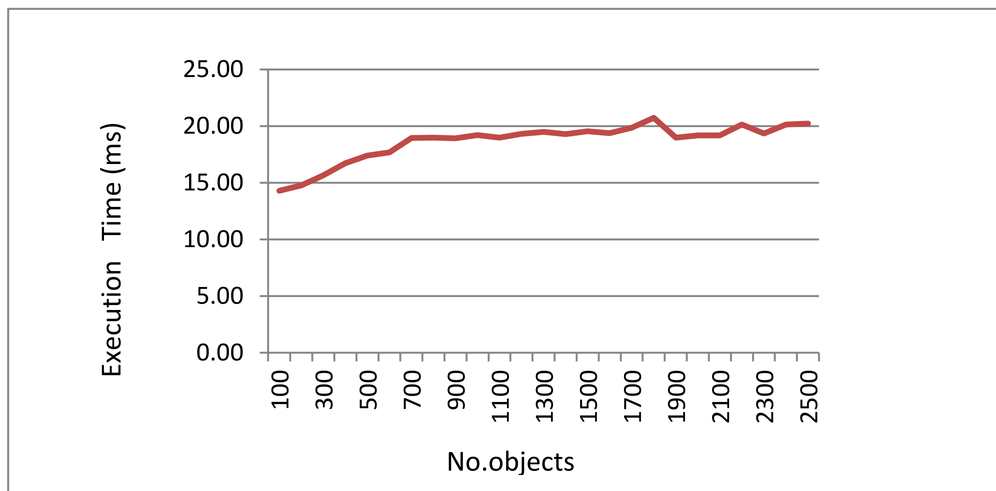

Figure 3.7: 15 scoped memory area data (2500 objects)

It is also noteworthy that the steepness of the curve in Figure 3.5 is greater at lower numbers of integer objects. One suggestion for this is that however many scoped memory areas are defined, there is a lower limit on execution time due to the overheads of actually creating the scoped memory areas and allocating the first n objects. After that point, the system appears to 'stabilize'. For 10 scoped memory areas (Figure 3.6), the execution time is higher than that of 5 scoped memory areas

(for the object configuration described). Figure 3.7 shows the effect on execution time of 15 scoped memory areas and shows a flatter slope.

Figure 3.8 and Figure 3.9 show the execution times for 20 and 25 scoped memory areas, respectively. The highest execution time amongst all configurations in fact belongs to 25 scoped memory areas (at configuration 26.40ms for 2500 objects), suggesting further that as the number of scoped memory areas increases, there is an associated natural overhead in execution time. Generally, the rise in execution times becomes flatter as the number of scoped memory areas increases.

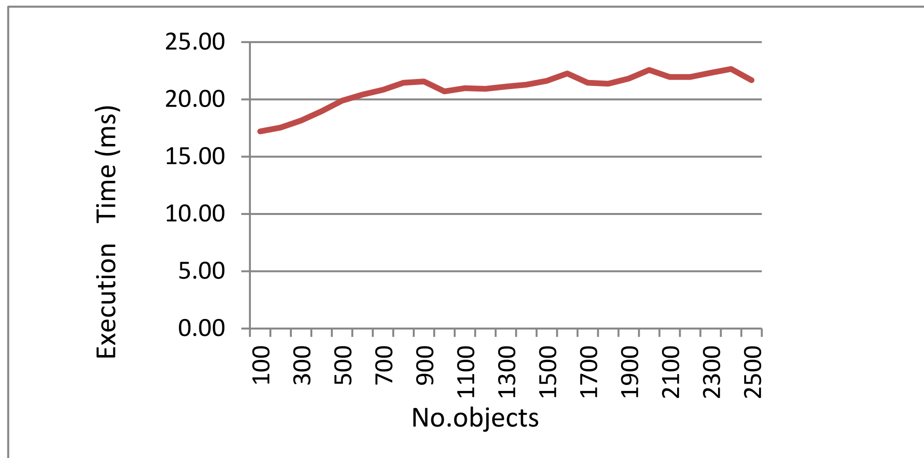

Figure 3.8:. 20 scoped memory area data (2500 objects)

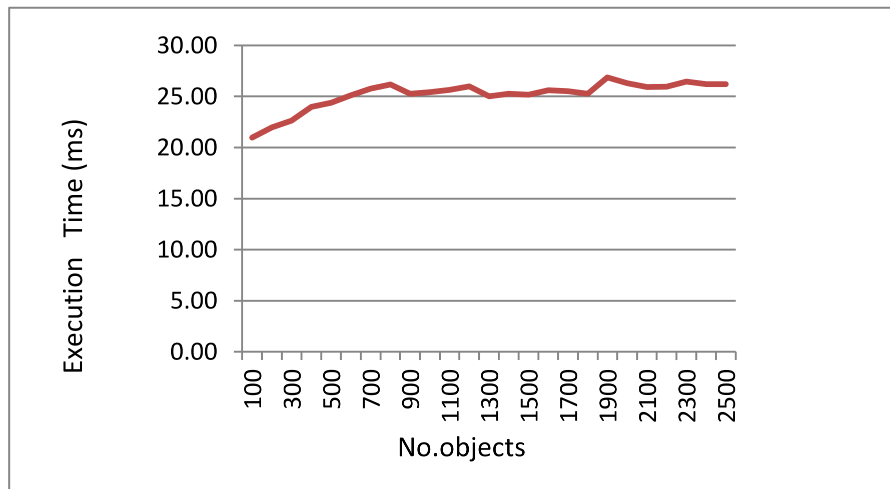

Figure 3.9: 25 scoped memory area data (2500 objects)

The general trend of the graphs in Figures 3.5, 3.6, 3.7, and 3.9 is upwards. However there are small falls in execution times along the graphs in the experiments due to the context switching jitter (Bruno and Bollella, 2009) of the multi-core machine upon which the experiments were run.

3.3.3 Nested Scoped memory areas

A key focus of this study is to assess, compare and contrast un-nested scoped memory areas with nested. To that end, experiments were repeated after updating the code to employ nested scoped memory areas. Figure 3.10 shows how increasing the number of nested scoped memory areas increases the execution time for four configurations of objects. When 100 objects are distributed across 5, 10, 15, 20 and then 25 nested scoped memory areas, execution time ranges from 6ms to 31ms. On the other hand, when 2500 objects are distributed across many scoped memory areas, execution times are higher, ranging from 12ms to 37ms. (For clarity, variations in time for 100, 500, 1500 and 2500 objects only are shown.) Again, as in un-nested scoped memory areas, it is interesting that, for a brief period, the execution time for 2500 objects is close to the execution time of 1500 objects, but this occurs at a lower number of scoped memory areas than for its un-nested counterpart.

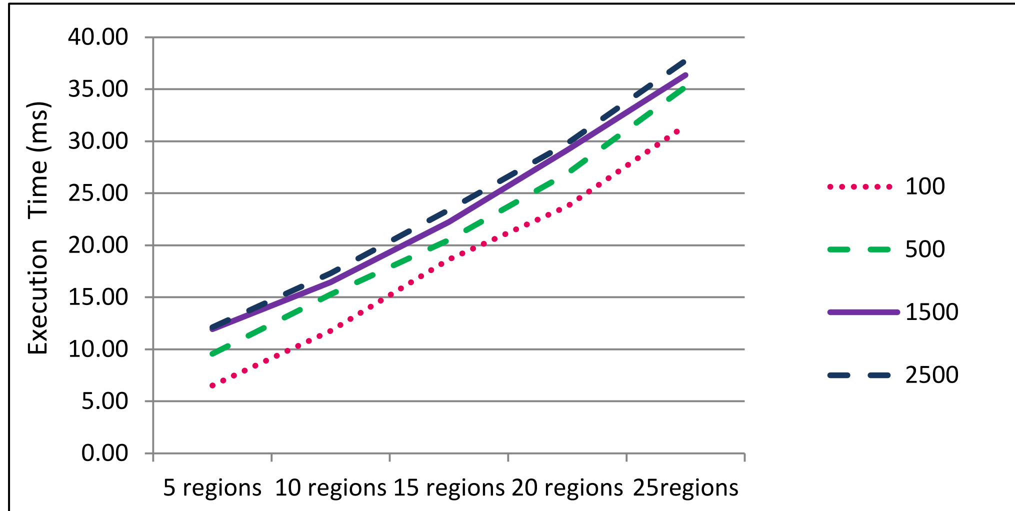

Figure 3.10: Execution time for nested scoped memory areas

Table 3.4 provides summary data (Mean, Median (Med.) and Standard Deviation (SD)) values for each of the set of nested scoped memory areas when allocating objects ranging from 100 to 2500 integer objects.

Number of Scoped Memory Areas	Mean	Med.	SD
5	11.07	11.84	1.71
10	15.97	16.39	1.53
15	21.85	22.32	1.57
20	28.21	28.79	1.79
25	36.11	36.57	1.73

Table 3.4: Summary data for nested scoped memory areas

The widest variation in execution times is for 20 scoped memory areas (with an SD of 1.79); the narrowest time is for 10 scoped memory areas (SD of 1.53).

Figures 3.11 and 3.12 show the average percentage increase in execution time when allocating the same number of objects into varying numbers of un-nested and nested scoped memory areas (5, 10, 15, 20, and 25), respectively. The execution time percentage increases between 5 and 10, 10 and 15, 15 and 20 and 20 and 25 scoped memory areas was calculated and the average of these values for each set of objects then calculated. For example, when 1000 integer objects were distributed across 5, 10, 15, 20, and 25 un-nested scoped memory areas, execution times were 11.58, 15.65, 19.23, 20.70 and 25.41 milliseconds, respectively (an average percentage increase of 22% - see Figure 3.11).

On the other hand, when 1000 integer objects are distributed across 5, 10, 15, 20, and 25 nested scoped memory areas, execution times were 11.84, 16.25, 21.40, 27.83 and 36.69 milliseconds, respectively (an average percentage increase of 33% - see Figure 3.12). All the values for increases in execution time are in the range 21%-37% for un-nested scoped memory areas and 30%-50% for nested scoped memory areas.

Figures 3.11 and 3.12 also exhibit a further interesting characteristic. The percentage increase at the beginning of the curve is higher when the number of objects is smaller. This implies that increasing the number of scoped memory areas for a small set of integer objects has a more significant impact on execution time than larger sets of integer objects. Clearly, the developer needs to choose the number of nested scoped memory areas carefully with a view to the direct effect this might have on resulting execution time.

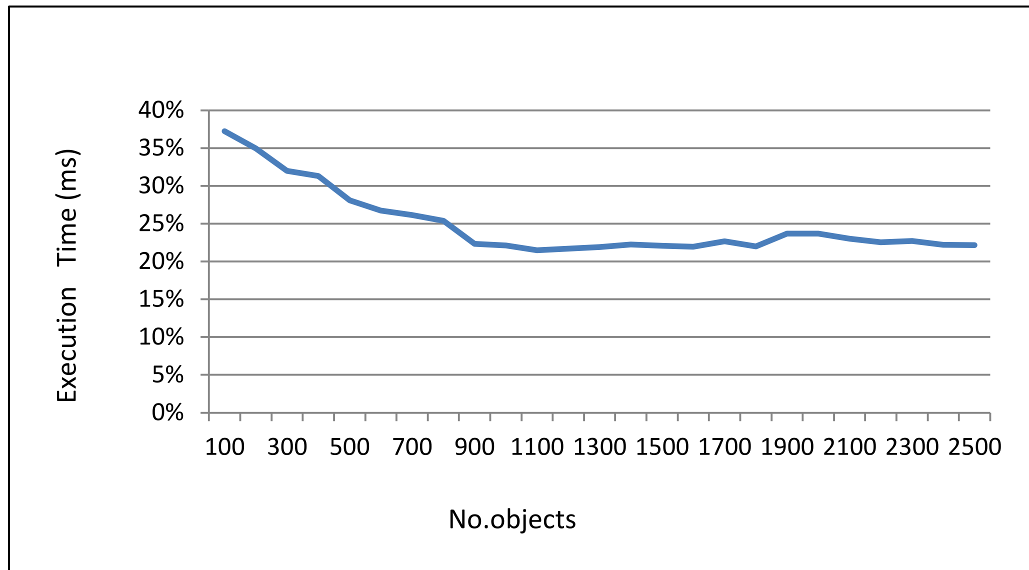

Figure 3.11: % in execution time increase (un-nested) scoped memory areas

Figure 3.12: % increase in execution time (nested scoped memory areas)

There is a clear difference in execution times when using nested scoped memory areas as opposed to un-nested scoped memory areas, especially when the level of nesting increases (Figure 3.13). One interpretation of this cost is due to the reference checks among nested scoped memory areas (to ensure that objects from outer scopes do not reference objects from inner scopes) and the single parent checks of the nested scopes (to ensure that each nested scope has only one parent scope). The more nesting

there is, the more checks there are among scoped memory areas. On the other hand, the 'Runnable' object of the child scope will be allocated into the parent scope; more objects will therefore be allocated in nested scopes than in un-nested scopes. Consequently, execution time will increase more in nested scoped memory areas than in un-nested ones.

Figure 3.13 shows the difference in execution times. For example, with 5 nested scoped memory areas, the execution time difference between it and its un-nested counterpart is on average of 0.39ms. There is even more variation in execution time for 10, 15, 20, and 25 nested scoped memory areas when compared to un-nested scoped memory areas. For example, the execution time for the 10 nested scoped memory areas code is approximately 1ms greater than that of the 10 un-nested scoped memory areas code. Similarly, there are 3ms, 7ms and 10ms approximate variations in execution time for 15, 20, and 25 nested scoped memory areas codes over 15, 20, and 25 un-nested scoped memory areas code, respectively. (All the values in Figure 3.13 are calculated by taking an average of the data for all sets of objects.)

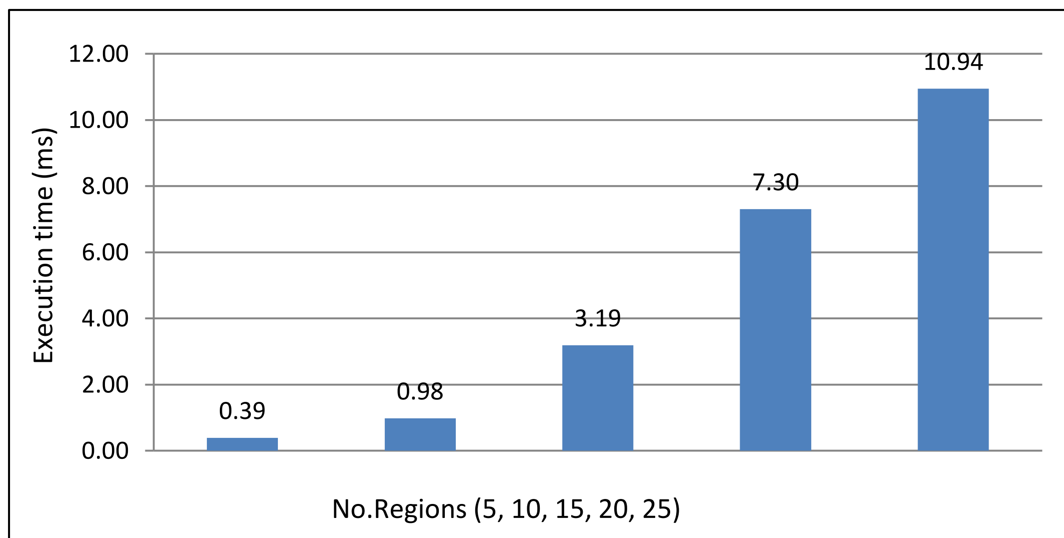

Figure 3.13: Differences in execution time (un-nested vs. nested)

Although using nested scoped memory areas saves memory space, the observed overhead on execution time is not trivial. Therefore, developing real-time applications using nested scopes should consider the balance between reducing space overhead and execution time overhead.

3.4 The entering/exiting time overheads of scoped memory areas.

This section introduces an empirical study measuring the overhead of entering /exiting active and non-active scoped memory areas at runtime. The motivation for this part of the study stems from the fact that scoped memory area can be entered by different threads at the same time. Investigating the difference between entering/exiting active and non-active scopes helps developers estimate the execution time overheads of different scoped memory design models. None of the studies in the literature have focused on entering and exiting time of active and non-active scopes. A syntactic real-time case study written in real-time Java that simulates a multi-threaded railway control system was developed (a full explanation on this case study is introduced in Chapter 4).

To compare the execution time overhead of entering/exiting scoped memory, two scoped memory design models were implemented in the case study. One is used to measure the entering and exiting time of an active scoped memory and the other one is used to measure the entering and exiting of non-active scoped memory. In both designs, to calculate the average of entering times and the average of exiting times of a scoped memory, the scoped memory that allocates the Train Status Table is considered since entering/exiting this scoped memory area occurs periodically

(frequent measurements are provided).

The first design model comprises one scoped memory area for each Train thread (Scopes Ai, i=1..n), one scope scoped memory area for each Emergency thread (Scopes Bj, j=1..m) and one scope for the Train Status Table (Scope C). The execution time of entering and exiting (Scope C) for allocating the Trains Status Table was measured. Scope C will be a non-active scope before being entered, to allocate the Trains Status table.

The second design model comprises one scoped memory area (Scope A) for all Train threads, the Train Status Table and one scoped memory area for each Emergency thread (Scope Bj, j=1..m). The execution time of entering and exiting (Scope A) for allocating and printing the Trains Status Table was measured; (Scope A) is an active scope since it has been entered beforehand by Train threads. Figure 3.14 shows how the entering and exiting times of a scoped memory area were calculated:

```
AbsoluteTime          beforeEnterTime,    enterTime,  beforeExitTime,
exitTime;
RelativeTime    enterOverhead, exitOverhead;
static  Clock    clock = Clock.getRealtimeClock();
beforeEnterTime=clock.getTime();
T_status_Mem.enter(new Runnable(){
        public void run(){
                  enterTime=clock.getTime();
                  enterOverhead=enterTime.subtract(beforeEnterTime);
                  // Allocate new objects
                  beforeExitTime=clock.getTime();
                  }
               }
           )
exitTime=clock.getTime();
exitOverhead= exitTime.subtract(beforeExitTime);
```

Figure 3.14: Calculation of entering and exiting times in scoped memory area

Figure 3.15 shows the entering time of the scoped memory for the two designs. The first design (non-active scoped memory) has greater entering time than the second design (the active scoped memory). The maximum value of entering a non- active scope is 22546ns while the maximum value of entering active scoped memory is 20395ns.

On the other hand, Figure 3.16 shows the exiting time of the scoped memory for the two designs. Apparently, a non-active scoped design model has a greater exiting time overhead than active scoped memory design. The maximum value of exiting a non-active scope is 13814ns while the maximum value of exiting active scoped memory is 7566ns.

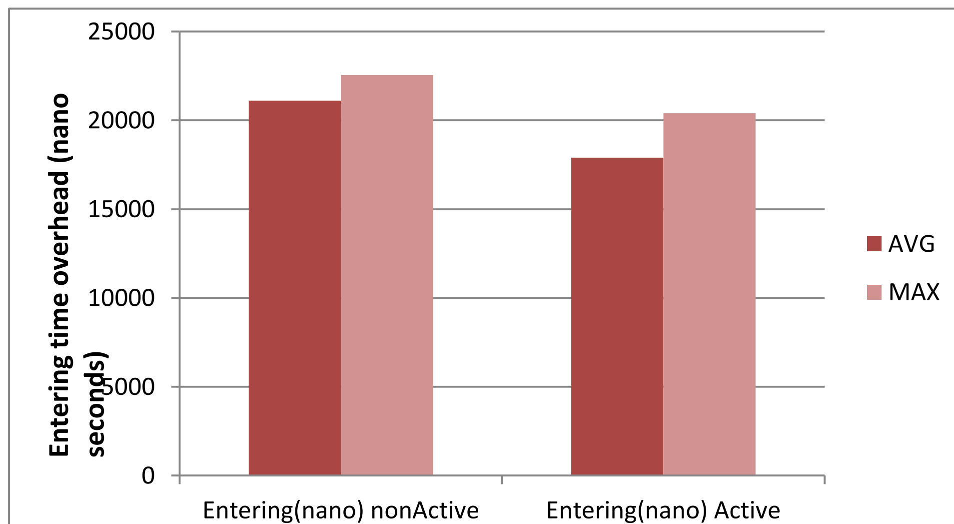

Figure 3. 15: Entering Scoped Memory Execution Time

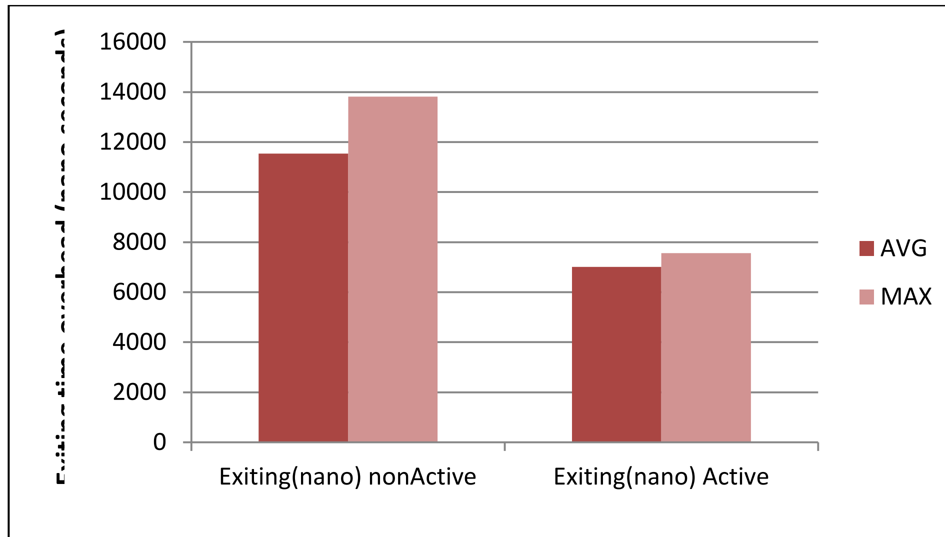

Figure 3. 16: Exiting Scoped Memory Execution Time

Summary data of the experiments for the two scoped memory design models is given in Table 3.5. It shows that Design 1 has more entering/exiting scopes time overhead than Design 2. Since the non-active scope needs to de-allocate objects after exiting the scoped memory area, it takes a longer time to exit; however, entering a non-active scoped memory should not show any differences when entering an active scope, since the backing store is allocated when the memory object itself is created. A possible explanation for this is that in this RTSJ implementation the work of clearing a scope is deferred to the next time the scope becomes in use. However, finalization of objects in the scope occurs as the last thread leaves.

