The Impact of Varying Memory Region Number on RTSJ Execution Time
(*Short Paper*)

H. HAMZA
Department of Computing,
Brunel University,
Uxbridge, Middlesex, UB8 3PH,UK
hamza.hamza@brunel.ac.uk

Abstract— **Developing a real time, embedded system in Java requires awareness of memory behaviour in addition to program logic. Region-based memory management has certain advantages over garbage collection in terms of predictability; however, the use of regions is in many ways arbitrary, since it is left to the developer to decide on the number of required regions prior to program execution. In theory, a memory model with a large number of regions will negatively impact the execution time of software since the run-time system has the overhead of managing the space in each of these regions. In this paper, we explore the effect of varying the number of regions on the performance of RTSJ execution times. RTSJ code was used to allocate varying numbers of objects (100 to 2500) into regions and then execution times were recorded. Results suggest that more regions *do* actually lead to increases in execution time. By applying a relatively simple *refactoring* to the original code, an improvement in execution times was achieved.**

Keywords-component, RTSJ, memory regions, Java, refactoring.

I. INTRODUCTION

Memory management in real time and embedded Java systems is still an open research area. Previous studies in the area have tended to focus on memory management for preventing memory 'leaks' and for simplifying the development of such systems [1]. Defining application parameters such as the garbage generation rate is problematic with respect to achieving low time and space overheads in an application [2]. The Real-Time Specification for Java (RTSJ) introduces a new programming model - the *memory region* to circumvent the need for garbage collection [3]. However, this model has many drawbacks [4,5]. For example, the need for reference checks between regions may introduce unexpected overheads [6,7], and more germane, the arbitrary decision as to how many regions is left to the developer to decide; mismanagement of region numbers can often result in run-time errors [8]. In this paper, the impact of increasing region numbers on the execution time is investigated. RTSJ code was executed with different number of regions with varying numbers of objects and the execution times then recorded. In theory, higher numbers of regions should lead to increased execution times since the runtime system has to determine where objects can be accommodated on a more frequent basis (the memory management burden is naturally higher). Results did confirm that with more regions, execution times did increase. A programming refactoring allowed execution times to be lowered.

II. MOTIVATION AND RELATED WORK

The motivation for this study stems from a number of senses. The first is that to our knowledge, very little work has been completed to assess the relative merits of different numbers of regions and execution times. Yet, the decision that a developer has to make on region numbers can have a significant impact on potential application efficiency and execution time. Second, we challenge the view that increasing the number of regions always results in a perfectly linear rise in execution times.

In terms of related work, work to evaluate real time Java features and performance for real-time embedded systems was undertaken by [9] who compared various RTSJ implementations. Results showed that memory allocation times were linear. Recent work by [10] evaluated the RTSJ from different perspectives, the relationship between allocation time and the object size allocated into memory regions was explored - the relationship was again shown to be linear. In another approach by [11], 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, and their results regarding memory allocation showed that the creation times for regions were again linear with region sizes. Furthermore, object allocation times were also linear with object sizes. The work presented in this paper focuses on the impact of varying memory region number on RTS2.0 execution time.

III. REGION-BASED MEMORY MANAGEMENT

In standard Java, all objects are allocated on heap memory and are subject to garbage collection. The RTSJ provides two other types of memory which are not subject to garbage collection: a) immortal memory which stores objects that will be allocated forever and, b) scoped memory which has a bounded life time and where similar length of lifetime objects should reside. Objects allocated in scoped memory

area are freed automatically when no active object exists, however the scoped memory object itself can be stored in heap, immortal, or scoped memory. For example, all objects in a region will be freed automatically when a thread finishes executing its 'run' method and when no other threads are executing inside that scope or region of memory. In turn, there are two types of scope memory in RTSJ: 1) VTMemory and, 2) LTMemory. In LTMemory, allocation time is linear with object size; however, in VTMemory allocation time varies [3]. In this paper, we use LTMemory, which thus guarantees linear-time allocation. In this study, each memory region is created by defining a new object memory area:

```
mem = new LTMemory(8*1024);
```

This creates a new LTMemory area with fixed size 8K bytes. The new object 'mem' will point to that region of memory. The next step is to start using the block of memory referenced by 'mem'. A 'Runnable' object should be used in the *enter* method of the 'mem' object and 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 no arguments called 'run' as follows:

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

All objects created inside the 'run' method of the Runnable object will be allocated in the memory area referenced by 'mem'. There are other ways to enter a memory region in addition to the 'enter' method such as newArray and executeInArea, each of which has its own use. For a full description on the use of these methods and for the code upon which we based our experiment see [3].