Scoped memory design model	Entering (nano-Seconds)		Exiting Time (nano-Seconds)	
	Avg	Max	Avg	Max
Design 1, Non Active scope	21096.8	22546	11537.4	13814
Design 2, Active scope	17888.6	20395	7014.4	7566

Table 3. 5: Summary Data for Entering/exiting Scoped Memory

3.5 Summary

Developing RTSJ applications using a scoped memory model is a challenging task. Different design scoped memory models may exist. Scoped memory design models have different costs in terms of execution time and total memory consumption of the application. This chapter presented an empirical study of using scoped memory in Sun RTSJ Implementation. Allocating different data objects in scoped memory areas has different impact on the execution time and memory space; therefore, choosing the right data objects and scoped memory size has an effect on the efficiency of the scoped memory model.

The impact of scoped memory areas on the execution time of RTSJ software was investigated. Sample RTSJ code was executed with different numbers of un-nested and nested scoped memory areas. Results showed that increasing the number of scoped memory areas did lead to higher execution times. It is therefore important to find the optimal number of scoped memory areas. Additionally, the developer has to use nesting scope techniques carefully and maintain the trade-off between the pros and cons of using nested scoped memory areas.

The overheads of entering and exiting active and non-active scoped memory areas were also presented. Results showed that the entering/exiting active scoped memory scoped memory area had lower execution time overheads than entering non-active ones. The empirical data presented highlights a relevant issue for RTSJ development; in order to decrease the impact of the number of scoped memory areas on application execution time (and to save on memory footprint) an optimum number of scoped memory areas should be an aspiration for RTSJ developers. Consequently, a research question here would be: "what are the guidelines and rules that can help developers

decide on the right number of scoped memory areas and which threads/objects would be allocated in each scoped memory area?". Developing different real-time Java applications can assist in providing these guidelines. Equally, implementing and comparing different scoped memory models of the same real-time Java application provides an understanding of the impact and efficiency of using the appropriate scoped memory model.

Chapter 4: A Case Study of Scoped Memory Consumption

4.1 Overview

Specifying different overheads of using the new RTSJ memory model and developing real time Java case studies which include persistent dynamic allocation over period of time is required. This helps to evaluate the expressiveness of this memory model by providing guidelines and solutions for building a robust memory design model. On the other hand, to verify the memory model exceptions at runtime (such as OutOfMemoryError exception) and to monitor immortal memory consumption, the availability of assisting development tools is essential (Kalibera et al., 2010). RTSJ Case studies and tools for scoped memory development are still very rare. The CDx case study (Pizlo and Vitek, 2006, Kalibera et al., 2009) based on simulated radar frames was used to evaluate the time efficiency of applications which used scoped memory compared with the same version of applications that used real-time garbage collection. Results showed that scoped memory out-performed real-time garbage collection. The JScoper tool was presented in Ferrari et al., (2005) as a tool to transform standard Java applications into RTSJ-like applications with scoped memory management. 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 is not compatible with RTSJ applications.

In this chapter, an RTSJ case study is presented, namely a railway control system which combines multi-threading and scoped memory model implementations. A simulation tool is developed to measure and show scoped memory consumption of the case study over a period of time. Simulation tends to mimic software process and

give comprehensive feedback on the behaviour of that software before it is set up in its physical environment (Kellner et al., 1999, Benjamin and Steve, 2008). For safety-critical real-time systems, since rigorous verification of their functionalities, timings and memory consumption is required, simulating these systems before putting them into their real environment is an important practice for eliminating the cost of testing, reducing the risk of failure and ensuring high quality results (Rosenkranz, 2004). The simulation tool measures the scoped memory consumption of different scoped memory design models and presents the status of trains during the simulation's running time. In theory, the best scoped memory design model should achieve the least memory footprint. However, in some specific domains of real-time applications, the memory footprint is not an issue as long as the deadlines of real-time events are met.

The primary contributions of this chapter are as follows:

1. Provision of an additional RTSJ case study which integrates scoped and immortal memory techniques to apply different memory models.

2. A simulation tool for a real-time Java application (the first in the literature that we know of) that shows scoped memory and immortal memory consumption of an RTSJ application over a period of time. The tool helps developers to choose the most appropriate scoped memory model by monitoring memory consumption and application execution time.

3. Recommendations and guidelines for developing RTSJ applications which use a scoped memory model.

The remainder of the chapter is organized as follows. Section 4.2 introduces the simulation model. The experimental design is presented in Section 4.3. Section 4.4

explains the simulation tool. Simulation results are then discussed in Section 4.5. Guidelines for using scoped memory in RTSJ are listed in Section 4.6. Finally, Section 4.7 concludes the chapter.

4.2 Simulation Model

In order to analyze and monitor the memory consumption of an immortal and scoped memory model in real-time multi-threading environments, a simulation model has been implemented which can be adapted to different real-time systems using real-time Java. A Model can be considered as a representation and abstraction of an entity, a real system or a proposed system. Simulation is experimenting the model for analysis purpose and problem solving objectives (Taylor et al., 2013). Figure 4.1 shows the proposed simulation model for the multi-threaded, real-time Java system. The simulation model consists of the following components:

1. A Main thread which initializes system threads and starts the application.
2. A Monitor thread which checks the safety of the studied real-time system.
3. A Control thread which updates the status of the control components.
4. Real-time threads; components that build the core system and distinguish it from other systems.
5. A live thread Monitor to re-activate real-time threads.
6. A GUI and Console tool to present the data obtained by running the simulation.

To measure the cost of the simulated system in terms of memory consumption and execution time, three criteria are identified: scoped memory consumption, immortal memory consumption and tuning of the parameters of the system. The parameters of the system configure the deadlines of periodical threads and the maximum space allocated for immortal and total scoped memory.

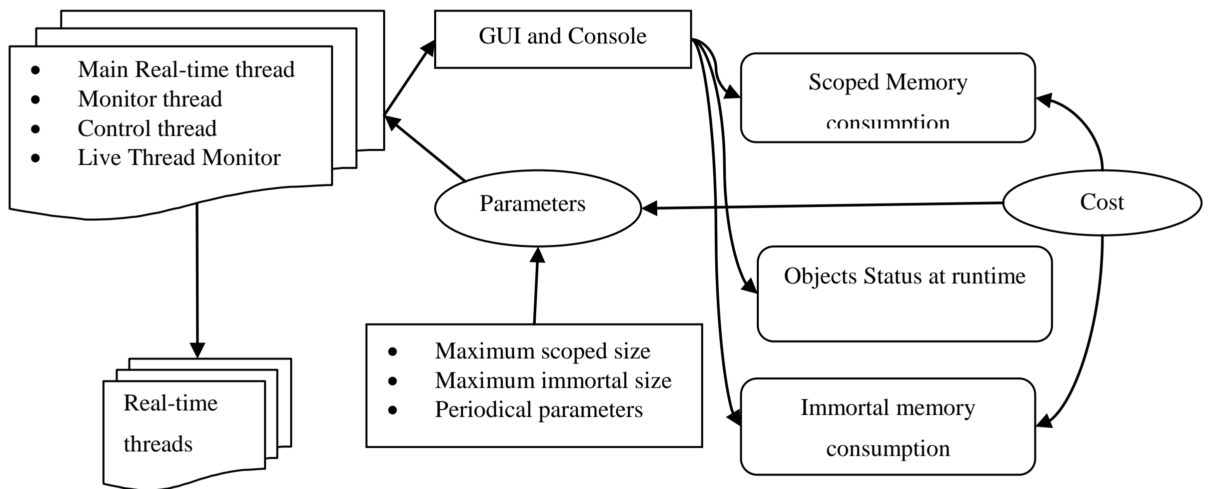

Figure 4.1: Simulation Model for a Real-Time Java Scoped memory Model

A railway control system is a safety critical, multi-threaded real-time system which needs to respond to events under hard, real-time constraints. This system must be aware of any emergencies that might happen. For instance, if two trains are given access to a specific track at the same time, a possible collision or delay may occur and, in this case, the system should send signals to both trains to make them slow down and/or to divert one of them onto an alternative track.

This case study has the following main objects and real-time threads (Figure 4.2). Care was taken to ensure that the simulation provided a model of a sufficient number of attributes of the system to promote realistic experiments. This simulation is an

event-based simulation since some events such as emergencies may arise and equally, trains starting a new route after finishing their first route are considered as waiting events. Since the railway control system runs in a multi-threaded environment and contains periodic threads, the simulation is considered as process-oriented. Therefore, this simulation is a mix of discrete-event and process-oriented simulation.

- **The Main Thread** is the main thread from which the railway control system starts. This will create and initialize the Track object and create and start the Train Threads, Monitor Thread and Control Thread.

- **Track object** is a Hashtable object which contains entries for the possible tracks in the system. Hashtables are data structures similar to arrays but are able to include different object types and may also have unlimited size. Each entry in any Hashtable comprises a key and a value. In the case study for example, each entry will comprise {TrackName - a key, TrackStatus - the value}. In this study, it is assumed that the system has 10 tracks and each has one sensor and two traffic lights on each side of the track. The initial status of the tracks is (sensors - ‘OFF’, traffic lights - ‘GREEN’).

- **Train Threads:** each train in the system is simulated by a real-time thread which has the following parameters in its constructor: route of the train, name of train and the scoped memory area in which the thread will run. The Train Threads send messages to the system when the train is waiting for a specific track to be freed.

- **Control Thread**: this thread checks sensors on the tracks periodically and updates the status of the traffic lights. If the sensors are ‘ON’, the traffic lights on the related track will be ‘RED’ preventing any other train passing

through this track; otherwise, the traffic lights are 'GREEN', allowing a waiting train to pass through. The sensors are set to 'ON' by a train that starts moving on the related track and, as a result, the traffic light on the other side of the track will be set to 'RED'. When the train exits the track and starts moving to the next track in its route, the Train Thread will set the sensors 'OFF' and the traffic light will be set to 'GREEN' by the Control thread.

- **Monitor Thread**: this thread runs periodically to update the status of trains and check if there is any possibility of collision between trains. If there is a possible collision according to a specific criteria then it will instantiate an Emergency Thread. It is assumed that a collision occurs when, for instance, the Control Thread delays updating of the status of the tracks due to any failure in the system; as a consequence, two trains are set on the same track, one at each end of the track. The Train Status Table object is generated periodically by the Monitor Thread to show the status of all trains (i.e., locations on their routes).

- **Live Thread Monitor**: this thread runs periodically every second to check whether all trains have terminated their routes so as to reassign to them new routes. This means creating new Train Threads with new routes; these new objects will be allocated into the same memory area running in the previous route.

- **Emergency Thread**: is a real-time thread with high priority that will execute in a different memory area. It prevents a possible collision between two trains by decreasing the speed of each and makes one of them

divert to a temporary track while waiting for the other train to pass. It also sends a message signal to both trains to notify them.

- **Restriction Object**: this object is created by the Emergency Thread to slow down the speed of both trains that might potentially collide and diverts one of them onto a temporary track until the other train has passed through.
- **Message Object**: this object is created by the Emergency Thread in order to pass a message to both trains' screens.

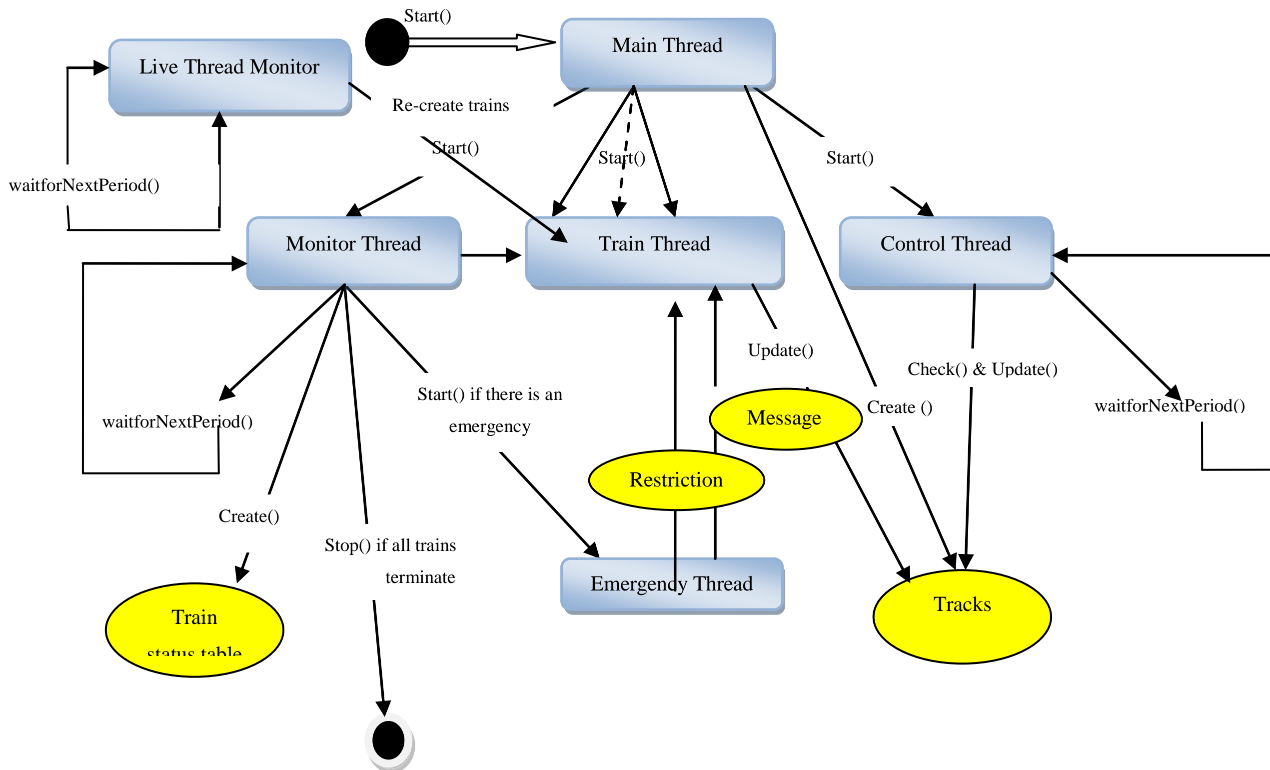

Figure 4. 2: The main objects and threads in the Simulator

4.2.1 Assumptions of the Simulator

- In order to make the simulation more realistic and run for a long period, trains were configured to run on 4 different routes; once a train finishes its specified route, it will run on its next specified route. Consequently, the simulation runs for approximately 6 minutes which is a reasonable time period to cover all cases that might happen and collect the right data. For the nature of the case study developed and for showing the salient aspects of the tool, running the simulation for that period of time is also sufficient to demonstrate the viability of the tool and for drawing appropriate conclusions about the scoped memory model.

- The Train Thread starts after the Control Thread and Monitor Thread start. The number of Train Threads in the experiments is 16 and this can be increased for other experiments. The Train Threads have different routes that are, *a priori* defined inside the Main Thread. The route is a ‘String’ array of track names such as route= {“T1”, “T2”, “T5”, “T8”}. When each train terminates at the end of its first route, the train will start a new trip immediately. Route objects in the experiments are defined randomly and they share similar tracks; for example in the following code, routes 1, 2, 3 and 4 all share the track “T4”.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
String[] route1={"T1","T4","T3"};     //Train1 route
String[] route2={"T4","T6","T7"};     //Train2 route
String[] route3={"T9","T8","T6","T5","T4"};//Train3 route
String[] route4={"T5","T4","T3","T2"};  //Train4 route
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- As a simulation of how trains move across the routes, a 'percentage of progress' variable for each Train Thread is defined. This variable increases its value from 0 to 100, where 0 denotes that the train will move on the current track and 100 denotes that the train finishes on the current track and will move over to the next track with a new zero-value assigned to its 'percentage of progress' variable.

- An assumption is made about the emergency checking condition inside the Monitor Thread. The condition checks whether any two trains in the system are allocated onto the same track from both ends of the track and that they are sufficiently far away from each other. Before they get close, the system should respond in real-time. For instance, the Emergency Thread could occur between Train 1 and Train 4 since both of them might arrive at the same time onto Track "T4"- the second track in their assigned routes in the case study routes: (route1={"T1","T4","T3"} and route4={"T5","T4","T3","T2"}).

- The Control Thread and Monitor Thread are both periodic real-time threads. Moreover, both have to meet strict timing deadlines for completing their tasks every period to satisfy the real-time constraints of the system. For example, the Monitor Thread should finish its checking of the status of the trains within 50ms. The Control Thread should run more frequently than the Monitor Thread since it needs to update the tracks' traffic lights instantly according to the sensor status. Therefore, the scheduling parameters for both Control Thread and Monitor Thread were tuned to ensure that both of them accomplished their tasks within very short periods. Through preliminary experiments of the case study, the

Control Thread can accomplish its tasks within 120ms and Monitor Thread within 300ms. Those two values of periodic parameters are fixed throughout all the experiments and can be fine-tuned if there is a need to increase or decrease the number of trains and/or tracks in the system.

4.2.2 Scoped Memory Design Models

Since the case study is a safety critical application, allocating objects and threads onto heap memory was avoided to ensure that no interference by the garbage collection process was encountered. Therefore, the first challenge was to know how many scopes the application needed and which objects and threads should be allocated into either these scopes or immortal memory. To decrease the memory footprint of the case study, similar lifetime objects should be allocated into the same scope; short lifetime objects should be allocated into different scopes to that where long lifetime objects reside. The lifetimes of different threads and objects in the case study vary and some are not specified at compile time. The Lifecycle Memory Managed Periodic Worker Threads pattern introduced in Dawson (2007) is used as a fundamental concept to design different scoped models for this case study; this pattern has four categories of object lifetimes:

- **Retain Forever**: Objects with this lifetime are alive until the application terminates and are accessible to all threads.
- **Retain Thread Group**: Objects with this lifetime will not be reclaimed until all the threads that share these objects have terminated. These objects are accessible only by threads within the group of threads.

- **Retain Thread**: Objects with this lifetime will be created by a specific thread and will not be accessible by other threads.

- **Retain Iteration**: Objects with this lifetime are created during the iteration and will not be used outside of the iteration.

Table 4.1 shows the initial and possible design memory solutions of the case study from a thread/object lifetime's perspective. From the initial design, it is essential to define which objects/threads should be allocated into either immortal memory or in scoped memory regardless of how many scopes are required. Since the Track object will be accessible from different threads during the application's lifetime, it is reasonable that it should be allocated into immortal memory (Retain Forever). As a result, the Track object will be accessible by all threads that run in different scoped memory areas, so the assignment rule of RTSJ is satisfied (i.e., references from scoped memory to immortal memory are always allowed).

On the other hand, the Main Thread will also be active until the application terminates; therefore, it is more appropriate to be allocated into immortal memory (Retain Forever). Similarly, the Control Thread lasts throughout the application's execution time and it should be allocated into immortal memory (Retain Forever). Finally, the Monitor Thread and Live Thread Monitor will be allocated into immortal memory (Retain Forever), since they will last for the entire application's lifetime.