IV. VARYING REGION NUMBERS

A. RTSJ experiments' code

Creating the objects in the RTSJ program was facilitated through an array of objects (see line 12 in Appendix A). The code can then easily be updated with larger numbers of objects (in this case of this study, from 100 to 2500 incremented by 100 objects each execution). Appendix A 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 regions of memory each with size 8*1024 bytes. The region itself is created when its 'enter' method is executed by a 'Runnable' object. 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 (see lines 13 and line 22 in Appendix A, respectively). Example 1 shows only 2 regions allocating 100 objects; we then updated the code to include 5, 10, 15, 20 and 25 regions respectively. All regions have the same size "8K bytes" and the number of objects distributed into each region for each set of region experiments is approximately equal. For example, for 5 regions and the allocation of 500 objects, each region has 100 objects allocated to it. These regions are de-allocated when Runnable objects finish executing their 'run' methods. A for-loop is used to execute the re-activation of the regions multiple times according to the number of parameters entered. The type of parameter is thus Integer, and the values of these parameters are the value of the Integer objects allocated into the regions. In our experiments we use two integer parameters to execute the for-loop twice. Execution time was measured using Java clock statements clock.getTime(). The code was run in RTSJ 2.0 Sun implementation, Linux openSUSE11.1 operating system-kernel 2.6.27.7-9 and on a stand-alone computer with Intel Pentium Dual core processor speed 2.8 GHZ and RAM, capacity 2GB.

B. Data analysis

Figures 1 and 2 show the impact of increasing memory regions on execution time for 5 and 10 regions, respectively with 100-2500 objects, at intervals of 100, giving 25 data points in total for each region. We ran the experiments 50 times for each data point and we calculate the average of the execution time. The R^2 (correlation coefficient) value for 5 regions is significantly higher than that for 10 regions (0.587 versus 0.379). It is interesting that the steepness of the curve in Figure 1 is greater at lower numbers of objects and that there are a number of falls in the execution times along the graph even though the trend is generally upwards. One suggestion is that, however many regions are defined, there is a lower limit on execution time due to the overheads of creating the regions and allocating the first *n* objects. After that point, the system 'stabilises'.

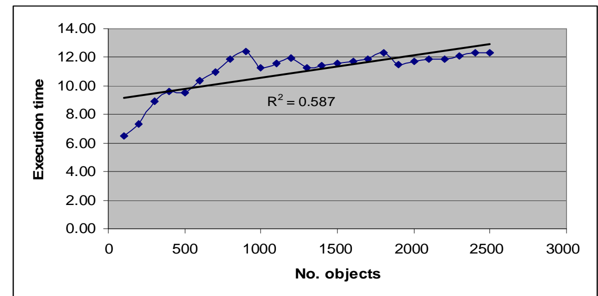

Figure 1. 5 region data (2500 objects)

For 10 regions (Figure 2), the execution time is higher than that of 5 regions, but the larger number of regions seems to provide greater flexibility to the run time system above the 900 object threshold not afforded by 5 regions. Of all the regions studied, 10 regions had the lowest R^2 value (0.38).

We also see that for 10 regions there is a sudden fall in execution time at 1900 objects and a further fall at 2500 objects. Again, 1900 objects might represent a threshold (borderline) region limit beyond which allocation of objects and region space limitations becomes less prevalent. Overall however, allocating 5 regions will lead to faster execution

times than for 10 regions (for the object configuration described). Figure 3 shows the effect on execution time of 15 regions.

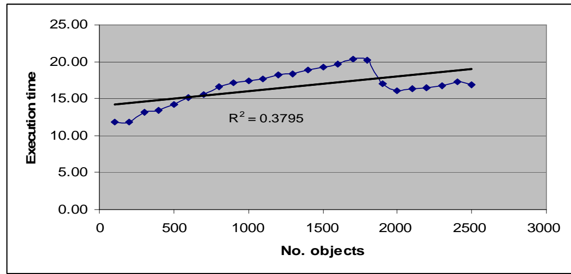

Figure 2. 10 region data (2500 objects)

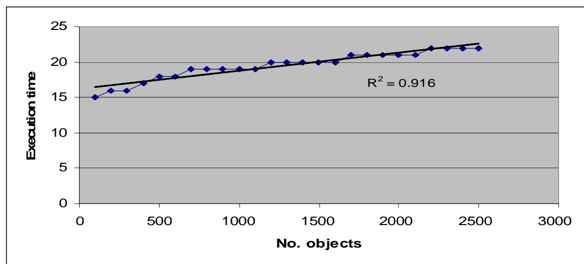

Figure 3. 15 region data

Figure 4 and 5 show the execution times for 20 and 25 regions, respectively. The smooth rise in execution times is evident from both figures and is again in contrast to that for 5 and 10 regions.