The Trains Threads are real-time threads and so their lifetimes are not specified at compile time; trains might wait for other trains to proceed and this is related to the status of the tracks; exactly how long each train needs to finish is not known beforehand. On the other hand, the Train Thread will create new temporary objects while it is running such as a new temporary object to read the current track from the

Hashtable entries and a string message issued when the train is in a waiting state. These objects should be de-allocated when the train terminates its route (Retain Thread). The Train Thread should therefore be allocated into a scoped memory area where all objects created by the Train Thread will be de-allocated (and when no threads run inside that scoped memory). The Train Status Table will be created by the Monitor Thread to periodically show the status of all trains (Retain Iteration). Allocation of the Train Status Table by the Monitor Thread to a scoped memory area saves on memory footprint. Each periodic run of the Monitor Thread will create a new Train Status Table de-allocated after the Monitor Thread finishes its current period. Hence, no memory leak occurs.

An Emergency Thread will be instantiated by the Monitor Thread when an emergency state occurs between two trains and it will last until the emergency is handled; the Emergency Thread is therefore a temporary thread and will be allocated in a scoped memory area (Retain Thread).

The Emergency Thread creates new objects such as the Message and Restriction objects. The Message object sends messages to both trains to inform them of the emergency state and the Restriction object handles the emergency by modifying the trains' parameters. The Emergency thread will communicate with two Train Threads which run in scoped memory areas; however, their references are stored in immortal memory, since the Main Thread that creates these references is allocated into immortal memory. Therefore, the Emergency Thread can access immortal memory and extract references to both Train Threads. If the Main Thread was not allocated into the immortal memory, the Emergency and Train Threads would not be able to communicate, since the reference between two separate scopes (not siblings) is not permitted under RTSJ rules.

Design	Immortal Memory	Scoped memory
Initial Design	Control Thread Monitor Thread Main Thread, Tracks object Live Thread Monitor	Train Threads Emergency Thread Train Status Table
Design 1	Control Thread Monitor Thread Main Thread, Tracks object Live Thread Monitor	• One scoped memory for EACH Train Thread • One scoped memory for each Emergency Thread • One scoped memory for Train Status Table
Design 2	Control Thread Monitor Thread Main Thread, Tracks object Live Thread Monitor	• One scoped memory for ALL Train Threads • One scoped memory for each Emergency Thread • One scoped memory for Train Status Table
Design 3	Control Thread Monitor Thread Main Thread, Tracks object Live Thread Monitor	• One scoped memory for ALL Train Threads and all Emergency Threads • One scoped memory for Train Status Table

Design 4	Control Thread Monitor Thread Main Thread, Tracks object Live Thread Monitor	• One scoped memory for ALL Train Threads and Train Status Table • One scoped memory for each Emergency Thread
Design 5	Control Thread Monitor Thread Main Thread, Tracks object Live Thread Monitor	• One scoped memory for ALL Train Threads • One scoped memory for Emergency Threads and Train Status Table
Design 6	Control Thread Monitor Thread Main Thread, Tracks object Live Thread Monitor	One scoped memory for all Train Threads, Emergency Threads and Train Status Table

Table 4.1: initial and possible design memory models of the case study

All the objects/threads in this case study are logically related and allocating them into many different scopes according to their lifetimes presents the possibility of obtaining a better memory footprint (as stated in the Lifecycle Memory Managed Periodic Worker Threads pattern). From the initial design, it was found that Train Threads, the Emergency Thread and the Train Status Table are allocated into scoped memory areas; deciding on the number of the scoped memory areas of the aforementioned objects is left to the developer. Accordingly, for the sake of the tool experiments, there are three different allocation scenarios as shown in Table 4.1 (Designs 1 to 6):

- All Train Threads, Emergency Threads and Train Status Table will be allocated into the same scope (Design 6). It is trivial to implement Design 6 since all Train Threads, Emergency Threads and the Train Status Table will be allocated into one scope; they are not de-allocated until all Trains Threads finish their routes at the end of the application.

- Each two of the three (Train Threads, Emergency Threads and Train Status Table) will be allocated into one scope and the third will be allocated to a different scope (Designs 3, 4 and 5).
- Each of: Train Threads, Emergency Threads and Train Status Table will be allocated into different scoped memory areas. On the other hand, since trains share tracks with other trains, their behaviour cannot be predicted in a control system of the type that has been defined and, accordingly, they will have different lifetimes. Therefore, it may be prudent to allocate them to different scopes. Here, two different designs can be implemented, since Train Threads can either all be running in one scoped memory area or each can be running in a different scoped memory area (Design 1 and Design 2).

4.3 Experimental Design

The experimental design of the simulation tool consists of:

- Implementing each scoped memory design model (Designs 1 to 5).
- Modeling the movement of trains: Each train has a variable named 'percentage_of_progress' which simulates the train's run on a specific track. This variable increases its value from 0 to 100, where 0 denotes that the train starts moving on the current track and 100 denotes that the train has completed its run on the current track. The following code illustrates how train movement is modeled on a specific track:

```
--------------------------------------------------------
while (percentage_of_progress <=100)  // train is  still running on the  current
                                       //track
```

```
{
  percentage_of_progress = percentage_of_progress +1;
  this.sleep(100);
 /* The Train Thread sleeps for 100 millisecondes and then it continues
    moving on the current track untill its percentage_of_progress
    variable  reaches 100.
 */
 };// while loop
// the train moves into the next track
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- Measuring, modeling and visualizing memory consumption: Different scoped design memory models of an RTSJ application might show different memory footprints during execution of the application. To capture the best scoped memory design model for the case study, it was run with different versions, each one of which implemented one of the scoped memory design models (Designs 1 to 5). As previously mentioned, Design 6 comprises one scoped memory area for all Train Threads, Emergency Threads and Train Status Table. There are therefore no benefits in implementing it, since one scoped memory will still be alive until all Threads terminate. Therefore, five different memory design models were implemented. Immortal memory and total scoped memory consumption for each design was then measured.

The following code shows an example of how scoped memory areas were assigned to Train Threads and immortal memory areas to Control Thread and Monitor Thread in the ‘run’ method of the Main Thread. New Scoped memory objects are created with different sizes to match the experiment’s requirements. The Train Thread instances are created and parameters are assigned to their constructors; those parameters are a) route, b) name of the train and, c) the memory scoped area in which it will run. Both Control Thread and Monitor Thread run inside the immortal memory instance.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
Train train1 =new Train(route1,"train1",ScopedMem1);
Train train2 =new Train(route2,"train2",ScopedMem2);
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ControlThread  Control=    new  ControlThread(ImmortalMemory.instance( ) );
MonitorThread Monitor=    new  MonitorRTThread(ImmortalMemory.instance() );
Control.start();
Monitor.start();
train1.start();
train2.start();
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The application will run until all Train Threads finish executing. The memory consumption of immortal and total scoped memory areas are calculated using the RTSJ memoryConsumed method of the MemoryArea object. Memory consumption is calculated every time the periodic Monitor Thread is run. An example of how total memory consumption of all scoped memory areas and how immortal memory is measured is shown in the following code:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
while ( waitForNextPeriod() )
    {
       // calculate immortal consumption
        Immo=(int) ImmortalMemory.instance().memoryConsumed() ;
      // calculate scopes consumption
      TotalScopesConsumption= Main.ScopedMem1.memoryConsumed()+
                              Main.ScopedMem2.memoryConsumed()+
                                Main.ScopedMem3.memoryConsumed()+
                                Main.ScopedMem4.memoryConsumed();
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    }//whileloop
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The experiments were repeated 50 times for each data point and the average memory consumption was calculated. To avoid jitter (i.e., fluctuation in execution time that may occur while loading and initializing classes at runtime), initialization time compilation mode (ITC) was used to compile and initialize classes at the virtual machine start-up time. Since each design may have different scoped memory consumption, the maximum size of scoped memory was tuned for each design. The maximum size needed for immortal memory was tested through the experiments and

it was equal to 12Mb. Those values were tuned before the virtual machine started executing. Table 4.2 shows the platform of the experiments.

OS	Solaris 10/x86
VM	Sun RTS 2.2
CPU	Intel Pentium Dual Core 2.8 GHZ
Immortal size	12Mb
Maximum size of scoped region	1600KB
RAM capacity	2GB

Table 4.2: The simulation platform

4.4 Simulation Tool

The simulation consists of two parts: the GUI and the Console. The simulator was run for approximately 6 minutes, after which all trains had finished their routes and the application then terminated. The GUI presents the status of tracks and trains during the simulation execution time and shows the total memory consumption of scoped and immortal memory of the implemented design. The status of the trains can be either one of the following:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Train is on wait.
Train has terminated at the end of its first route.
Train has terminated at the end of its second route.
Train has terminated at the end of its third route.
Train has terminated at the end of its fourth route.
Train has stopped waiting for train(x) to finish its current track.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The two rectangle elements at the top of the tool interface show the memory consumption percentage of the maximum memory assigned at runtime for each immortal memory and scoped memory areas. In the simulation, the maximum space of memory allocated for immortal memory was assigned 12Mb and the maximum space of memory allocated for scope areas 1600Kb for all designs. The white box at

the right bottom corner of the tool presents the track status during the simulation execution time. T0 to T9 represent the names of the tracks; the status of each is either “Green” or “Red” which reflects the status of their traffic lights to allow or prevent trains from running on that specific track. The emergency message at the bottom of the GUI is displayed if there is an emergency between two trains in the system. The time label shows the time at which the simulation runs.

Figure 4.3 shows a screenshot of the simulation’s GUI part at a period of 140 seconds during Design 1. It also shows the status of all trains and current traffic lights of the tracks. Track T3 for example, is Green at that moment which means that there are no trains running on it. Train 1 status for example is ‘T7’.

There is a possibility of two trains being on the same track as seen in Figure 4.3 where Train11 and Train2 are in an emergency state but no collision result; in this case, either both trains were running in the same direction but with acceptable speed and there was no possibility of a collision or the trains were far enough from each other and both safe. When they moved closer to each other, one of them was stopped on an alternative track until the other train passed. Figure 4.3 shows that Train11 is on wait state until Train2 finishes its run on track T8. Choosing which train to be stopped to wait is defined randomly by the system which will send a message object to both trains to give the appropriate instruction. An Emergency Thread created at that time between Train11 and Train2 is shown at the bottom of the screenshot. The percentage string shown on scoped memory component displays the current consumption percentage of the maximum scoped memory allowed in the system. Similarly, the percentage string shown on the immortal memory component displays the current consumption percentage of the maximum immortal memory allowed in the system.

Figure 4.4 shows the simulation at period of 299 seconds when most trains terminated in Design 1. The scoped memory consumption is 7.39Kb and immortal memory consumption is 9.77Mb.

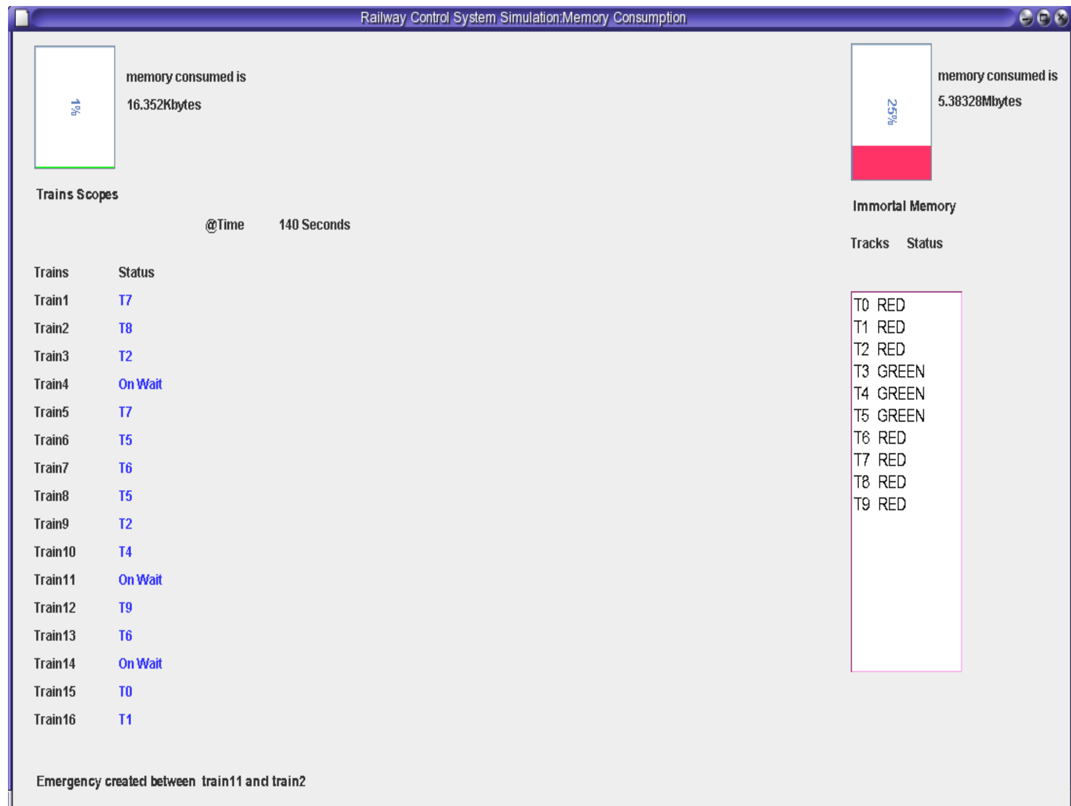

Figure 4.3: Simulation GUI element at 140 seconds (Design 1)

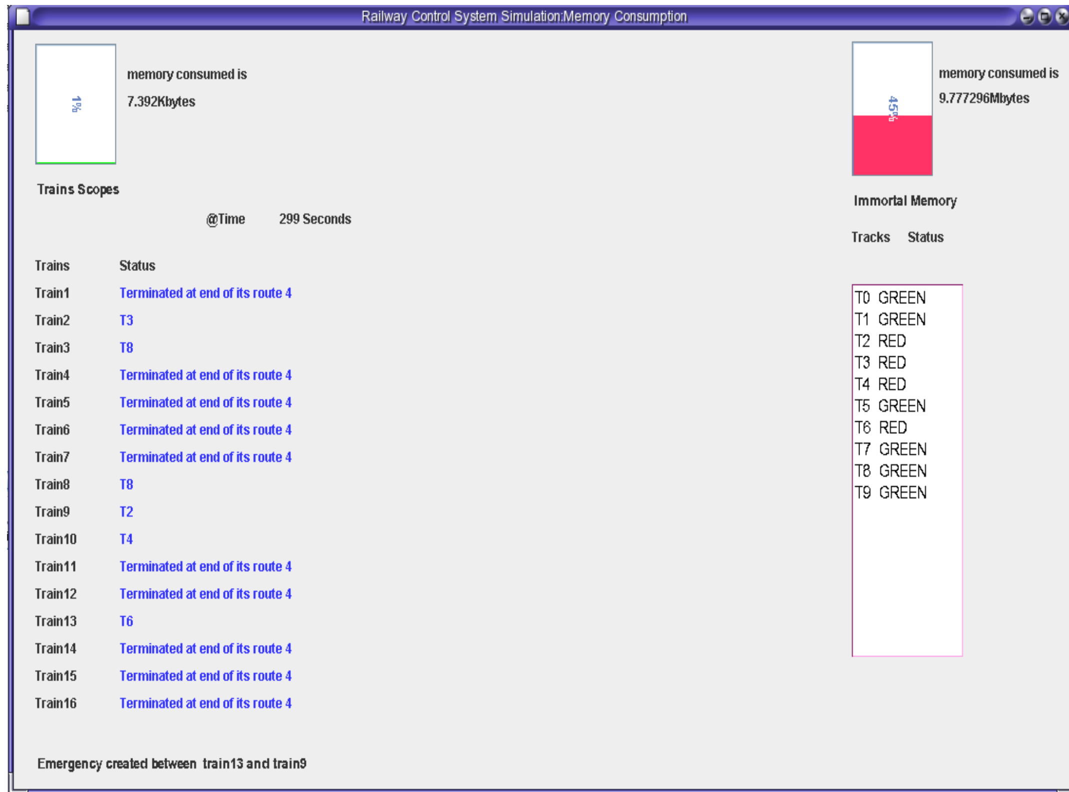

Figure 4.4: Simulation GUI element at 299 seconds (Design 1)

Figure 4.5 shows the simulation at period of 142 seconds where Design 2 was implemented in that run. This screenshot shows more scoped memory consumption at that time than the scoped memory consumption in Design 1.

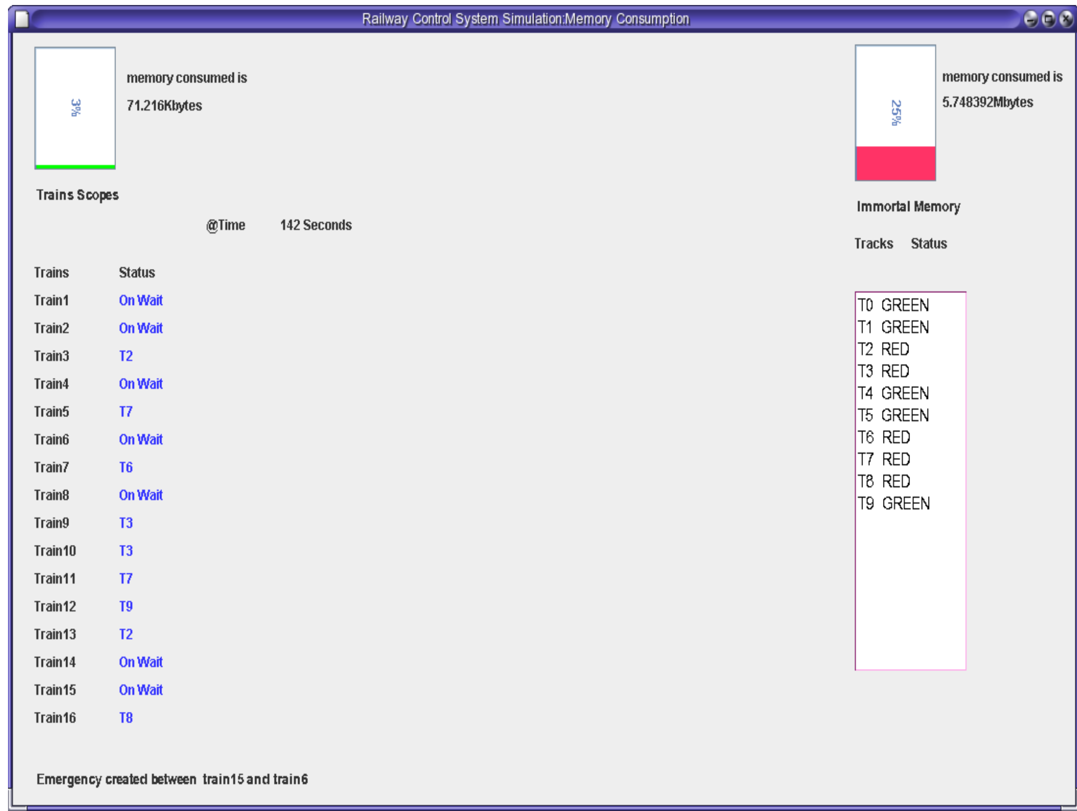

Figure 4.5: Simulation GUI element at 142 seconds (Design 2)

Figure 4.6 also shows the screenshot of the simulation at period of 300 seconds where Design 2 was implemented. The scoped memory consumption in Design 2 at 300 seconds was (138.36Kbytes) compared with scoped memory consumption at similar time in Design 1 was 7.39Kb as shown in Figure 4.4.

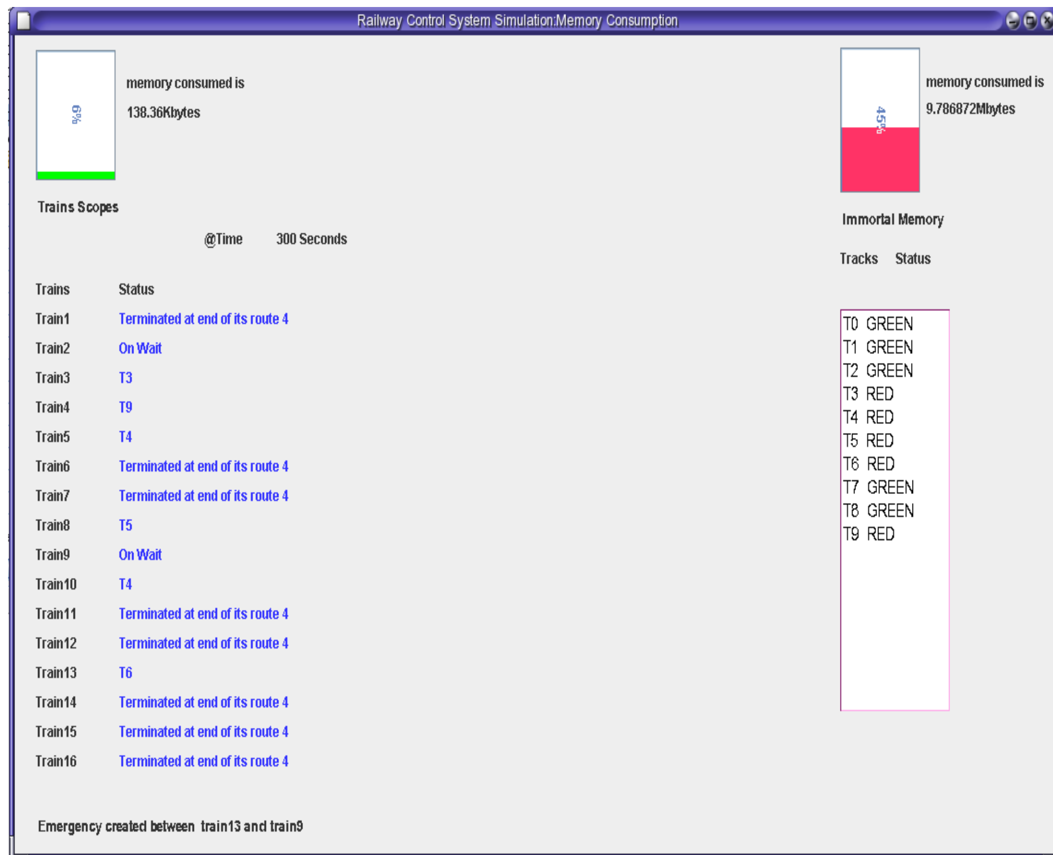

Figure 4.6: Simulation GUI element at 300 seconds (Design 2)

The other part of the simulation tool is the console (shown in Figure 4.7). The console shows more detail of the application at runtime and outputs this information into a text file. For instance, the trains changing status over periods of time (lines 9, 10, and 11) and when emergency states occur between trains (line 8) are printed on the console. Memory consumption over periods of time is also displayed. The information provided by the console is recorded for the developer so that they can review this information at a later point. The story-lines in Figure 4.7 maintain the data that will be used by the developer for later analysis. This simulation simulates the events that may occur in the real-world. As seen from Figure 4.7, trains may wait to run on a specific track for other trains when the traffic light is red; for example Train10 is “on wait” status (line 10). Train13 is stopped until Train16 finishes its current track (line 9) since an emergency is created between Train13 and Train16

(line 8); the simulation tool specifies randomly which train should stop and which one should continue running in case of emergencies.

1. **The current Time is 150 seconds**
2. Immortal memory consumption is 5.97Mb
3. Scoped memory consumption is 17.3 Kb
4. ……..
5. **The current Time is 198 seconds**
6. Immortal memory consumption is 7.18 Mb
7. Scoped memory consumption is 13.9 Kb
8. Emergency is created between train13 and train16
9. Train13 has been stopped until train16 finishes its current Track
10. Train10 is waiting until the traffic light sets green
11. Train3 is waiting until the traffic light sets green
12. ………
13. **The current Time is 347 seconds**
14. Immortal memory consumption is 10.89 Mb
15. Scoped memory consumption is 2.94 Kb
16. Train9 Has finished its current route
17. Tain8 Has finished its current route

Figure 4.7: Simulation Console element (Design 1)

4.5 Simulation Analysis

The total memory consumption of all scoped memories created in each design was measured over time. The simulation was run for 350 seconds (approximately 6 minutes) at which point all trains had finished their routes and the application had terminated.

Figure 4.8 shows the immortal memory consumption of Designs 1, 2, 3, 4 and 5 from 1 second to 350 seconds after which the application terminated. Consumption increased from 2.7Mb to 11.2Mb for Designs 1 to 5. The increases are almost identical for all scoped memory design models except for Design 4, which ran for a relatively longer time than the other designs. The difference in termination times for all designs is small since the execution time of the simulation relies on the random status of trains and tracks.

The immortal memory consumption gradually increased while the application was running. The increase in immortal memory was due to temporary objects allocated periodically by the Monitor Thread and Control Thread which both run in immortal memory. For instance, the Monitor Thread allocated string objects to print current memory consumption; after 350 seconds, all trains had finished their routes and no more temporary objects were then allocated by the Monitor Thread. The immortal memory consumption started to flatten after 350 seconds.

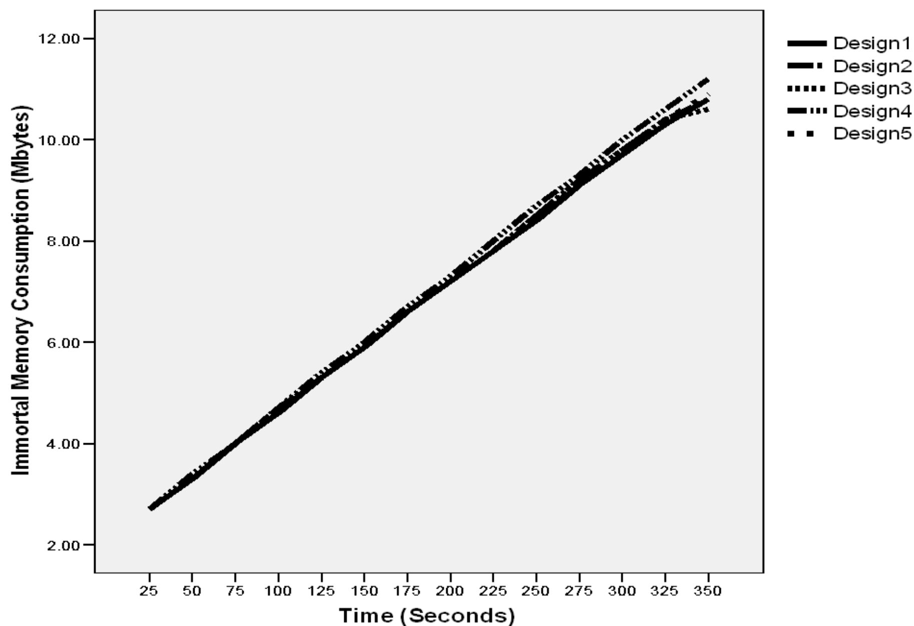

Figure 4.8: Immortal memory consumption in Designs 1, 2, 3, 4 and 5

Table 4.3 presents the summary data for immortal memory consumption of all designs. The Track object consumed non-trivial amounts of memory inside the immortal memory area. The maximum value of the immortal memory reached over time for Designs 1, 2 and 5 was 10.8Mb. It is relatively higher in Design 4 since its execution time is longer than the execution times of remaining memory design models. Since threads that run in immortal memory are the same in Designs 1 to 5, immortal memory consumption for all of them is almost identical.

	Minimum	Maximum	Mean	Std. Deviation
Design1	2.70	10.80	6.84	2.64
Design2	2.70	10.80	6.86	2.67
Design3	2.70	10.60	6.86	2.64
Design4	2.70	11.20	7.00	2.75
Design5	2.70	10.80	6.86	2.66

Table 4. 3: Summary Data for Immortal consumption

Figure 4.9 shows the total amount of all scoped memory areas consumption during Design 1 that assigns one scoped memory for each Train Thread, one scoped memory for each Emergency Thread and one scoped memory for the Train Status Table. The maximum value of consumption in Design 1 was 19Kb at 100 seconds. After that point, total consumption starts to fall when the Train Threads start to terminate and exit their specific scoped memory at different times; scoped memory areas will be freed at different times and total consumption will degrade until reaching zero. Memory consumption falls at a relatively slow rate after 100 seconds, a feature not observable in any of the other four designs. There is a simple explanation for this feature. In Figure 4.9, since each Train Thread runs in a different scoped memory area (which will be freed immediately after that train itself terminates); there is a staggered freeing up of memory dictated by when each train terminates.

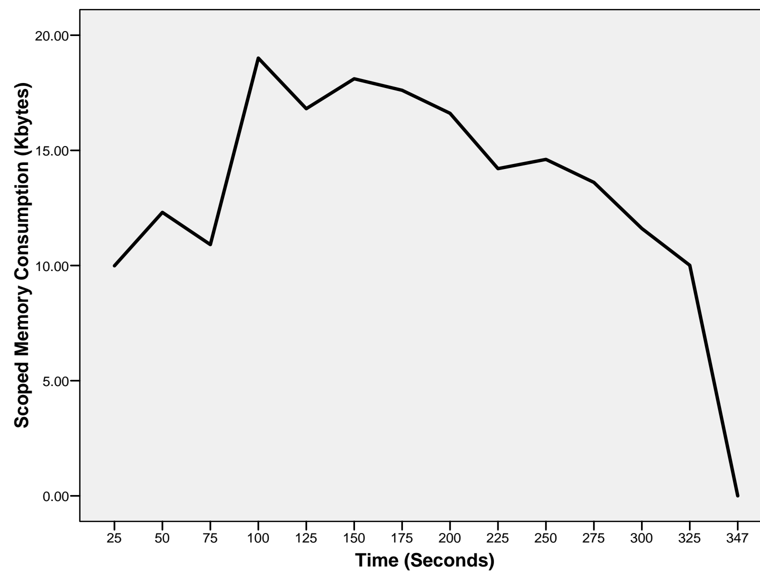

Figure 4.9: Scoped memory consumption in Design 1

Figure 4.10 shows the total amount of the scoped memory consumption for Design 2; this is the same as for Design 1 except that in Design 2 there is just one scoped memory for all Train Threads in addition to one scoped memory for each Emergency Thread and one scoped memory for the Train Status Table. The maximum value of consumption was 155.81Kb at 325s for this design. The memory consumption of scopes over time was greater than that for Design 1 since, in Design 2, all Train Threads were allocated into one scoped memory area which tended to create more new objects that were not freed until all the trains had finished their routes; in Design 1, each train was assigned to one scope which was freed immediately after the specific train finished. The sudden fall in the memory consumption occurs because there is only one scoped memory for all Train Threads and this scoped memory is not freed until all trains terminate; in this design, the last train terminates at approximately 341 seconds.

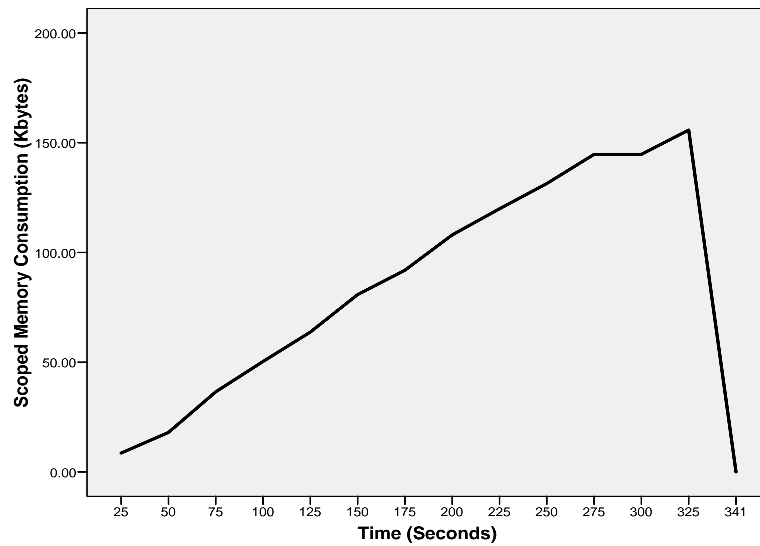

Figure 4.10: Scoped memory consumptions in Design 2

Figure 4.11 shows the total amount of scoped memory consumption for Design 3, characterized by one scoped memory for all Train Threads and all Emergency Threads; there is one scoped memory for the Train Status Table. Considerable growth in memory consumption is evident in Design 3, since one scoped memory model is allocated for all Train Threads and Emergency Threads in the application and scoped memory will not therefore be freed until all Train/Emergency Threads finish executing inside it. The maximum value of memory consumption reached 224Kb.

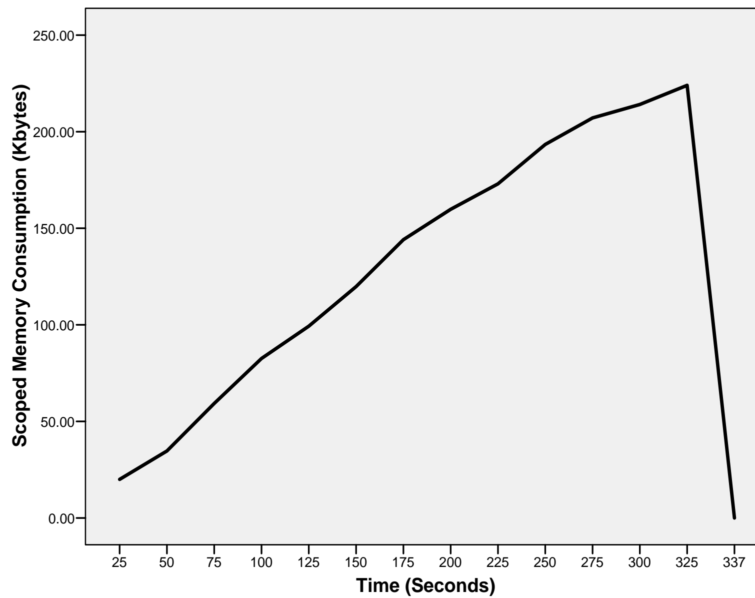

Figure 4.11: Scoped memory consumptions in Design 3

Figure 4.12 shows the total amount of scoped memory consumption over time for Design 4. This design is characterized by one scoped memory for all Train Threads and the Train Status Table; there is one scoped memory for each Emergency Thread. The resulting memory consumption reaches a maximum value of 1487.40Kb. It would seem, at face value that the poorest design memory model is Design 4 where all Trains Threads are running in one scoped memory area and the Train Status Table will also be allocated into the same scope every time the Monitor Thread executes. This is why a consistent increase in memory consumption is observed. A sudden fall in total scoped memory consumption occurs at 351 seconds, since this scoped memory area will be freed immediately after all Train Threads terminate and no more memory will be allocated to store the Train Status Table.

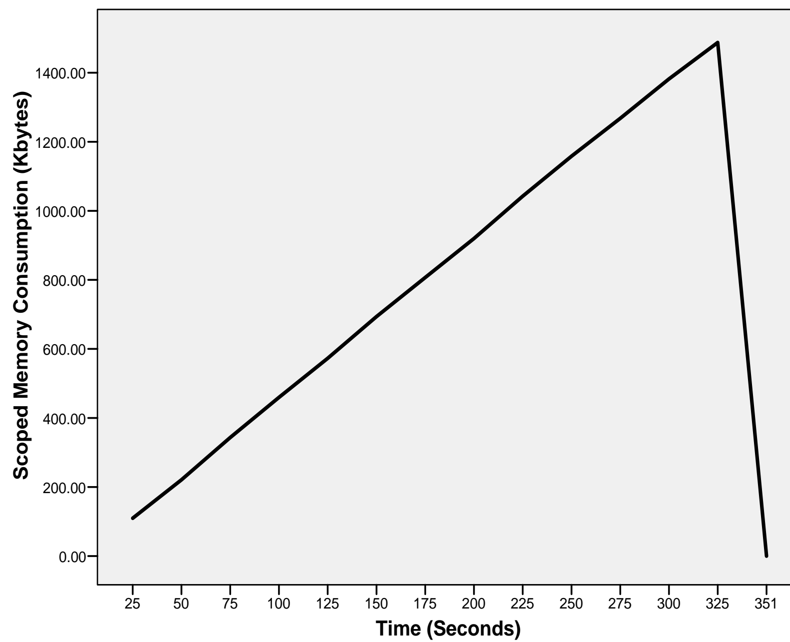

Figure 4.12: Scoped memory consumption in Design 4

Figure 4.13 shows the total amount of the scoped memory area consumption over time for Design 5. For this design, there is one scoped memory for all Train Threads and one scoped memory for Emergency Threads and the Train Status Table. The maximum value of memory consumption for this design is 153Kbytes. This design is similar to Design 2 in memory consumption since both designs have one scoped memory for all Trains Threads.

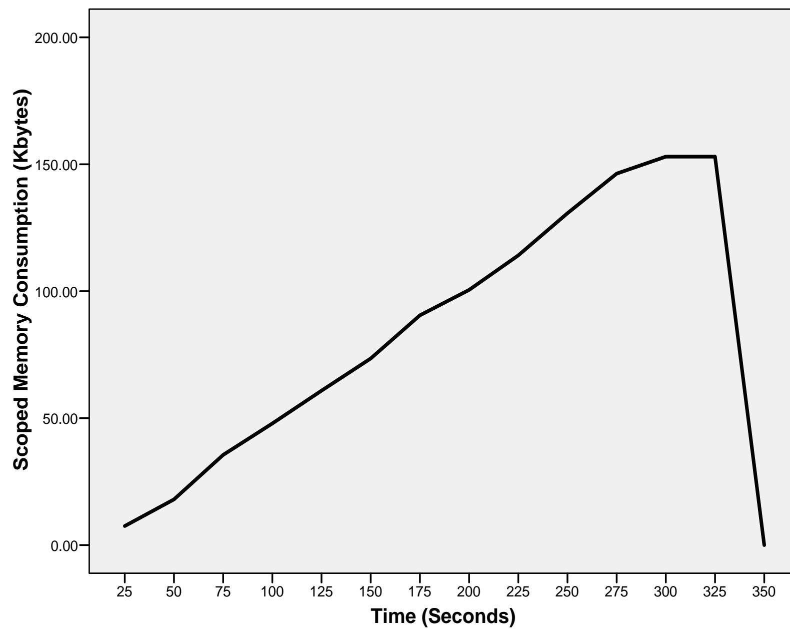

Figure 4.13: Scoped memory consumption in Design 5

Table 4.4 presents summary data (maximum, minimum, median, mean and standard deviation (SD)) values for the five designs for the total scoped memories consumption of all designs. As indicated by Figure 4.12, Design 4 is clearly the most expensive in terms of its memory consumption. Designs 1, 2, 3 and 5 are comparable in terms of their memory consumption.

	Minimum	Maximum	Mean	Std. Deviation
Design 1	.00	19.01	13.2417	4.87027
Design 2	.00	155.81	82.4560	53.76607
Design 3	.00	224.01	123.6646	75.76341
Design 4	.00	1487.40	747.3886	482.96625
Design 5	.00	153.01	80.8146	54.22990

Table 4.4: Summary Data for Scope consumption

The preferred design scoped memory model and that showing the best performance is Design 1 where one scoped memory area is assigned for each Train Thread and freed when the related thread finishes its execution. The maximum value of the memory

consumption of Design 1 reaches 19Kb. Running periodical threads in immortal memory needs to be taken into consideration, since temporary objects that might be created by these periodic threads have to be allocated in immortal memory.

As a recommendation, developers should use scopes to allocate temporary objects that will not be used in the next iteration of the thread. Developers should also be aware when choosing the number of scopes in their memory model, the higher the number of scopes, the less the footprint. However, increasing the number of scopes impacts throughput. Execution time of the application will generally increase and does not always bring better a memory footprint as noted in the differences between Designs 3, 4, and 5. There, the number of scopes was the same (two scopes in each); however, Design 3 was superior in terms of its memory footprint. Allocating the right objects/threads into the right scopes is therefore important for achieving an efficient memory design model.

4.6 Guidelines for Using Scoped Memory in RTSJ

Through the development of the railway case study using RTSJ and its memory model, it has been demonstrated that scoped memory is not a trivial approach to implement since reference rules complicate that process. It is mandatory to place some objects in immortal memory to enable communication between scopes. If scopes are not siblings, references between them are not allowed; to reference a shared object by objects created in these scopes, the shared object should be allocated into immortal memory where all scopes can reference it. Guidelines for using scoped memory in RTSJ are summarized as follows:

1. Developers should avoid allocating string objects into immortal memory, especially if those string objects change their current states over time, since this leads to a constant increase in immortal memory consumption. We note that through experimentation when updating the status of the GUI objects, the GUI component is allocated in immortal memory since this will be alive until the application terminates. Finding design patterns to decrease for immortal memory consumption is a necessity.

2. Developers should use nested scopes to allocate short lifetime objects (such as a scoped memory for the Train Status Table).

3. Developers should allocate code that runs periodically in a real-time thread in scoped memory (such as the Train Status Table).

4. Developers should allocate real-time threads that have relatively short lifetimes into scoped memory areas (such as Train Threads).

5. Developers should bear in mind that the default memory context of any real-time thread is immortal memory.

6. Developers should recycle Runnable objects rather than creating them every time a thread enters a scoped memory area.

7. Threads that run until the application terminates should be allocated into immortal memory; however, if threads have to run some code periodically, then the code that runs periodically should be allocated into a scoped memory area.

4.7 Conclusions

Simulating safety-critical real-time systems enables the testing of the behaviour of systems before installing them in the real-world. This chapter introduced a railway case study for RTSJ run on the RTS2.2 virtual machine which combines multi-threading and scoped memory model implementations. It is the first empirical study using RTSJ in the analysis of scopes and exploration of criteria for object allocation therein. A simulation tool for a real-time Java application was presented which can be abstracted further in future to a wide spectrum of real-time applications. The focus was on testing the memory consumption of a specific case study of a railway control system. Different scoped memory design models were implemented to measure memory consumption for each over time. The simulation provided runtime information about memory consumption of different scoped memory models which can assist in selecting the most appropriate scoped memory design model for achieving a minimal memory footprint.

Memory consumption of five possible designs for scoped memory models was measured. Results showed that the memory design model that had the greater number of scopes achieved the best memory footprint. However, number of scopes did not always indicate a 'good' memory footprint; choosing the right objects/threads to be allocated into scopes is an important factor to be considered. Recommendations and guidelines for developing RTSJ applications that use a scoped memory model were presented in this chapter. Finally, the next chapter introduces and discusses a solution to stop immortal memory increasing while the application runs.

Chapter 5: Slicing and Patterns for RTSJ Immortal Memory Optimization

5.1 Overview

In the previous chapter, the railway control case study showed the complexity of using the new RTSJ memory model and the space overhead occurred in immortal memory. The case study illustrated how simulation of critical safety real-time applications in Java can be used to investigate the implementation of possible scoped memory design models and their memory consumption in multi-threaded environments. Results showed that a memory design model with a higher number of scopes achieved the least memory footprint. However, the number of scopes *per se* did not always indicate a satisfactory memory footprint; choosing the right objects/threads to be allocated into scopes was an important factor to be considered. The case study showed a constant increase in immortal memory at runtime in all of the memory design models implemented in the case study.

This phenomenon motivated the work presented to define objects which cause immortal memory space overheads and eliminate constant increases in immortal memory. In this chapter, dynamic code slicing is employed as a debugging technique to explore constant increases in immortal memory. Two programming design patterns are presented for decreasing immortal memory overheads generated by specific data structures. Experimental results showed a significant decrease in immortal memory consumption at runtime. This chapter thus makes two contributions:

1. It motivates the use of a dynamic slicing technique to debug RTSJ code and to define the objects that specifically affect immortal memory constant increase at runtime.

2. It introduces two programming design patterns to decrease immortal memory consumption when Hashtable data structures are manipulated inside immortal memory.

The remainder of this chapter is organized as follows: The methodology of this work is proposed in Section 2. The new programming design patterns are then explained in Section 3. Section 4 discusses the experimental results and the outcomes of the applied methodology and design patterns. Finally, Section 5 concludes the chapter.

5.2 Methodology

As seen in Chapter 4, the Main thread, Control thread and Monitor thread are allocated in immortal memory as they all run until the application terminates. Since the Track object is a fixed size object and is accessible by all threads that run in different scoped memory area, it is allocated in immortal memory and no reference violation at runtime occurs. All remaining threads and objects (Train Threads, Emergency Thread and the Train Status Table) are allocated into scoped memory areas since they have different lifetimes and a better footprint is achieved.

To uncover the reasons behind constant increases in immortal memory, verification and debugging techniques are required. Since some of the objects might have been generated through native methods, it is difficult to determine statically from the code the new objects allocated into immortal memory at runtime. Therefore, program

slicing could potentially be used as one of the techniques to debug and eliminate the problem and to simplify the testing approach (Harman and Danicic, 1995).

Program slicing is "a reverse engineering technique consisting of decomposing a program into slices according to certain criteria (e.g., fragments of source code that use a specific program variable)" (Pérez-Castillo et al., 2012). It is one of the techniques used in software engineering for maintenance purposes such as debugging, program understanding, testing, tuning compilers, program analysis and reverse engineering (Gallagher and Lyle, 1991, Tip, 1995). Literally speaking, a program slice (Weiser, 1979) is a set of all program statements and predicates that might affect value of a variable (v) at a program point (p). Figure 5.1 shows an example of slicing on variable (product) at line 10 of the program (Tip, 1995). In Figure 5.1 part (a), the original code is presented. To analyze how the variable product can be affected in the program, a sliced code (part (b) of Figure 5) is created which includes all the statements and predicates that might affect value of the variable product. All other statements at lines such as (3, 6, and 90) are removed from the slice since the computations at those code lines are not relevant to the final value of the variable product.

(a)	(b)
(1) read(n);	(1) read(n);
(2) i := 1;	(2) i:=1;
(3) sum := 0;	(3)
(4) product := 1;	(4) product := 1;
(5) **while** i<=n **do**	(5) **while** i<=n **do**
begin	**begin**
(6) sum := sum + i;	(6)
(7) product := product * i;	(7) product := product * i;
(8) i:=i+1	(8) i:=i+1
end;	**end**;
(9) write(sum);	(9)
(10) write(product)	(10) write(product)

Figure 5.1: (a) An example program. (b) A slice of the program criterion (10, product).

A slicing technique was first introduced by (Weiser, 1979) as static slicing based on data flow and dependence graphs. There are two types of slicing – 'static' and 'dynamic'. Static slicing can be produced by collecting information about the program statically such as the structure of the application, number of threads, types of objects, connection between objects, etc (Harman and Hierons, 2001). Dynamic slicing collects information about application behaviour at runtime in relation to a specific user input in addition to the static data of the application (Harman and Hierons, 2001). The notion of dynamic slicing was introduced by (Korel and Laski, 1988) stating that it was impossible to identify dynamic objects through static analysis. Dynamic slicing identifies a subset of executed statements expected to contain faulty code (Zhang et al., 2005). It is more useful in OO programs which consist of different types of objects, methods and in multi-threaded programming. In

OO programs, statements in the methods of a particular object that might affect the slicing criterion are identified (object slicing) (Liang and Harrold., 1998). An overview of slicing techniques for OO programs can be found in (Mohapatra et al., 2006).

The case study explained in Chapter 5 is multi-threaded application. Its behaviour at runtime may generate new objects through native methods in immortal memory. Therefore, static analysis is not enough to debug immortal memory consumption at runtime; using dynamic slicing is more suitable to trace the objects/methods that cause an instant increase in the immortal memory. Pan and Spafford, (1992) found that experienced programmers debugged code through four debugging tasks: (1) determining statements involved in program failures, (2) selecting suspicious statements that might contain faults, (3) making hypotheses about suspicious faults (variables and locations), and (4) restoring the program state to a specific statement for verification. In this work, the approach to dynamic slicing is similar to that of Pan and Spafford (1992) which used heuristics for fault localization by defining suspicious statements that caused the software to fail. Two heuristic are used in this work; heuristic 1 (cover all statements in all available dynamic slices) and heuristic 7 (indicate statements with high influence frequency which appear or is executed many times in one dynamic slice) (Pan and Spafford, 1992). Accordingly, to find the statements which impact immortal memory increase, the main focus is on statements that are executed in the immortal memory within periodic threads such as a Control thread and/or within loop structures. Next, code slices are generated to measure the impact of each statement on immortal memory increase.

Code slices are initially allocated in a scoped memory area to monitor any decrease in the immortal memory consumption or to find out whether a reference violation occurs

by new objects created inside that slice. The approach is divided into 5 circular steps which can be repeated to capture the places where immortal memory constant increases occur. Figure 5.2 summarizes the methodology approach. Since two periodic threads run in immortal memory in the case study (the Control and Monitor threads), the debugging techniques were applied on only those two. In the Control thread, the code that most likely generates new objects was sliced; then the slice was placed inside a scoped memory area. Next, the application was executed to measure immortal memory consumption; if an error occurred at runtime inside the sliced code, a reference violation occurred meaning a new object was generated; the code that produced reference errors was removed from scoped memory. Design patterns were created to solve the problem of reference violations and to eliminate the space overhead generated by the newly created objects. If there was no error and the immortal memory decreased, the code was kept inside a scoped memory area.

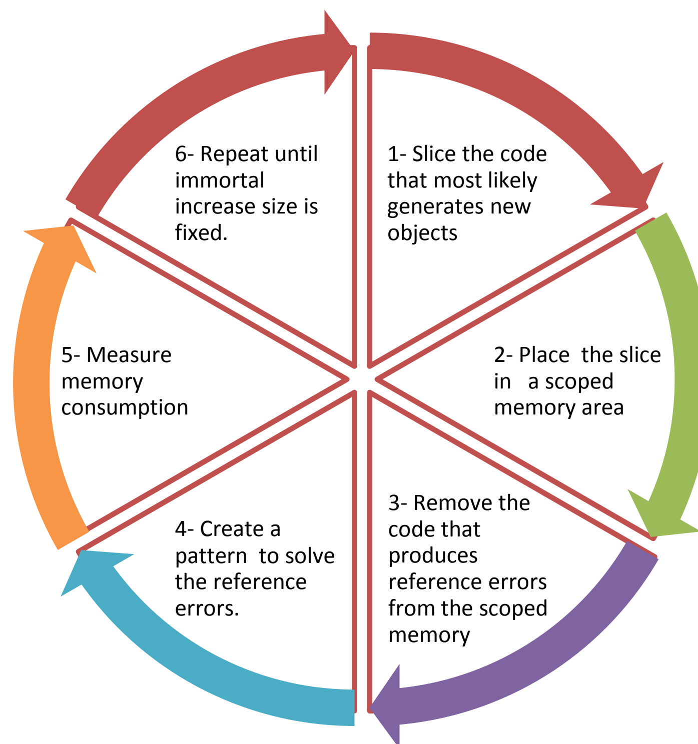

Figure 5.2: The Slicing Methodology.

The steps were repeated until it was no longer possible to either decrease immortal memory or until the size of the immortal memory at runtime was fixed. Through this methodology, the objects that caused immortal memory to increase were identified, namely: a) the String object of the print statement inside the Control and Monitor threads and b) the Hashtable reading and modifying operations inside a loop in the Control thread.

All String objects of the print statements were eliminated by allocating them inside scoped memory areas to be de-allocated immediately after printing string messages. Hence, immortal memory consumption at runtime in the case study decreased by 25%. When using different data objects in Java such as Arraylist, Hashtable, Vector and String, it is important to monitor memory consumption where objects of these data types reside. In the case study, a Hashtable was used to represent the tracks' status at runtime; some of the case study entities need to locate a specific track to update its status according to the emergency state or according to the train threads that run on different tracks. The Hashtable is created in the case study inside immortal memory to be accessible by all objects and threads. A Hashtable is used because it is thread-safe and can be shared between multiple threads; on the other hand, the order of the values in Hashtable is not important. As stated in Strøm and Schoeberl (2012) a full knowledge of the library code is required to prevent the creation of objects in wrong scopes and producing dangling references as a consequence.

In Chapter 3, the impact of scoped memory on execution time and the footprint of an application were explored when different types of objects (Vector, Float, Hashtable, and Integer) and numbers of regions were used. Float objects consumed more

memory and affected the execution time more than other objects. Hashtable was the second worst in terms of both memory footprint and application execution time.

5.3 Immortal Memory Patterns

5.3.1 Hashtable Reading Pattern

To read a value associated with a specific key in the Hashtable, the get(Object key) method can be used. This method returns null if the key does not exist or returns the value of the key if it does. Interestingly, through the slicing approach of the case study, the 'get' method of the Hashtable was tested and, as a result, it was noticed that it generated temporary objects at runtime which, in turn, increased immortal memory consumption. Significant impact on immortal memory consumption occurred when the reading operations took place inside a loop of a periodic thread. This motivated a new design pattern to allocate the slice of code which reads values from the Hashtable in a scoped memory area. Any temporary object generated during the reading operations will be allocated inside scoped memory area and de-allocated once exiting the scoped memory area. However, according to the value read in the Hashtable, the flow of the application outside the scoped memory will change as a result. Communication between the inside of scoped memory area and its outside is *via* a static primitive variable with an 'if'-statement. The if-statement will change the value of the primitive variable according to the value that has been read in the Hashtable. A y object is used as a reference to the value returned by the 'get' method. After that, the same object is not required and will be de-allocated once exiting the scoped memory. To pass the value outside the scoped memory area, another 'if'-statement outside the scoped memory is used to define the application flow. The new design

used is similar to the "execute with primitive return value" design pattern in Rios et al., (2012); however, in this work, new object parameters were not created and instead static primitive types were used; although the variable is allocated in immortal memory, it is smaller than creating a new object that might not be de-allocated until the application terminates. The new design pattern communicates with the outside of the scoped memory area and decreases immortal memory consumption caused by reading operations of the Hashtable. The template of this design pattern is illustrated below:

1. Pattern name: Hashtable Immortal/Scoped-Safe Reading Pattern

2. General context: This pattern is used to allocate the slice of code which reads values from the Hashtable in a scoped memory area. Any temporary object generated during the reading operations will be allocated inside a scoped memory and de-allocated once exiting that scoped memory. If the flow of the application changes according to the value read in the Hashtable, then a primitive static variable will be used to communicate between the scoped memory and its outer allocation context.

3. Motivation for use: to reduce the immortal memory consumption resulting from reading values of the Hashtable keys.

4. Diagram or source code to illustrate general application of pattern

```
1.  Key=IntergerVar;// it can be any data type
2.    Runnable  HashTableRead=new Runnable()
3.      {
4.        public void run()
5.        {
6.          String[]   y=MyHashTable.get(Key);
7.          if ( y[0].equals(Str1))
8.                PrimitiveVariable=1;
9.         else
10.            PrimitiveVariable=2;
11.          }
12.     };
13.
14.    ------
15.
16.    ScopedMemory1.enter(HashTableRead);
17.    if ( PrimitiveVariable==1)
18.     {
19.       //statemnts(A)
20.     }
21.   Else
22.   {
23.   //statemnts(B)
24.   }
```

5. Constraints: It is used only when the Hashtable is allocated in immortal memory or in a parent memory of the scoped memory area.

6. Related patterns or anti-patterns: This pattern is derived from "Execute with Primitive Return Value" Design Pattern in Rios et al., (2012); the main aim of that pattern is to communicate between scoped memory and its outer context using input and output objects to pass the information In this work, input and output objects are not created and, instead, a static primitive variable is used; although the variable is allocated in immortal memory, it is smaller than creating a new object that might not be de-allocated until the application terminates. On the other hand, the new design pattern proposed in this work achieves two aims; it communicates with the outside of the scoped memory area and decreases immortal memory consumption caused by the reading operations of the Hashtable.

Figure 5.3 shows the implementation of the design pattern which reads the Hashtable value of a key T and passes the result in a primitive variable to the outside of the scoped memory area. In line 5, a temporary string array y is used to refer to the returned value of a specific key T through the ‘get’ method. The key here represents the track name in the case study. The y object is checked in line 6 and a new value is assigned to the primitive variable in lines 7 and 8 to be checked outside of the scoped memory area (line 13).

```
1.   Runnable  DesignPatternToRead=new Runnable()
2.   {
3.     public void run()
4.     {
5.       String[]   y=Main.Tracks.get(T);
6.       if (  y[0].equals("OFF"))              Main.PrimitiveVariable=1;
7.       else if ( y[0].equals("ON"))           Main.PrimitiveVariable=2;
8.             else if ( y[0].equals(""))        Main.PrimitiveVariable=0;
9.     }
10.  };
11.  -------
12.  ScopedMemory1.enter(DesignPatternToRead);
13.  if ( Main.PrimitiveVariable==1)
14.  {
15.    // Update the Track status
16.  }
17.  -------
```

Figure 5.3: Design Pattern 1 (Reading Hashtable Values)

5.3.2 Hashtable Modifying Pattern

In the Control thread, modifying the values of existing keys of the Hashtable (the Track object which is allocated in immortal memory) frequently at runtime is required. However, the new value objects used to modify the Hashtable keys are previously created in immortal memory. One method of modifying the value of a Hashtable’s key is to use the ‘put’ method. The put(K key, V value) method is used to map the specified key to the specified value in the Hashtable. The ‘put’ method returns the old value of the key if the key exists, or null if a new key is used.

When the code was sliced and executed inside a scoped memory area to decrease immortal memory consumption, a 'put' method statement was included in that slice. Subsequently, a throw-boundary error caused by the 'put' method was received even though no new keys were passed - the Hashtable size was not increased; only existing keys with previously created value objects were passed to the 'put' method of the Hashtable periodically. In other words, no new key or value objects were created when the 'put' method was used to modify key values. (A throw-boundary error occurs when a violation of reference rules takes place such as reference from immortal memory to a scoped memory.)

The 'put' method appears to generate unknown objects even though no new keys or values are added to the Hashtable; consequently, there will be a reference violation as the Track object (Hashtable) allocated in immortal memory will reference unknown objects allocated in a scoped memory area. One important question here is how to modify the Hashtable values allocated in immortal memory without increasing immortal memory consumption?

The template of the proposed new design pattern is explained below:

1. Pattern name: Hashtable Scoped-Safe Modifying Pattern

2. General context: To periodically modify the Hashtable values allocated in immortal memory using previously created objects in immortal memory without increasing immortal memory consumption.

3. Motivation for use: The 'put' method appears to generate unknown objects even though no new keys or values are added to the Hashtable. The aim is to reduce immortal memory consumption resulting from the modification of the Hashtable key values using previously created objects in immortal memory.

4. Diagram or source code to illustrate general application of pattern:

```
1.  R = value_of_any_data_type
2.  Key=IntergerVar;// it can be any data type
3.  Set<Entry<String, String[]>> entries  =Main.
    MyHashTable.entrySet();
4.  Runnable DesignPatternToModify=new Runnable()
5.  {
6.     public  void run()
7.      {
8.        for(Entry<String, String[]> ent: entries)
9.           if (ent.getKey().equals(key))
10.            {
11.             ent.setValue(R);
12.             break;
13.            }
14.       }
15.   };
16.   ------------------
17.  ScopedMemory2.enter(DesignPatternToModify);
18.   ------------------
```

5. Constraints: The limitation of this design pattern is that it modifies values of existing keys of Hashtable with only previously allocated object values in immortal memory.

6. Related patterns or anti-patterns: N/A

Figure 5.4 shows the implementation of the design pattern which uses a set of entry objects to modify a Hashtable's value of an existing key T without using the 'put' method. The new value is passed by a parameter R which is previously allocated into immortal memory. The set interface in Java is a collection which contains no duplicate elements. The entry object is a map entry (key-value pair) which links to one key in the Hashtable. The Hashtable.entrySet method returns a collection-view of the map so that any changes to the set are reflected in the Hashtable and *vice versa*. Iterating over the elements of the set generates new temporary entry objects; however, by using the design pattern entry, objects will only be allocated inside scoped memory and will be de-allocated once exiting that scoped memory. This, in

turn, has no effect on immortal memory and, as a result, memory consumption of the immortal memory decreases. Testing this design pattern inside a scoped memory area did not throw a boundary error as it occurred when the 'put' method was used earlier.