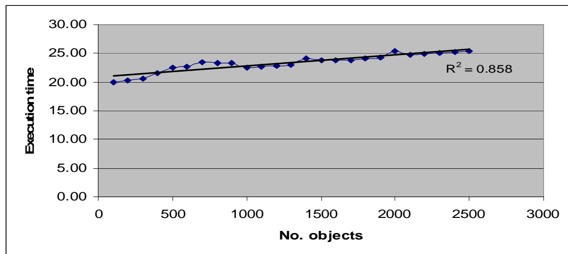

Figure 4. 20 region data

The highest execution time amongst all configurations in fact belongs to 25 regions (29.87ms for 2100 objects), suggesting further that as the number of regions increases, there is an associated natural overhead in execution time. The rise in execution times becomes flatter as the number of regions increases. It also interesting that of all regions (5, 10, 15, 20 and 25), 10 regions caused the widest variation in execution times (11ms to 20ms – an interval of 9ms).

C. *Summary data*

Table 1 provides summary data (Mean, Median (Med.) and Standard Deviation (SD)) for each of the set of regions. The widest variation in execution times is for 10 regions (SD of 2.35); the narrowest times are for 5 and 20 regions (1.54 in each case).

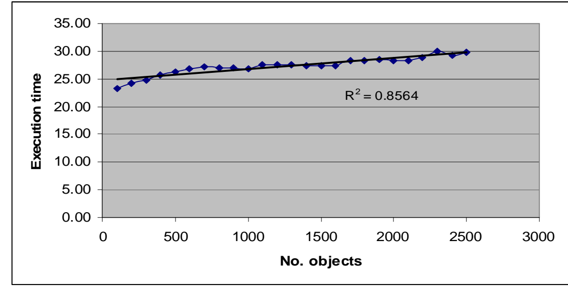

Figure 5. 25 region data

Table 1. Summary data for regions

Regions	Mean	Med.	SD
5	11.05	11.58	1.54
10	16.63	16.85	2.35
15	19.52	20.00	1.98
20	23.33	23.38	1.54
25	27.33	27.46	1.61

V. A PROGRAMMING VARIATION

In some cases, if the run methods of two or many Runnable objects are the same, then it would be useful to create just one Runnable object and then make a reference to it. Each time a memory region is entered, this reference will be used in the enter method of the memory region instead of creating a new Runnable object. For example, if we have an object 'foo', shareable between two threads and is allocated in one memory region, each thread will modify this object in a synchronized way. Each thread will run in a different region, but both will run concurrently and both are instantiated from the same class (NoHeapRealTimeThread). The 'run' method of both new Runnable objects in mem1 and mem2 include the creation and start of a new NhRTT thread; the same code exists in the two run methods. In this case, we can then refactor the code [12] by creating one Runnable object and make a reference to it (x). Hence, we updated the code to make mem1 and mem2 use the same Runnable object as shown Figure 6. We ran the code twice, the first time without changing the code (see Appendix B), and the second time by using the changes in Figure 6.

```
Runnable x = new Runnable(){
                        public void run() {
                          NhRTT nhrtt = new NhRTT(2, foo);
                          System.out.println(" new Thread  started");
                          nhrtt.start();
                         } };
mem1.enter(x);
try {       //Put a little delay between the threads
               sleep(100);
    } catch (InterruptedException e) { };

mem2.enter(x);
```

Figure 6. Refactored Code

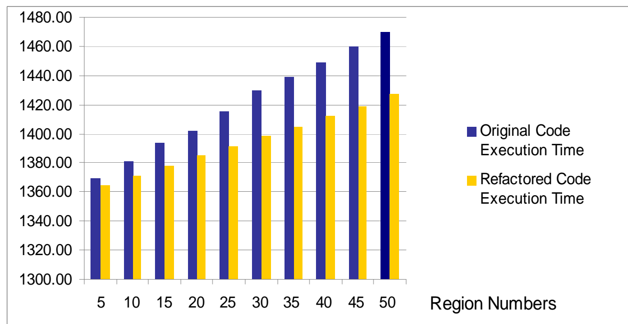

Figure 7. Effect of refactoring original code

Results show that execution time decreases after refactoring the code to have only one Runnable object instead of two, three or more.