```
1.  Set<Entry<String, String[]>> entries  =Main.Tracks.entrySet();
2.  Runnable DesignPatternToModify=new Runnable()
3.  {
4.     public  void run()
5.      {
6.         for(Entry<String, String[]> ent: entries)
7.            if (ent.getKey().equals(T))
8.              {
9.               ent.setValue(R);
10.              break;
11.              }
12.        }
13.    };
14.   ------------------
15.  ScopedMemory2.enter(DesignPatternToModify);
16.    ------------------
```

Figure 5. 4: Design Pattern 2 (Modifying Hashtable Values)

The limitation of this design pattern is that it modifies values of existing keys of Hashtable with previously allocated object values in immortal memory. Appendix B shows the original code, two slices that have been indentified according to dynamic (and static) slicing and the modified code after implementing the two design patters

5.4 Discussion

The experiments ran on the same platform used in Chapter 4 (see Table 4.2). Since the case study is a multi-threaded application and there are 16 train threads running at the same time on different tracks, the execution time and memory consumption at runtime may differ slightly from one run to another. Repeating the experiments is needed where most non-determinism occurs in the experiment (Kalibera and Jones, 2013). Since compilation is not random in the case study (it is deterministic and performance does not depend on code layout) there is thus no need to repeat it to get reliable results. However, the start-up of a VM execution includes some random

variation due to input/output bound and scheduling order, in which case VM executions must be repeated. Consequently, and in order to obtain reliable results, the case study was executed many times until insignificant variation in memory consumption data (0.007Mbytes) was reached. The execution time is not the main focus in this work and it only considers variation in the memory consumption. To avoid jitter (i.e., fluctuation in execution times which may occur while loading and initializing classes at runtime), the initialization time compilation mode (ITC) was used to compile and initialize classes at the virtual machine start-up time. After implementing Design Pattern 1 (reading from Hashtable) and Design Pattern 2 in the case study and running the code in scoped memory, immortal memory consumption decreased by 50%.

Table 5.1 shows the results of the experiment when three versions of the case study were implemented and compared. The first version is when Design Patterns 1 and 2 were not used; the second version is when only Design Pattern 1 was implemented and the third version is when Design Patterns 1 and 2 were implemented. The execution times of the case study fluctuate over runs; however, on average, the version that implemented Design Patterns 1 and 2 out-performed the old version in terms of immortal memory consumption. This is despite execution time slightly increasing according to the overhead occurred by entering the scoped memory area through Design Patterns 1 and 2, periodically. Results show a decrease in immortal memory consumption after implementing Design Pattern 2. The decrease is not significant (0.091Mb) and in different situations where more frequent modifications of the Hashtable's values occur inside immortal memory it may well be worth exploring Design Pattern 2 further. It is noticeable that Design Pattern 1 has decreased immortal memory significantly, in other words, Hashtable reading

operations consumed greater amounts of immortal memory at runtime than Hashtable modification operations. As a suggestion from this work, the enter method should be improved to return a value e.g., a Boolean value to pass the result of the code generated in a scoped memory without the need to create a primitive variable outside of the scoped memory area

Figure 5.5 shows the immortal memory consumption over 10 runs before and after implementing both Design Patterns 1 and 2. Figure 5.6 shows the impact of implementing Design Pattern 2 on the immortal memory consumption over 10 runs.

	Before Implementing Design Patterns 1 and 2	***After Implementing Design Pattern 1***	***After Implementing Design Patterns 1 and 2***
Immortal Memory (Mb)	*7.9*	4.205	4.114
Execution Time (sec)	340.5	347.5	348.4

Table 5.1:. Before/After Implementing Design Patterns 1 and 2.

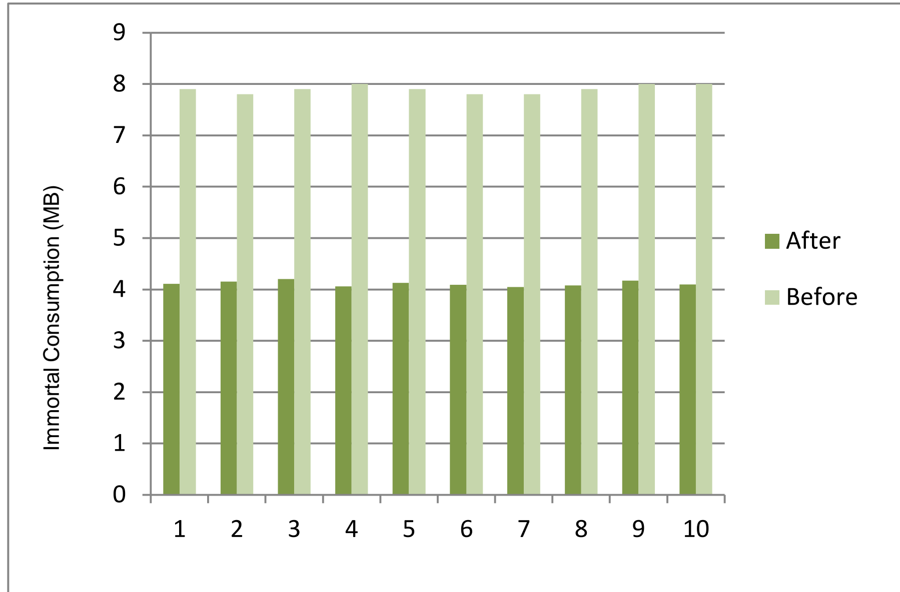

Figure 5.5: Before/After Implementing Design Patterns 1 and 2

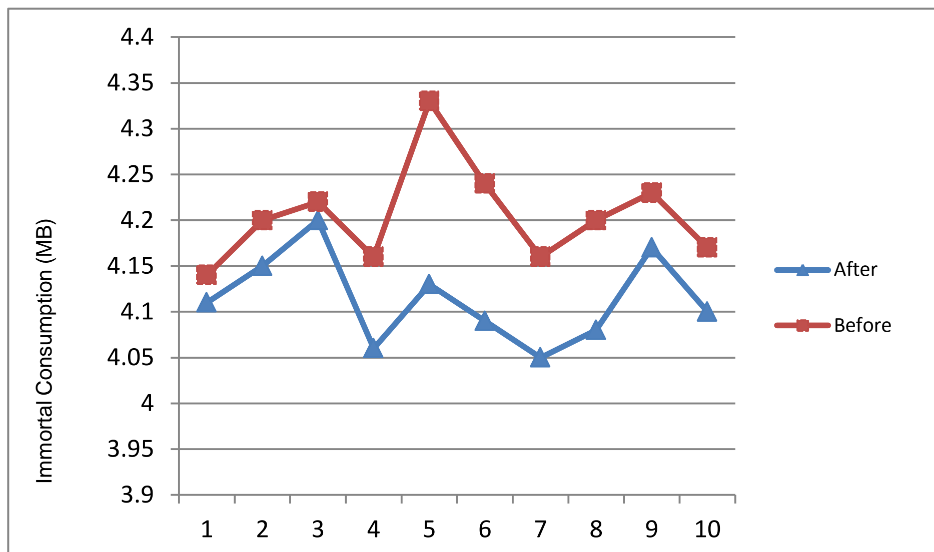

Figure 5. 6: Before/After Implementing Design Pattern 2

5.5 Summary

In this chapter, the focus was to decrease immortal memory consumption at runtime. Code slicing was used as a debugging technique to find the reasons behind immortal memory constant increases in an RTSJ case study. Two main causes were identified:

the String object of the print message and Hashtable read/modify operations. Print message statements were executed inside a scoped memory area which reduced immortal memory consumption. Two design patterns were proposed to decrease immortal memory overheads generated by Hashtable reading /modifying operations. Experiments showed new aspects of dealing with Hashtable and by using new design patterns a significant decrease in immortal memory consumption at runtime was achieved. Although the new design patterns are specific to Hashtable, they provide an insight into how to solve allocation problems with other data structures such as Vector and ArrayList when using an immortal and scoped memory model. In terms of future work, different data structures will be studied to analyze their behaviour at runtime when immortal memory and a scoped memory model are used.

Chapter 6: Conclusions and Future Work

Programming languages use different memory management models. A Static memory management model allocates variables at specific memory locations; there is therefore no change in the memory footprint at application runtime. However, a dynamic memory model allocates and de-allocates objects at application runtime, so the memory footprint is changed constantly.

Java uses garbage collection techniques to manage the memory dynamically and automatically. Hence, developers are not involved in the allocation and de-allocation process. Garbage collection interrupts the application several times to reclaim objects that are not in use by the application to free memory space. However, in real-time systems this approach is not recommended as it may delay the application and cause real-time events to miss their deadlines. The Java Community Process (JCP) proposed the real-time specification of Java (RTSJ) introducing a new semi-automatic memory management model which includes scoped and immortal memory. In addition to the heap memory, there is only one immortal memory and one or more scoped memory areas in real-time Java applications. Scoped and immortal memory areas are not subject to garbage collection and therefore no delays or interruptions by the garbage collection process occur. Developing RTSJ applications using scoped and immortal memory model needs significant effort by the developers and case studies of the use of this memory model are not widely available in the literature. On the other hand, developing real-time Java case studies helps developers to understand the different variables of this memory model. Some design patterns and guidelines are necessary for developers to simplify the process of real-time applications that use scoped memory approach.

This chapter discusses the Thesis conclusions and presents contributions and future research areas. Section 6.2 summarizes the findings of each chapter of this thesis. Section 6.3 explains how the research conducted in this thesis meets its objectives. A summary of the Thesis contributions is then presented in Section 6.4. Section 6.5 identifies the research limitations and, finally, Section 6.6 points to future research ideas.

6.1 Research Summary

The research presented in this Thesis aimed to simplify and improve scoped and immortal memory development in real-time Java applications.

Chapter 1 gave an overview of the Thesis research topic and highlighted the motivation of this research. A set of research objectives were identified to fulfill the research aim. The Thesis main contributions were introduced.

Chapter 2 reviewed previous research and state of art issues related to the scoped and immortal memory area in RTSJ implementations. The scoped and immortal memory model was explained in detail. Problems and solutions along with the benchmarks used to evaluate this model were also provided. Most of the research in RTSJ scoped memory has focused on two important issues. First, decreasing the impact of reference checks and secondly, 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) the variety of

real-time benchmarks that cover more aspects of scoped memory model, c) tools to decrease the difficulty of developing real-time Java applications using a scoped memory model, and d) design patterns to simplify the development process and decrease the impact of using scoped and immortal memory on application execution time and space overheads.

Chapter 3 presented an empirical study scoped memory in Sun RTSJ Implementation. The impact of scoped memory areas on execution time of RTSJ software was investigated. Sample RTSJ code was executed with different numbers of un-nested and nested scoped memory areas. Results showed that increasing the number of scoped memory areas did lead to higher execution times. It was therefore important to find the optimal number of scoped memory areas. Additionally, the developer has to use nesting scope techniques carefully and maintain the trade-off between the pros and cons of using nested scoped memory areas. The overheads of entering and exiting active and non-active scoped memory areas were also presented. Results showed that entering/exiting active scoped memory scoped memory areas had lower execution time overheads than entering non-active ones. Allocating different data objects in scoped memory areas had different impacts on execution time and memory space; therefore, choosing the right data objects and scoped memory size had an effect on the efficiency of the scoped memory model.

Chapter 4 presented a simulation of a railway control system executed on the Sun RTS2.2 virtual machine. It illustrated how simulation of critical safety real-time applications in Java could be used to investigate the implementation of possible scoped memory design models and their memory consumption in multi-threaded environments. The simulation would help a developer to compare and choose the most appropriate scoped memory design model that achieves the least memory

footprint. Results showed that the memory design model with a higher number of scopes achieved the least memory footprint. However, the number of scopes *per se* does not always indicate a satisfactory memory footprint; choosing the right objects/threads to be allocated into scopes is an important factor to be considered. Recommendations and guidelines for developing RTSJ applications which use a scoped and immortal memory model were also presented in this chapter. Developers should avoid allocating string objects into immortal memory especially if those string objects change their current states over time. Using nested scopes is necessary to allocate short lifetime objects. Allocating code that runs periodically in a real-time thread in scoped memory would decrease the impact of memory space overhead. Developers should allocate real-time threads that have relatively short lifetimes into scoped memory areas to ensure any unexpected allocations would be reclaimed automatically after the thread finished its execution. Developers should bear in mind that the default memory context of any real-time thread is immortal memory. Developers should recycle Runnable objects rather than creating them every time a thread enters a scoped memory area. Threads that run until the application terminates should be allocated into immortal memory; however, if threads have to run periodically, the code that runs periodically should be allocated into a scoped memory area.

Chapter 5 provided a new approach for assisting developers in debugging and optimizing scoped and immortal memory implementation. This was motivated by the immortal memory increase encountered in the case study. A dynamic code slicing approach was proposed as a debugging technique to explore constant increases in immortal memory in the case study. The main causes of immortal memory increase were identified. Two programming design patterns were presented for decreasing

immortal memory overheads generated by using Hashtable data structures. Experimental results showed a significant decrease in immortal memory consumption at runtime.

6.2 Research Objectives Re-visited

The main aim of the Thesis was to optimize the use of scoped and immortal memory in real-time Java applications. This section shows how this research successfully achieved its objectives.

Objective 1: ‘to provide state of art issues on the use of scoped memory in real-time Java and discuss the current solutions and challenges to generate a set of research questions’. The first objective was achieved in Chapter 2 by reviewing the literature on using scoped and immortal memory.

Objective 2: ‘to provide an empirical study on the use of the scoped and immortal memory model and its impact on the memory space and execution time of the application’. This objective was achieved in Chapter 3 by experimenting with the impact of using scoped memory on execution time and space overheads of the application. Different data types, allocation sizes, number of scoped memory areas, level of nesting and entering/exiting active/non-active scoped memory area’s features were tested.

Objective 3: ‘To develop a real-time Java case study which uses scoped and immortal memory model in a multi-threading environment where dynamic allocations of objects takes place constantly’. This objective was achieved in Chapter 4 by developing a railway case study and experimenting with different scoped memory models. The simulation tool developed measured the memory consumption and the

execution time of the application. The case study showed possible development pitfalls which may lead to memory leaks.

Objective 4: 'To provide debugging techniques which help in decreasing the overheads of using the scoped and immortal memory model by implementing programming design patterns and evaluating their outcomes'. This objective was achieved in Chapter 5 by proposing a dynamic slicing approach to identify objects that cause the immortal memory increase and providing two design patterns to help decrease the immortal memory footprint.

6.3 Summary of Research Contributions

The main research contributions are summarized as follows:

1. A survey of state of art issues of the new memory model introduced by RTSJ highlighting the issues (time overheads, space overhead, development complexity) and the current solutions (assisting tools, separation memory concerns from program logic, design patterns and components). It also categorized the benchmarks, where they have been used and why they have been used in the research. The survey ended with potential research directions that help to simplify and optimize the use of a scoped and immortal memory model in RTSJ applications.
2. Studying the impact of using scoped memory on execution time and memory space of the application when different data types are allocated into scoped memory areas and when different scoped memory numbers and nesting are used. A comparison between entering and exiting times of active and non-active scoped memory area was introduced.

3. Introducing an additional RTSJ case study which integrates scoped and immortal memory techniques to apply different memory models.
4. Development of a simulation tool of a real-time Java application which is the first in the literature that shows scoped memory and immortal memory consumption of an RTSJ application over a period of time.
5. An implementation of dynamic slicing technique to debug RTSJ code and to define the objects that specifically affect immortal memory constant increases at runtime.
6. Proposition and validation of two programming design patterns to decrease immortal memory consumption when Hashtable data structures are manipulated inside immortal memory.

6.4 Research Limitations

This section identifies a set of research limitations encountered and suggests a set of complementary future work to address them.

- The use of only one implementation of RTSJ (RTS 2.2 by Sun Microsystems which provided a free version for academic research) is one of the limitations of this research. Each RTSJ implementation (such as TimeSys www.timesys.com, and Websphere http://www-03.ibm.com/software/products/us/en/real-time/) can be applied only on specific platforms (Solaris and Linux). Since the main aim of this Thesis is to optimize the use of RTSJ scoped memory in general and not specific to one implementation, this study only considered one implementation. However, implementing the case study in different platforms may give an overview of

the common problems of all implementations. On the other hand, each implementation may have different execution time and space allocation features for scoped and immortal memory.

- The lack of case studies that use scoped and immortal memory. Having different case studies would enable better understanding of the memory model; studying developer experience of using a scoped memory model through different case studies would help in defining more issues and common designing criteria for application of the scoped memory model.
- The research in this Thesis mainly focused on the space overhead even though it would not appear an issue for vast memory storage in railway systems; however, through the experiments, it was discovered that some objects and their methods may generate unexpected objects in scoped and immortal memory which may overflow the memory system over the time. On the other hand, some real-time systems are embedded in small devices which have limited resources and which require careful design and implementation of memory management strategies. The case study did not discuss worst case memory consumption to find the optimal size of scoped memory. The worst case execution time also was not investigated in this study due to time constraints. That would help scheduling analyses to determine (in a formal way) whether all tasks met their deadlines (Puffitsch et al., 2010). In this study, through random experimenting, the scheduling attributes of threads were configured to ensure all threads met their deadlines; however, rigid scheduling analysis is required in the future to help adjust the case study to run on different platforms

6.5 Future Work

The provided limitations offer significant opportunities for future research. Firstly, design patterns proposed in Chapter 5 discussed only problems with the HashTable data structure. In future work, different data structures will be considered such as Vector and ArrayList and their allocation overheads could be analyzed on different platforms and different RTSJ implementations. Secondly, running the case study using the garbage collection process helps in comparing the development complexity, efficiency and space overhead of two versions of the case study. That requires implementing scheduling analysis to configure the garbage collection correctly. Thirdly, developing the tool described in this Thesis to enable a developer to choose from the GUI a number of scopes for each run would be a further avenue of future work; currently, this can only be achieved manually by a developer by updating the simulation code. Further studies in this area to find new methods for improving the performance of scoped memory management are firmly encouraged; implementing software metrics such the ones recommended in (Singer et al., 2008) to help in identifying similar lifetime objects is a future work of the research conducted in this thesis to allocate similar lifetime objects into specific scoped memory areas. To that end, all datasets and simulation tool source code used in this research are included in Appendix A and available to other researchers. Electronic copies can be made available on request of the author.

Lastly, but not least, the research has reflected positively on my personal and professional development. I have learnt how to plan effectively, manage my time appreciating the effort required for the PhD. Effective searching for the most relevant information, seeking help from different people who are knowledgeable in the area,

thinking critically about the problem, decomposing it into smaller parts and finding solutions in step-by-step patterns were the main outputs of my research experience. I have learnt to be patient in order to achieve my aim. I have understood that anything can be in a right or wrong context depending on where it has been used. I have learnt that successes come by hard work, desire, intent, motivation and even from failure. Recovering from failure is the most important factors that lead to success.

References

ALRAHMAWY, M. & WELLINGS, A. (2009) Design patterns for supporting RTSJ component models. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems,* Madrid, Spain: ACM, pp. 11-20.

ANDREAE, C., COADY, Y., GIBBS, C., NOBLE, J., VITEK, J. & ZHAO, T. (2007) Scoped types and aspects for real-time Java memory management. *Real-Time Systems,* 37(1), pp. 1-44.

ARMBRUSTER, A., BAKER, J., CUNEI, A., FLACK, C., HOLMES, D., PIZLO, F., PLA, E., PROCHAZKA, M. & VITEK, J. (2007) A real-time Java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst.,* 7(1), pp. 1-49.

AUERBACH, J., BACON, D. F., BLAINEY, B., CHENG, P., DAWSON, M., FULTON, M., GROVE, D., HART, D. & STOODLEY, M. (2007) Design and implementation of a comprehensive real-time java virtual machine. *Proceedings of the 7th ACM & IEEE international conference on Embedded software,* Salzburg, Austria: ACM, pp. 249 - 258.

BACON, D. F. (2007) Realtime Garbage Collection. *Queue,* 5(1), pp. 40-49.

BAKER, J., CUNEI, A., FLACK, C., PIZLO, F., PROCHAZKA, M., VITEK, J., ARMBRUSTER, A., PLA, E. & HOLMES, D. (2006) A Real-time Java Virtual Machine for Avionics - An Experience Report. *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium,* IEEE Computer Society, pp. 384-396.

BEEBEE, W. & RINARD, M. (2001) An Implementation of Scoped Memory for Real-Time Java. *Embedded Software, Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, Volume 2211, pp. 289-305.

BENJAMIN, S. & STEVE, C. (2008) A framework for the simulation of structural software evolution. *ACM Trans. Model. Comput. Simul.,* 18(4), pp.1-36.

BENOWITZ, E. & NIESSNER, A. (2003) A Patterns Catalog for RTSJ Software Designs. *On The Move to Meaningful Internet Systems 2003: OTM 2003Workshops*. pp 497-507.

BØGHOLM, T., HANSEN, R. R., RAVN, A. P., THOMSEN, B. & SØNDERGAARD, H. (2009) A predictable Java profile: rationale and implementations. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems,* Madrid, Spain: ACM, pp. 150-159.

BØGHOLM, T., HANSEN, R. R., RAVN, A. P., THOMSEN, B. & SØNDERGAARD, H. (2010) Schedulability analysis for Java finalizers. *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems,* Prague, Czech Republic: ACM, pp 1-7.

BOLLELLA, G., BROSGOL, B., GOSLING, J., DIBBLE, P., FURR, S. & TURNBULL, M. (2000) *The Real-Time Specification for Java* Addison Wesley Longman

BOLLELLA, G., CANHAM, T., CARSON, V., CHAMPLIN, V., DVORAK, D., GIOVANNONI, B., INDICTOR, M., MEYER, K., MURRAY, A. & REINHOLTZ, K. (2003) Programming with non-heap memory in the real time specification for Java. *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* Anaheim, CA, USA: ACM, pp. 361-369.

BORG, A. & WELLINGS, A. (2006) Scoped, coarse-grain memory management and the RTSJ scoped memory model in the development of real-time applications. *International Journal of Embedded Systems,* 2(3/4)**,** pp. 166 - 183.

BORG, A., WELLINGS, A., GILL, C. & CYTRON, R. K. (2006) Real-Time Memory Management: Life and Times. *Proceedings of the 18th Euromicro Conference on Real-Time Systems,* Dresden, Germany: IEEE Computer Society, pp. 237 - 250

BOYAPATI, C., SALCIANU, A., BEEBEE, W. & RINARD, M. (2003) Ownership types for safe region-based memory management in real-time Java. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation,* San Diego, California, USA: ACM, pp. 324 - 337.

BROSGOL, B. & WELLINGS, A. (2006) A Comparison of Ada and Real-Time JavaTM for Safety-Critical Applications. *Reliable Software Technologies – Ada-Europe. Lecture Notes in Computer Science*, Volume 4006, Springer Berlin, pp. 13-26.

BRUNO, E. J. & BOLLELLA, G. (2009) *Real-Time Java Programming: with Java RTS,* Prentice Hall.

BURNS, A. & WELLINGS, A. (2001) *Real Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time C/POSIX*, Addison Wesley.

CHANG, Y. (2007) Garbage Collection for Flexible Hard Real-Time Systems. *PhD Thesis*, York University, York.

CHEREM, S. & RUGINA, R. (2004) Region analysis and transformation for Java programs. *Proceedings of the 4th international symposium on Memory management,* Vancouver, BC, Canada: ACM, pp. 85 – 96.

CORSARO, A. & CYTRON, R. K. (2003) Efficient memory-reference checks for real-time java. *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems,* San Diego, California, USA: ACM, pp. 51 - 58.

CORSARO, A. & SCHMIDT, D. C. (2002) Evaluating real-time Java features and performance for real-time embedded systems. *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE.* CA, USA. pp. 90-100

CORSARO, A. & SCHMIDT, D. C. (2003) The Design and Performance of Real-Time Java Middleware. *IEEE Transactions on Parallel and Distributed Systems,* 14(11), pp.1155-1167.

DAWSON, M. (2007) Real-time Java, Part 6: Simplifying real-time Java development. http://www.ibm.com/developerworks/java/library/j-rtj6/. Last accessed on July 6, 2012.

DAWSON, M. H. (2008) Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine. *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)* Orlando, Florida, USA: IEEE Computer Society, pp.241-247.

DEFOE, D., LEGRAND, R. & CYTRON, R. K. (2007) Cost analysis for real-time java scoped-memory areas. *Journal of Systemics, Cybernetics and Informatics,* 5(4)**,** pp. 70-77.

DETERS, M. & CYTRON, R. K. (2002) Automated discovery of scoped memory regions for real-time Java. *Proceedings of the 3rd international symposium on Memory management,* Berlin, Germany: ACM, pp. 132 - 142

DIBBLE, P. & WELLINGS, A. (2009) JSR-282 status report. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems,* Madrid, Spain: ACM, pp. 179-182

DIBBLE, P. C. (2002) *Real-Time Java Platform Programming*, Prentice Hall PTR; 1st edition.

DIBBLE, P. C. (2008) *Real-Time Java Platform Programming*, BookSurge.

DVORAK, D., BOLLELLA, G., CANHAM, T., CARSON, V., CHAMPLIN, V., GIOVANNONI, B., INDICTOR, M., MEYER, K., MURRAY, A. & REINHOLTZ, K. (2004) Project Golden Gate: Towards Real-Time Java in Space Missions. *the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004),* Vienna, Austria: pp. 15-22.

ENERY, J. M., HICKEY, D. & BOUBEKEUR, M. (2007) Empirical evaluation of two main-stream RTSJ implementations. *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems,* Vienna, Austria: ACM, pp. 47-54.

ETIENNE, J.-P., CORDRY, J. & BOUZEFRANE, S. (2006) Applying the CBSE paradigm in the real time specification for Java. *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems,* Paris, France: ACM, pp. 218 - 226

FERRARI, A., GARBERVETSKY, D., BRABERMAN, V., LISTINGART, P. & YOVINE, S. (2005) JScoper: Eclipse support for research on scoping and instrumentation for real time Java applications. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange,* San Diego, California: ACM, pp. 50 - 54.

FRIDTJOF, S. (2006) Proving the absence of RTSJ related runtime errors through data flow analysis. *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems,* Paris, France: ACM, pp. 152 - 161.

GABBAY, F. & MENDELSON, A. (1997) Can program profiling support value prediction? *Proceedings of Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture,* Research Triangle Park, North Carolina: pp. 270-280.

GALLAGHER, K. B. & LYLE, J. R. (1991) Using program slicing in software maintenance. *IEEE Transactions on Software Engineering,* 17(8)**,** pp. 751–761

GAMMA, E., HELM, R., JOHNSON, R. & VLISSIDES, J. M. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

GARBERVETSKY, D., NAKHLI, C., YOVINE, S. & ZORGATI, H. (2005) Program Instrumentation and Run-Time Analysis of Scoped Memory in Java. *Electronic Notes in Theoretical Computer Science,* pp.105-121.

GARBERVETSKY, D., YOVINE, S., BRABERMAN, V., ROUAUX, M. & TABOADA, A. (2009) On transforming Java-like programs into memory-predictable code. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems,* Madrid, Spain: ACM, pp. 140-149.

GUYER, S. Z. & MCKINLEY, K. S. (2004) Finding your cronies: static analysis for dynamic object colocation. *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* Vancouver, BC, Canada: ACM, pp. 237 – 250.

HAMZA, H. & COUNSELL, S. (2010) The Impact of Varying Memory Region Numbers and Nesting on RTSJ Execution. *Proceedings of the 3rd International Conference on Computer and Electrical Engineering (ICCEE 2010)* Chengdu, China, pp. 90-96.

HAMZA, H. & COUNSELL, S. (2012) Simulation of safety-critical, real-time Java: A case study of dynamic analysis of scoped memory consumption. *Simulation Modeling Practice and Theory,* 25, pp 172-189.

HARMAN, M. & DANICIC, S. (1995) Using Program Slicing to Simplify Testing. *Journal of Software Testing, Verification and Reliability,* 5(3)**,** pp 143-162.

HARMAN, M. & HIERONS, R. (2001) An overview of program slicing. *Software Focus,* 2(3)**,** pp. 85-92.

HENRIKSSON, R. (1998) Scheduling Garbage Collection in Embedded Systems. *PhD Thesis,* Lund University, Sweden.

HENTIES, T., HUNT, J. J., LOCKE, D., NILSEN, K., SCHOEBERL, M. & VITEK, J. (2009) Java for Safety-Critical Applications. *2nd InternationalWorkshop on the Certification of Safety-Critical Software Controlled Systems,* York, United Kingdom:

HIGUERA-TOLEDANO, M. T. (2006) Analyzing the Memory Management Semantic and Requirements of the Real-time Specification of Java JSR-0000001. *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing,* Gyeongju, Korea IEEE Computer Society, pp. 419 - 423.

HIGUERA-TOLEDANO, M. T. (2008a) Allowing Cycle References by Introducing Controlled Violations of the Assignment Rules in Real-Time Java. *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC),* pp. 463-467.

HIGUERA-TOLEDANO, M. T. (2008b) Making stronger and flexible the single parent rule in the real-time specification of Java. *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems,* Santa Clara, California: ACM, pp. 19-28.

HIGUERA-TOLEDANO, M. T. (2012) About 15 years of real-time Java. *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems,* Copenhagen, Denmark: ACM, pp. 34-43.

Hirzel, M., Diwan, A., Hertz, M., (2003). Connectivity-based garbage collection. *In Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (OOPSLA '03). New York, NY, USA: ACM, pp. 359-373.

JONES, R. (2007) Dynamic Memory Management: Challenges for Today and Tomorrow. *International Lisp Conference,* Cambridge, England, pp. 115–124.

JUMP, M. & MCKINLEY, K. S. (2010) Detecting memory leaks in managed languages with Cork. *Software—Practice & Experience,* 40(1)**,** pp. 1 - 22.

KALIBERA, T. (2009) Scheduling Hard Real-Time Garbage Collection. *Proceedings of the 30th IEEE Real-Time Systems Symposium,* pp. 81-92

KALIBERA, T., HAGELBERG, J., PIZLO, F., PLSEK, A., TITZER, B. & VITEK, J. (2009) CDx: a family of real-time Java benchmarks. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems,* Madrid, Spain: ACM, pp. 41-50.

KALIBERA, T. & JONES, R. (2013) Rigorous benchmarking in reasonable time. *Proceedings of the 2013 international symposium on International symposium on memory management (ISMM '13),* Seattle, Washington, USA: ACM, pp. 63-74.

KALIBERA, T., PARIZEK, P., HADDAD, G., LEAVENS, G. T. & VITEK, J. (2010) Challenge benchmarks for verification of real-time programs. *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification,* Madrid, Spain: ACM, pp. 57-62.

KELLNER, M. I., MADACHY, R. J. & RAFFO, D. M. (1999) Software process simulation modeling: Why? What? How? *Journal of Systems and Software,* 46(2-3)**,** pp. 91-105.

KELVIN, N. (2012) Revisiting the "perc real-time API". *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems,* Copenhagen, Denmark: ACM, pp. 165-174.

KIM, J.-S. & HSU, Y. (2000) Memory system behavior of Java programs: methodology and analysis. *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems,* Santa Clara, California, United States: ACM, pp. 264 - 274.

KOREL, B. & LASKI, J. (1988) Dynamic Program Slicing. *Information. Processing Letter,* 29(3)**,** pp.155 -163.

KRZYSZTOF PALACZ & VITEK, J. (2003) Java Subtype Tests in Real-Time. *Proceedings of the European Conference on Object Oriented Programming (ECOOP03),* Darmstadt, Germany: Springer, pp. 378-404.

KWON, J. & WELLINGS, A. (2004) Memory Management Based on Method Invocation in RTSJ. *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops.* Cyprus, Lecture Notes in Computer Science, Volume 3292, pp 333-345.

KWON, J., WELLINGS, A. & KING, S. (2002) Ravenscar-Java: a high integrity profile for real-time Java. *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande,* Seattle, Washington, USA: ACM, pp. 681 - 713.

LIANG, D. & HARROLD., M. J. (1998) Slicing Objects Using System Dependence Graphs. *Proceedings of the International Conference on Software Maintenance (ICSM '98),* Washington, DC, USA: IEEE Computer Society, pp. 358 - 367.

LIANG, S. (1999) *Java(TM) Native Interface: Programmer's Guide and Specification*, Addison-Wesley Longman.

MAGATO, W. & HAUSER, J. (2005) Real-time memory management and the Java specification. *48th Midwest Symposium on Circuits and Systems.* Cincinnati, Ohio, pp. 1767-1769

MEERSMAN, R., TARI, Z., CORSARO, A. & SANTORO, C. (2004) Design Patterns for RTSJ Application Development. *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops.* Springer Berlin / Heidelberg..

MOHAPATRA, D. P., MALL, R. & KUMAR, R. (2006) An Overview of Slicing Techniques for Object-Oriented Programs. *Informatica,* 30(2)**,** pp. 253-277.

NAKHLI, C., RIPPERT, C., SALAGNAC, G. & YOVINE, S. (2006) Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems. *Implementation, compilation, optimization of object-oriented languages, programs and systems (ICOOOLPS),* Nantes, France: O. Zendra, pp. 1-8.

NILSEN, K. (2006) A type system to assure scope safety within safety-critical Java modules. *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems,* Paris, France: ACM, pp. 97 - 106.

OTANI, T. W., AUGUSTON, M., COOK, T. S., DRUSINSKY, D., MICHAEL, J. B. & SHING, M. (2007) A design pattern for using non-developmental items in real-time Java. *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems,* Vienna, Austria: ACM, pp. 135 - 143.

PABLO, B.-V., MARISOL, G., A, V., IRIA, E.-A. & CARLOS, D.-K. (2006) Extended portal: violating the assignment rule and enforcing the single parent rule. *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems,* Paris, France: ACM, pp. 30 - 37.

PAN, H. & SPAFFORD, E. H. (1992) Heuristics for automatic localization of software faults. *Technical Report SERC-TR-116-P*, Purdue University.

PÉREZ-CASTILLO, R., CRUZ-LEMUS, J. A., GUZMÁN, I. G.-R. D. & PIATTINI, M. (2012) A family of case studies on business process mining using MARBLE. *Journal of Systems and Software,* 85(6)**,** pp. 1370-1385.

PIZLO, F. (2004) Real-Time Java Scoped Memory: Design Patterns and Semantics. *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing,* Vienna, Austria, pp.101-110

PIZLO, F. & VITEK, J. (2006) An Empirical Evaluation of Memory Management Alternatives for Real-Time Java. *Proceedings of the 27th IEEE International Real-Time Systems Symposium,* Rio de Janeiro, Brazil, pp. 35-46.

PIZLO, F. & VITEK, J. (2008) Memory Management for Real-Time Java: State of the Art. *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing,* Orlando, Florida, pp. 248-254.

PLŠEK, A. (2009) SOLEIL : An Integrated Approach for Designing and Developing Component-based Real-time Java Systems. *PhD Thesis*, INRIA Lille,Lille, France.

PLSEK, A., MERLE, P. & SEINTURIER, L. (2008) A Real-Time Java Component Model. *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing,* Orlando, FL, pp. 281 – 288.

POTANIN, A., NOBLE, J., ZHAO, T. & VITEK, J. (2005) A High Integrity Profile for Memory Safe Programming in Real-time Java. *The 3rd workshop on Java Technologies for Real-time and Embedded Systems.* San Diego, CA, USA,

PUFFITSCH, W., HUBER, B. & SCHOEBERL, M. (2010) Worst-Case Analysis of Heap Allocations. *Leveraging Applications of Formal Methods, Verification, and Validation. Lecture Notes in Computer Science*, Volume 6416, Springer Berlin Heidelberg, pp. 464-478.

R.WILSON, P., JOHNSTONE, M. S., NEELY, M. & BOLES, D. (1995) Dynamic Storage Allocation: A Survey and Critical Review. *Lecture Notes in Computer Science,* Springer Berlin / Heidelberg, Volume 986, pp. 1-116.

RAMAN, K., ZHANG, Y., PANAHI, M., COLMENARES, J. A. & KLEFSTAD, R. (2005a) Patterns and Tools for Achieving Predictability and Performance with Real-Time Java. *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications,* Hong Kong, pp. 247 - 253.

RAMAN, K., ZHANG, Y., PANAHI, M., COLMENARES, J. A., KLEFSTAD, R. & HARMON, T. (2005b) RTZen: highly predictable, real-time java middleware for distributed and embedded systems. *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware,* Grenoble, France: Springer-Verlag New York, Inc., pp. 225- 248.

RIOS, J. R., NILSEN, K. & SCHOEBERL, M. (2012) Patterns for safety-critical Java memory usage. *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems,* Copenhagen, Denmark: ACM, pp. 1-8.

ROBERTZ, S. G. (2003) Flexible automatic memory management for real-time and embedded systems. Licenciate thesis, Lund University, Sweden.

ROBERTZ, S. G., HENRIKSSON, R., NILSSON, K., BLOMDELL, A. & TARASOV, I. (2007) Using real-time Java for industrial robot control. *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems,* Vienna, Austria: ACM, pp. 104-110.

ROSENKRANZ, J. (2004) Rtjsim: A Simulator for Real-Time Java. Diploma Thesis, Salzburg University, Salzburg, Austria.

SALAGNAC, G. (2008) Synthèse de gestionnaires mémoire pour applications Java temps-réel embarquées.PhD, Joseph Fourier University, PhD Thesis, Grenoble, France.

SALAGNAC, G., RIPPERT, C. & YOVINE, S. (2007) Semi-Automatic Region-Based Memory Management for Real-Time Java Embedded Systems. *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications,* Daegu, Korea, pp. 73-80.

SCHOMMER, J. F., FRANKE, D., KOWALEWSKI, S. & WEISE, C. (2009) Evaluation of the real-time Java runtime environment for deployment in time-critical systems. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems,* Madrid, Spain: ACM, pp. 51-60.

SINGH, N. K., WELLINGS, A. & CAVALCANTI, A. (2012) The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada. *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems,* Copenhagen, Denmark: ACM, pp. 62-71.

Singer, J., Marion, S., Brown, G., Jones, R., Lujan, M., Ryder, C., Watson, I., (2008) An Information Theoretic Evaluation of Software Metrics for Object Lifetime Prediction. *In: 2nd Workshop on Statistical and Machine learning approaches to Architectures and compilation (SMART'08),* Page 15.

SPEC-CORPORATION (1999) Java SPEC benchmarks, Technical report. SPEC, Available by purchase from SPEC.

STANKOVIC, J. A. & RAMAMRITHAM, K. (Eds.) (1989) *Tutorial: hard real-time systems*, IEEE Computer Society Press.

STRØM, T. B. & SCHOEBERL, M. (2012) A desktop 3D printer in safety-critical Java. *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems,* Copenhagen, Denmark: ACM, pp. 72-79.

Taylor, S.J.E., Balci, O., Cai, W., Loper, M.L., Nicol, D.N., and Riley, G. (2013) Grand challenges in modeling and simulation: expanding our horizons. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation* (SIGSIM-PADS '13). ACM, New York, NY, USA, pp. 403-408.

TIP, F. (1995) A survey of program slicing techniques. *Journal of Programming Languages,* 3(3)**,** pp. 121-189.

TOFTE, M., BIRKEDAL, L., ELSMAN, M. & HALLENBERG, N. (2004) A Retrospective on Region-Based Memory Management. *Higher Order Symbol. Comput.,* 17(3), pp. 245-265.

TOFTE, M. & TALPIN, J.-P. (1997) Region-Based Memory Management. *Information and Computation,* 132(2), pp. 109-176.

WEISER, M. D. (1979) Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD Thesis, University of Michigan, Ann Arbor, MI., USA.

ZHANG, X., HE, H., GUPTA, N. & GUPTA, R. (2005) Experimental evaluation of using dynamic slices for fault location. *Proceedings of the sixth international symposium on Automated analysis-driven debugging (AADEBUG'05),* Monterey, California, USA: ACM, pp. 33-42.

ZHAO, T., BAKER, J., HUNT, J., NOBLE, J. & VITEK, J. (2008) Implicit ownership types for memory management. *Sci. Comput. Program.,* 71(3), pp. 213-241.

Appendix A Simulation RTSJ Code

Control Thread

```
public class ControlRTThread extends RealtimeThread {

	final static String[]  TrafficR={"","RED"};
	final static String[]  TrafficG={"","GREEN"};
	final static String[]  SensorsOn={"ON",""};
	final static String[]  SensorsOff={"OFF",""};
	final static String T="T";
	final static String testSTR="TEST";
	public ControlRTThread(SchedulingParameters sched, ReleaseParameters rel,MemoryArea mem1)
	{
		super(sched,rel,null,mem1,null,null);

	}

	public void run()
	{ String[] y =new String[2];
	for (int i=0;i<10;i++)Main.z.list1.add(testSTR,i);
	  while(waitForNextPeriod()){

		try {
			if (Main.Tracks.isEmpty())
			{Main.z.list1.removeAll();
			for (int i=0;i<10;i++)Main.z.list1.add("T"+i+"  "+ "GREEN",i);
			break;
			};

			for (int i = 0; i < 10; ++i) {

				y=(String[]) Main.Tracks.get(T+i);

				if (  y[0].equals("OFF"))

				{
					Main.z.list1.remove(i);
					Main.z.list1.add(T+i+"  "+ TrafficG[1],i);
					Main.Tracks.remove("T"+i);
					Main.Tracks.put(T+i,TrafficG);
```

```
                              }//initilaize
                              else  { if (  y[0].equals("ON")){
                                     Main.z.list1.remove(i);
                                     Main.z.list1.add(T+i+"  "+ TrafficR[1],i);
                                     Main.Tracks.remove("T"+i);
                                     Main.Tracks.put(T+i,TrafficR);
                              }}
                       }
               }
               catch ( Exception e ) {
                       e.printStackTrace();
               }

        }
        System.out.println("control exit");

        }

}
```

EmergencyThread.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package traincontrolproject;

import javax.realtime.*;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.*;

/**
 *
 * @author root
 */
public class EmergencyThread extends NoHeapRealtimeThread {
       static int MAX_PRI = PriorityScheduler.instance().getMaxPriority();
```

```
        static RelativeTime TWO_MSEC = new RelativeTime(2, 0);
        PriorityParameters sched = new PriorityParameters(MAX_PRI - 1);
        PeriodicParameters period = new PeriodicParameters(TWO_MSEC);
        Train train1;
        Train train2;

        public EmergencyThread(PriorityParameters priority,
                        PeriodicParameters period, MemoryArea area, Train trainA,
                        Train trainB) throws Exception {
                super(priority, period, null, area, null, null);
                this.train1 = trainA;
                this.train2 = trainB;

                System.out.println("Emergency created between   " + train1.name
                                + " and " + train2.name);

        }

        public void run() {
                try {
                        // delay the trains and show message
                        String screen = "OOOOOOOOOOOPPPPPPPPPPPPPPPPPSSSSSSSSSSSS";
                        Message mes1 = new Message(train1, train2, screen);
                        RestrictionObject Res1 = new RestrictionObject(train1, train2);
                        Res1.Decrease();
                        System.out.println("--------EmeregencyThread1 scope-------------- "
                                        + this.getMemoryArea().memoryConsumed());

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

}
```

LiveThreadControl.java

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package traincontrolproject;

import javax.realtime.*;
import java.awt.*;

/**
 *
 * @author root
 */
public class LiveThreadMonitor extends RealtimeThread {

        public LiveThreadMonitor(SchedulingParameters sched, ReleaseParameters rel,
                        MemoryArea mem1) {
                super(sched, rel, null, mem1, null, null);

        }

        public void check(Train train, String[] Reverserout1, LTMemory mem) {

                if (train.isAlive() == false) {
                        if (train.finish != true) {
                                System.out.println("not alive" + train.name);
                                train = new Train(Reverserout1, 0, train.name, mem, 2);
                                train.start();
                        }
                        ;
                }

        }

        public void run() { /*


                String[] rout1 = { "T7", "T6", "T0", "T3" };
                String[] rout2 = { "T8", "T4", "T2", "T8", "T3" };
                String[] rout3 = { "T2", "T3", "T5", "T2", "T8" };
```