For example, when using 10 regions, the execution time for the original code where 10 Runnable objects are created is 1381 milliseconds. In the refactored code, where only one Runnable object is created, the execution time is 1371 milliseconds. This supports our view that Runnable objects causes the creation of regions to take longer. We note that this programming approach works only when many 'Runnable' objects have similar run methods. Figure 7 shows the results after refactoring the code, i.e. reducing the number of Runnable objects in the application, leading to a reduction in execution time.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, the impact of the number of memory regions on the execution time of RTSJ software was investigated. Sample RTSJ code was executed with different numbers of regions. The results showed that increasing the number of regions did tend to lead to longer execution times. One obvious conclusion from the study is 'if increasing the number of regions increases execution time, why not have just one large region or at least a minimum number of regions?' Of course, such a policy has to be judged against the advantages of scoped memory (i.e., the ability to free up unused regions or expired memory and storage of similar object (lifetime) types in dedicated regions). Additionally, with fewer regions, the possibility of dead objects lying around and being unable to be reclaimed is also a feature.

Future work will focus on extending the analysis to different types of objects and region numbers. In particular, we will explore the reasons behind the sudden fall visible in Figure 2. Furthermore, we will investigate other methods to decrease the impact of varying region numbers on execution time in order to cover different patterns in the code. The impact of having nested scopes on the execution time of the application will be explored. We also will develop tools to help developers to define the lifetime of objects and distribute them into different regions; this will aid specification of the right number of regions in an effective way. We encourage further studies in this area – to that end, the dataset used as part of this empirical study can be made available for further analysis upon request of the authors.

REFERENCES

[1] F. Pizlo and J. Vitek, "Memory Management for Real-Time Java: State of the Art", in Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, IEEE Computer Society, Orlando, Florida, USA, 2008, pp. 248-254.

[2] G. Salagnac, C. Rippert, and S. Yovine, "Semi-Automatic Region-Based Memory Management for Real-Time Java Embedded Systems", in Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. IEEE Computer Society, Daegu, Korea. 2007, pp. 73-80.

[3] P.C. Dibble, "Real-Time Java Platform Programming", 2 ed., BookSurge, 2008.

[4] F. Pizlo and J. Vitek, "An Emprical Evaluation of Memory Management Alternatives for Real-Time Java", in Proceedings of the 27th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Rio de Janeiro, Brazil, 2006, pp. 35-46.

[5] W. Magato and J. Hauser. "Real-Time Memory Management and the Java Specification", In 48th IEEE International Midwest Symposium on Circuits and Systems, Cincinnati, Ohio, 2005, pp. 1767-1769.

[6] Y. Chang, "Garbage Collection for Flexible Hard Real-Time Systems", PhD thesis, York University, 2007.

[7] A. Borg, A. Wellings, C. Gill, and R.K. Cytron, "Real-Time Memory Management: Life and Times", in Proceedings of the 18th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Dresden, Germany, 2006, pp. 237 – 250.

[8] J. Kwon and A. Wellings, "Memory Management Based on Method Invocation in RTSJ", in On the Move to Meaningful Internet Systems Workshops, Springer, Agia Napa, Cyprus, 2004, pp. 333-345.

[9] A. Corsaro and D.C. Schmidt, "Evaluating real-time Java features and performance for real-time embedded systems", in Proceedings of The 8th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, San Jose, CA, USA, 2002, pp. 90-100.

[10] J.F. Schommer, D. Franke, S. Kowalewski, and C. Weise, "Evaluation of the Real-Time Java Runtime Environment for Deployment in Time-Critical Systems", in Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, ACM, Madrid, Spain. 2009, pp. 51-60.

[11] J.M Enery, D. Hickey, and M. Boubekeur, "Empirical Evaluation of Two Main-Stream Rtsj Implementations", in Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, ACM Vienna, Austria, 2007, pp. 47-54.

[12] M. Fowler, "Refactoring : Improving the Design of Existing Code", Addison-Wesley, 1999.

APPENDIX A –

```
1.public class Example1with2regions100objects extends RealtimeThread {
2.---------------
 3. public void run(){
 4.             mem1 = new LTMemory(8*1024);
 5.             mem2 = new LTMemory(8*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 Example1with2regions100objects

 31.   RealtimeThread rt = new Example1with2regions100objects(args);
 32.  oldTime= clock.getTime();

 33.   rt.start();
 34.   try {
 35.   rt.join();
 36.  }
 37. catch (Exception e) { };
 38. }
 39. }
```

APPENDIX B –

```
1.        //   Start the threads.
2.          mem1.enter(new Runnable () {
3.                  public void run() {
4.                                NhRTT nhrtt = new NhRTT();
5.                                System.out.println(" Thread  started");
6.                                nhrtt.start();
7.                   } });
8.         try { //Put a little delay between the threads
9.         sleep(100);
10.        } catch (InterruptedException e) { };

11.        mem2.enter(new Runnable () {
12.                 public void run() {
13.                                NhRTT nhrtt = new NhRTT();
14.                                System.out.println(" Thread  started");
15.                                nhrtt.start();
16.        } });
```