```
String[] rout4 = { "T6", "T1", "T3", "T9", "T8" };
String[] rout5 = { "T4", "T5", "T2", "T4", "T7" };
String[] rout6 = { "T2", "T5", "T4" };
String[] rout7 = { "T3", "T4", "T1", "T6" };
String[] rout8 = { "T8", "T2", "T5", "T8", "T0" };
String[] rout9 = { "T2", "T5", "T1", "T3", "T7" };
String[] rout10 = { "T3", "T4", "T5", "T4" };
String[] rout11 = { "T6", "T7", "T1", "T8" };
String[] rout12 = { "T9", "T8", "T3", "T2" };
String[] rout13 = { "T2", "T1", "T6", "T2" };
String[] rout14 = { "T9", "T7", "T8", "T4" };
String[] rout15 = { "T0", "T3", "T5", "T6" };
String[] rout16 = { "T2", "T8", "T3", "T1" };
int[] journeysNO = new int[17];
for (int i = 0; i < 17; i++) {
        journeysNO[i] = 1;
}

while (waitForNextPeriod()) {
        // just to simplify the proces we omit the function check
        try {
           for(int i=0; i<NoOfTrains;i++)
            {
              if (Main.TrainSet[i].isAlive() == false) {
                if (Main.TrainSet[i].finish != true) {
                 System.out.println("not alive" + Main.TrainSet[i].name);
                 journeysNO[i] = journeysNO[i] + 1;
                 Main.TrainSet[i] = new Train(rout[i], 0,Main.TrainSet[i].name, Main.trains_mem[i],
                    journeysNO[i]);
                Main.TrainSet[i].start();
                }       ;
              }
            };




        } catch (Exception e) {
                e.printStackTrace();
        }

        try {
```

```
                        this.sleep(100);
                } catch (Exception e) {
                        e.printStackTrace();
                }

            }
            System.out.println("LiveThreadExit");

    }

}
```

MonitorThread.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package traincontrolproject;

import com.sun.org.apache.bcel.internal.generic.BREAKPOINT;
import javax.realtime.*;
import java.io.BufferedReader;
```

```
import java.io.InputStreamReader;
import java.util.*;
import javax.swing.JApplet;
import java.awt.*;
import javax.swing.JFrame;
import java.awt.event.*;

/**
 *
 * @author root
 */
public class MonitorRTThread extends RealtimeThread {

        double x, zs, zn, Immo;
        int y, ImmoInt, zt;
        static String EmgString = "Emergency created between  ";
        static AbsoluteTime oldTime, newTime;
        static RelativeTime interval;
        static Clock clock = Clock.getRealtimeClock();
        LTMemory T_status_Mem = new LTMemory(1024 * 10);

        public MonitorRTThread(SchedulingParameters sched, ReleaseParameters rel,
                        MemoryArea mem1) {
                super(sched, rel, null, mem1, null, null);

        }

        public void check(Train train1, Train train2) {
                try {

                        if (train1.finish != true && train2.finish != true) {
                                if (train1.pos == train2.pos && train1.pos != "On Wait") {
                                        if (train1.emg != true && train2.emg != true) {
                                                if ((train1.speed + train2.speed) < 80) {// do
                                                                                                                                                        //
emeergency
                                                                                                                                                        // thread
                                                        train1.emg = true;
                                                        train2.emg = true;
                                                        PriorityParameters sched = new PriorityParameters(
                                                                        PriorityScheduler.instance()
                                                                                        .getMaxPriority());
                                                        PeriodicParameters period = new PeriodicParameters(
```

```
                                                        new RelativeTime(10, 0));

                                                LTMemory EmgMem = new LTMemory(1024 * 16);
                                                EmergencyThread EmeregencyThread1 = new EmergencyThread(
                                                        sched, period, EmgMem, train1, train2);
                                                EmeregencyThread1.start();
                                                Main.z.EmgLabel.setText(EmgString + train1.name
                                                        + " and " + train2.name);
                                        }

                                        else { // crash happened}
                                                System.out.println(" there is a crash between"
                                                        + train1.name + " and " + train2.name);
                                        }
                                }
                        }
                }
        } catch (Exception e) {
                e.printStackTrace();
        }
}

public void run() {
        int i = 1;
        int k = 0;
        oldTime = clock.getTime();
        Runnable Runnable2 = new Runnable() {
                public void run() {
                        Timetable Table1 = new Timetable();
                        for (int i=0;i<NoOfTrains;i++)
                         {
                            Table1.add(i, Main.TrainSet[i].name, Main.TrainSet[i].pos, Main.TrainSet[i].rout);
                          }

                        System.out.println("--------Timetable scope inside--------"
                                        + T_status_Mem.memoryConsumed());
                        // www.setVisible(true);

                }
        };

        while (waitForNextPeriod()) {
```

```
try {

        newTime = clock.getTime();
        // calculate immortal consumption
        Immo = (this.getMemoryArea().memoryConsumed());
        ImmoInt = (int) Immo;
        // calculate scopes consumption
        for (int i=0; i<NoOfTrains; i++)
          {
           x = x+ Main.trains_mem[i].memoryConsumed();
          };

        y = (int) x;
        Main.z.ScopeLabel.setText(String.valueOf(x / 1000) + "Kbytes");
        Main.z.ImmLabel.setText(String.valueOf(Immo / 1000000)
                        + "Mbytes");

        int currenttime = (int) newTime.subtract(oldTime)
                        .getMilliseconds();
        System.out.println("******** The current Time is "
                        + currenttime + "  ******** ");
        Main.z.TimeLabel.setText(String.valueOf(currenttime / 1000)
                        + " Seconds");

        Main.z.ststustableProgressBare.setMaximum(2000);
        Main.z.ststustableProgressBare.setValue(y / 1000);

        Main.z.jProgressBar2.setMaximum(20);
        Main.z.jProgressBar2.setValue(ImmoInt / 1000000);

        System.out.println("--------Immortal memory consumed is  "
                        + Immo / 1000000 + " MB");
        System.out.println("--------  Scopes memory consumed is  " + x
                        / 1000 + " KB");

       for (int i=0; i<NoOfTrains;i++)
          {
           for(int j=i+1; j<=NoOfTrains; j++)
            {
              check(Main.TrainSet[i], Main.TrainSet[j]);
            }
          }
```

```
        T_status_Mem.enter(Runnable2);

 } catch (Exception e) {
        e.printStackTrace();
 }

 int counter=0;
for (int i=1;i<=NoOfTrains;i++)
{
   if (Main.TrainSet[i].finish == true) counter=counter+1;
}

if (counter ==NoOfTrains) Main.Tracks.clear();

    try {
                this.sleep(100);
        } catch (Exception e) {
                e.printStackTrace();
        }
        Main.trains_mem1.enter(new Runnable() {
                public void run() {
                        System.out
                                        .println("Immortal Memory after all thread finish is "
                                                        + ImmortalMemory.instance()
                                                                        .memoryConsumed());

                }
        });

        newTime = clock.getTime();
        interval = newTime.subtract(oldTime);
        System.out.println("interval time:"
                        + interval.getMilliseconds() / 1000);

        JFrame f = new JFrame("Line");
        f.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                        System.exit(0);
                }
        });
```

```
                              break;

                       }
                       ;

               }
               System.out.println("monitor exit");

        }

}
```

Train.java

```
package traincontrolproject;

import javax.realtime.*;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.*;

/**
 *
 * @author root
 */
public class Train extends RealtimeThread {
        String[] rout;
        int speed;
        String pos;
        boolean finish;
        String name;
        String screen;
        boolean emg;
        int routNO;

        public Train(String[] rout1, int speed, String name, LTMemory mem1,
                       int routNO) {
               super(null, null, null, mem1, null, null);
               this.speed = speed;
               this.rout = rout1;
               this.name = name;
```

```
		this.routNO = routNO;

	}

	public Train(String[] rout1, int speed, String name, LTMemory mem1) {
		super(null, null, null, mem1, null, null);
		this.speed = speed;
		this.rout = rout1;
		this.name = name;
		this.routNO = 0;
		this.emg = false;
	}

	public String[] getRout() {
		return this.rout;
	};

	public int getSpeed() {
		return this.speed;
	};

	public void setScreen(String screen) {

		this.screen = screen;
		System.out.println(screen);
	};

	public void run() {
		try {
			this.finish = false;
			String[] z = new String[2];
			for (int i = 0; i < this.rout.length; ++i) {
				z = (String[]) Main.Tracks.get(this.rout[i]);
				if (z[1].equals("RED")) {
					System.out.println(this.name
							+ " is waiting until the traffic light sets green");
					this.pos = "On Wait";
				}
				while (z[1].equals("RED")) {

					z = (String[]) Main.Tracks.get(this.rout[i]);

				}
```

```
                        synchronized (this) {
                        Main.Tracks.put(this.rout[i], ControlRTThread.SensorsOn);
                        this.pos = this.rout[i];
                        while (this.speed <= 100 && this.speed != -100)// moving on
                                                                                                    // the first
                                                                                                    // track
                        {
                                this.speed = speed + 1;
                                this.sleep(125);
                        }
                        ;
                        if (this.speed == -100) {
                                i = i - 1;
                                if (i < 0) {
                                        i = 0;
                                }
                                this.pos = "On Wait";
                                this.speed = 0;
                                this.emg = false;

                        } else {
                                this.speed = 0;
                                Main.Tracks.put(this.rout[i],
                                                ControlRTThread.SensorsOff);
                        }
                }

        }
        ;

        if (this.routNO == 1) {
                this.pos = "Terminated at end of its route 1 ";

        }

        else if (this.routNO == 2) {
                this.pos = "Terminated at end of its route 2 ";
        } else if (this.routNO == 3) {
                this.pos = "Terminated at end of its route 3 ";

        } else
```

```
                        {
                                this.pos = "Terminated at end of its route 4 ";
                                this.finish = true;
                        }
                        Clock Clock1 = Clock.getRealtimeClock();
                        AbsoluteTime finishtime = Clock1.getTime();
                        RelativeTime period = finishtime.subtract(Main.start);
                        System.out.println(this.name
                                        + " Has finished its current route at " + period);
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Main.java

```
package traincontrolproject;

import javax.realtime.*;
import java.util.*;

/**
 *
 * @author Hamza Hamza
 */
public class Main {

        static int MAX_PRI = PriorityScheduler.instance().getMaxPriority();
        public static Hashtable Tracks;
        static RealtimeThread rt;
       public static final NoOfTrains=16

       public static LTMemory[] trains_mem = new LTMemory[NoOfTrains];

        public static Train[] TrainSet;
        public static LTMemory test_mem4 = new LTMemory(1024 * 120);
        static Clock clock = Clock.getRealtimeClock();
        static AbsoluteTime start;
```

```
        public static NewJFrame z = new NewJFrame();
        static {
                rt = new RealtimeThread(new PriorityParameters(MAX_PRI - 1), null, // new
                                // PeriodicParameters(new
                                // RelativeTime(20,0)),
                                null, ImmortalMemory.instance(), null, null) {
                        public void run() {
                                TrainSet = new Train[17];
                                Tracks = new Hashtable();
                                String[] y = new String[2];
                                y[0] = "OFF";
                                y[1] = "GREEN";
                                for (int i = 0; i < 10; ++i) {
                                        Tracks.put("T" + Integer.toString(i), y);// initilaize
                                }
                                // initialize the routs

                                String[] rout1 = { "T1", "T4", "T3" };
                                String[] rout2 = { "T4", "T6", "T7" };
                                String[] rout3 = { "T9", "T8", "T6", "T5", "T4" };
                                String[] rout4 = { "T5", "T4", "T3", "T2", "T1" };
                                String[] rout5 = { "T6", "T3", "T2", "T1" };
                                String[] rout6 = { "T3", "T8", "T2" };
                                String[] rout7 = { "T2", "T1", "T7", "T9" };
                                String[] rout8 = { "T8", "T9", "T5", "T9", "T8" };
                                String[] rout9 = { "T6", "T3", "T1", "T8", "T0" };
                                String[] rout10 = { "T0", "T1", "T2" };
                                String[] rout11 = { "T3", "T4", "T8", "T9" };
                                String[] rout12 = { "T2", "T5", "T4", "T1" };
                                String[] rout13 = { "T3", "T1", "T7", "T8", "T4" };
                                String[] rout14 = { "T7", "T1", "T3" };
                                String[] rout15 = { "T9", "T6", "T4", "T0" };
                                String[] rout16 = { "T6", "T3", "T2", "T1" };
                                // assign routs to trains with the initial speeds

                                trains_mem[0] = new LTMemory(1024 * 2000);
                                for (int i=1 i<NoOfTrains; i++)
                                 {
                                   trains_mem[i] = new LTMemory(1024 * 32);;
                                 };

                                PriorityParameters schedControl = new PriorityParameters(
                                                MAX_PRI);
```

```
                            PriorityParameters schedMonitor = new PriorityParameters(
                                            MAX_PRI - 1);
                            PriorityParameters schedLiveThreads = new PriorityParameters(
                                            MAX_PRI - 5);
                            ReleaseParameters relLiveThreads = new PeriodicParameters(
                                            new RelativeTime(1000, 0));
                            ReleaseParameters relControl = new PeriodicParameters(
                                            new RelativeTime(120, 0));
                            ReleaseParameters relMonitor = new PeriodicParameters(
                                            new RelativeTime(300, 0));
                            ControlRTThread MyControlRTThread = new ControlRTThread(
                                            schedControl, relControl, ImmortalMemory.instance());
                            MyControlRTThread.start();
                            for(int i=1;i<=NoOfTrains;i++)
                             {
                              TrainSet[i] = new Train(Araaylist.get(i), 0, "train+"i, trains_mem[i], 1);
                              TrainSet[i].start();
                              };

                            MonitorRTThread myMonitorRTThread = new MonitorRTThread(
                                            schedMonitor, relMonitor, ImmortalMemory.instance());
                            myMonitorRTThread.start();

                            System.out.println("********   trains start moving   ********");
                            LiveThreadMonitor MyLiveThreadMonitor = new LiveThreadMonitor(
                                            schedLiveThreads, relLiveThreads,
                                            ImmortalMemory.instance());
                            MyLiveThreadMonitor.start();
                    };
            };
    };

    public static void main(String[] args) {
            // initialize the traks-hashtable we have 5 tracks with 5 sensors and 5
            // switches
            start = clock.getTime();
            rt.start();
            z.setLocation(300, 300);
            z.setVisible(true);
            z.ststustableProgressBare.setStringPainted(true);
            z.jProgressBar2.setStringPainted(true);

            // TODO code application logic here
```

```
        }

}
```

Restricted object.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package traincontrolproject;

/**
 *
 * @author root
 */
public class RestrictionObject {
        // int speed;

        public RestrictionObject(Train train1, Train train2) {
                train1.speed = -100;// stop the train for a while
                System.out.println(train1.name + " has been stopped untill "
                                + train2.name + "  finishes its current Track  ");
                train2.speed = train2.speed - 20;
                // divertthe rout;
        }

        public void Decrease() {
                // speed=speed-1;
        }
}
```

Message

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package traincontrolproject;
```

```
/**
 *
 * @author root
 */
public class Message {

        public Message(Train train1, Train train2, String screen)

        {
                train1.setScreen(screen);
                train2.setScreen(screen);
        };

}
```

Trainstatus

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package traincontrolproject;

/**
 *
 * @author root
 */
public class trainstatus {
String train_name; String train_pos; String[] train_rout;
}
```

Timetable

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package traincontrolproject;

/**
 *
 * @author root
 */
public class Timetable {

        trainstatus[] arrayOftrainStatus;

        public Timetable() {

                arrayOftrainStatus = new trainstatus[18];

        }

        public void add(int x, String train_name, String train_pos,
                        String[] train_rout) {// need to be modified later..
                this.arrayOftrainStatus[x - 1] = new trainstatus();
                this.arrayOftrainStatus[x - 1].train_name = train_name;
                this.arrayOftrainStatus[x - 1].train_pos = train_pos;
                this.arrayOftrainStatus[x - 1].train_rout = train_rout;
        };
}
```

Appendix B Control Thread Slicing

Original code

```
public class ControlRTThread extends RealtimeThread {

        final static String[]  TrafficR={"","RED"};
        final static String[]  TrafficG={"","GREEN"};
        final static String[]  SensorsOn={"ON",""};
        final static String[]  SensorsOff={"OFF",""};
        final static String    T="T";
  public ControlRTThread(SchedulingParameters sched, ReleaseParameters
                          rel,MemoryArea mem1)
        {
                super(sched,rel,null,mem1,null,null);
        }
        public void run()
        { String[] y =new String[2];
        for (int i=0;i<10;i++)Main.z.list1.add(testSTR,i);
          while(waitForNextPeriod()){
                try {
                        if (Main.Tracks.isEmpty())
                        {Main.z.list1.removeAll();
                        for (int i=0;i<10;i++)Main.z.list1.add("T"+i+"  "+
                           "GREEN",i);
                        break;
                        };
                        for (int i = 0; i < 10; ++i) {
                                y=(String[]) Main.Tracks.get(T+i);
                                if (  y[0].equals("OFF"))
                                {
                                        Main.z.list1.remove(i);
                                        Main.z.list1.add(T+i+"  "+ TrafficG[1],i);
                                        Main.Tracks.remove("T"+i);
                                        Main.Tracks.put(T+i,TrafficG);
                                }//initilaize
                                else  { if (  y[0].equals("ON")){
                                        Main.z.list1.remove(i);
                                        Main.z.list1.add(T+i+"  "+ TrafficR[1],i);
                                        Main.Tracks.remove("T"+i);
                                        Main.Tracks.put(T+i,TrafficR);
                                }}
                        }
                }
                catch ( Exception e ) {
                        e.printStackTrace();
                }

        }
        System.out.println("control exit");

    }

}
```

Slice 1

```
public class ControlRTThread extends RealtimeThread {
        final static String[]  TrafficR={"","RED"};
        final static String[]  TrafficG={"","GREEN"};
        final static String[]  SensorsOn={"ON",""};
        final static String[]  SensorsOff={"OFF",""};
public ControlRTThread(SchedulingParameters sched, ReleaseParameters
                       rel,MemoryArea mem1)
    {
        super(sched,rel,null,mem1,null,null);
    }
   public void run()
    {
        String[] y =new String[2];
               while(waitForNextPeriod())
                  {
                    for (int i = 0; i < 10; ++i)
                       {
                            y= Main.Tracks.get(T+i);
                            }//end_for
                        } //end_while
    }
  }
}
```

Slice 2

```
public class ControlRTThread extends RealtimeThread {
        final static String[]  TrafficR={"","RED"};
        final static String[]  TrafficG={"","GREEN"};
        final static String[]  SensorsOn={"ON",""};
        final static String[]  SensorsOff={"OFF",""};
public ControlRTThread(SchedulingParameters sched, ReleaseParameters
                       rel,MemoryArea mem1)
    {
        super(sched,rel,null,mem1,null,null);
    }
  public void run()
    {
        String[] y =new String[2];
               while(waitForNextPeriod())
                {
                  for (int i = 0; i < 10; ++i)
                   {
                      Main.Tracks.put(T+i,TrafficG);
                   }//end_for
                } //end_while

    }
  }
```

Control Thread Updated code

```
public class ControlRTThread extends RealtimeThread {
        final static String[]  TrafficR={"","RED"};
        final static String[]  TrafficG={"","GREEN"};
        final static String[]  SensorsOn={"ON",""};
        final static String[]  SensorsOff={"OFF",""};
        Set<Entry<String, String[]>> entries =Main.Tracks.entrySet();
         Runnable DesignPatternToModify1 =new Runnable(){public void run(){
              for(Entry<String, String[]> ent: entries){
               if (ent.getKey().equals(T+Main.counter)){
                   ent.setValue(TrafficR);
               break;}
               };
             }};
         Runnable DesignPatternToModify2=new Runnable(){public void run(){

               for(Entry<String, String[]> ent: entries){
               if (ent.getKey().equals(T+Main.counter)){
               ent.setValue(TrafficG);
               break;}
               };
             }};
         Runnable DesignPatternToRead =new Runnable(){public void run(){
           String[] y=(String[]) Main.Tracks.get(T+Main.counter);
           if (  y[0].equals("OFF")){
                 Main.ref=1;
            }
           else if (  y[0].equals("ON")) Main.ref=2;
           else if (  y[0].equals("")) Main.ref=0;
         }};

         static Runnable Runnable2=  new Runnable(){public void run(){
                  System.out.println(ImmortalMemory.instance().memoryConsumed());
                   }
           };
        public ControlRTThread(SchedulingParameters sched, ReleaseParameters
                            rel,MemoryArea mem1)
         {
           super(sched,rel,null,mem1,null,null);
         }
           public void run()
            {
             while(waitForNextPeriod()){
               if (Main.Tracks.isEmpty())
                 {
                 Main.z.list1.removeAll();
                  for (int i=0;i<10;i++)Main.z.list1.add("T"+i+"  "+ "GREEN",i);
                  break;
                };
              for (int i = 0; i < 10; ++i)
                {
                  Main.counter=i;
                  Main.test_mem4.enter(DesignPatternToRead);
                   if (Main.ref==1) Main.test_mem4.enter(DesignPatternToModify2) ;
                   else
                   {  if  (Main.ref==2)  Main.test_mem4.enter(DesignPatternToModify1)
;

                   }
             };
       }
  }
}
```