2008-08-15

A comparison of priority-based and incremental real-time garbage collectors in the implementation of the shadow design pattern

Otani, Thomas W.; Drusinsky, Doron; Michael, James Bret; Shing, Michael

Monterey, California. Naval Postgraduate School

NPS-CS-08-011

NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

A Comparison of Priority-based and Incremental Real-Time Garbage Collectors In the Implementation of the Shadow Design Pattern

by

T.W. Otani, D. Drusinsky, J.B. Michael and M. Shing

15 August 2008

Prepared for: Missile Defense Agency
7100 Defense Pentagon
Washington, D.C. 20301-7100

This report was prepared for and funded by the Missile Defense Agency.

This report was prepared by:

____
Thomas Otani
Associate Professor of Computer Science
Naval Postgraduate School

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** May 2008	**3. REPORT TYPE AND DATES COVERED** Technical Report	
4. TITLE AND SUBTITLE: Title (Mix case letters) A Comparison Of Priority-Based And Incremental Real-Time Garbage Collectors In The Implementation Of The Shadow Design Pattern			**5. FUNDING NUMBERS** MD7080101P0630
6. AUTHOR(S) Thomas W. Otani, Doron Drusinsky, James Bret Michael, and Man-Tak Shing			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER** NPS-CS-08-011
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Missile Defense Agency, 7100 Defense Pentagon, Washington, DC 20301-7100			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			**12b. DISTRIBUTION CODE**
13. ABSTRACT (maximum 200 words) This is our third report on real-time Java. Our previous work to develop and evaluate the Shadow Design Pattern was couched in the context of real-time garbage collection with assignable priorities as implemented for example in the Sun Java Real-Time System. In this report, we present our investigation of the pattern from the perspective of non-assignable priorities. Our experiment consisted of running the real-time application we used in our previous study on IBM WebSphere Real Time. IBM WebSphere Real Time automatically sets Metronome, its incremental real-time garbage collector, to a priority higher than the highest priority of the real-time threads that use the heap. The results from the experiment show that the modified code for the Shadow Design Pattern runs well under Metronome.			
14. SUBJECT TERMS Real-time system, Java programming language, Garbage collection, Design pattern.			**15. NUMBER OF PAGES** 25
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

1. Introduction

Writing a correct real-time Java program with the no-heap real-time threads can be difficult [1-3]; the use of no-heap real-time threads involves a complex programming model that is difficult to understand and hard to analyze [4-7]. In [8], we concluded that it is preferable to use only the regular, heap-using real-time threads for a class of real-time applications whose computations must be terminated by their hard deadlines and have to return the best approximations to their clients if they cannot finish their computations by the deadlines; we developed a new design pattern, called the Shadow Design Pattern, for this class of applications. We described how well this design pattern works with the Sun Java Real-Time System (RTS) 2.0 in [9]. The key feature that makes the Shadow Design Pattern successful is the availability of the real-time garbage collector (RTGC) whose priority is assignable.

Since not all RTSJ implementations support such priority-assignable RTGC, our next task is to determine the effectiveness of the Shadow Design Pattern when it is used with other types of RTGC, namely the incremental RTGC with non-assignable priorities. In this paper, we describe the experiment we performed by running the Shadow Design Pattern on the IBM WebSphere Real Time that includes the real-time garbage collector called the Metronome. Unlike the Sun RTGC, we cannot change the priority of the Metronome RTGC. Metronome is set to run at a priority 0.5 higher than the highest priority of the RealtimeThread thread. We present a comparison of the two kinds of RTGCs regarding their suitability to support the Shadow Design Pattern.

The rest of the report is organized as follows. Sections 2, 3 and 4 describe the Shadow Design Pattern and the two kinds of RTGCs. Section 5 describes the experiment we performed on the Sun Java RTS 2.0 and the modifications made to the code for it to run correctly on IBM WebSphere Real Time. Sections 6 and 7 present our findings and conclusions, respectively.

2. The Shadow Design Pattern

The Shadow Design Pattern is a Java real-time design pattern for applications in which the goal of conforming to real-time constraints is more important than the computation of ultra-accurate numeric results. The pattern creates two threads per task, a refined, or accurate, computation and a coarse, or nominal, computation.

For easy reference, we will describe briefly in this section our Shadow Design Pattern for real-time applications reported in [9, 10], which is defined by the following two key features:

- Real-time threads are divided into two groups, with the threads in the first group having a priority higher than the one assigned to the RTGC and the threads in the second group having a priority lower than the one assigned to the RTGC.

- In case the set deadline is missed, a predetermined or approximate value (e.g., via table-lookup or most recent value of the approximation) is used as the result of the computation.

Figure 1 shows the participants of the design pattern and Figure 2 shows the interactions among the participants.

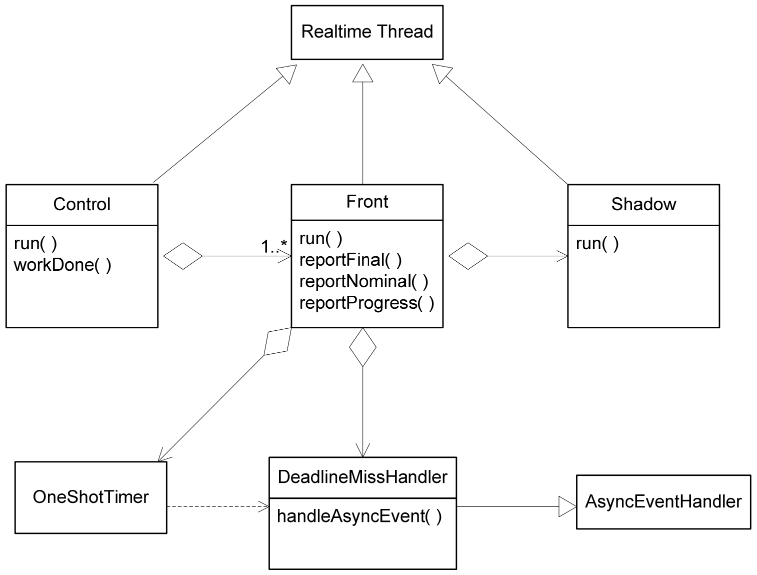

Figure 1. Participants of the Shadow Design Pattern

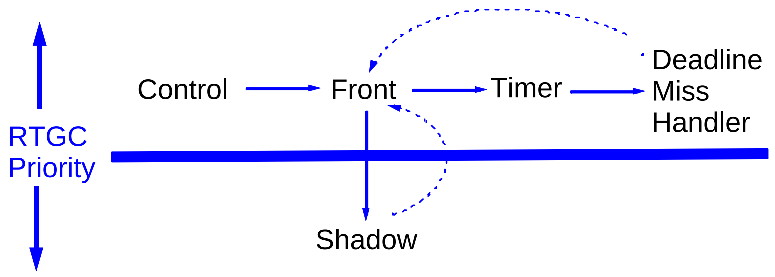

Figure 2. Interaction Among the Participants

The RealtimeThread, OneShotTimer and AsyncEventHandler are Java Real-time API Standard classes. The Control object is the main controller of the program. It creates the Front objects to carry out the time-constrained computations, and it destroys the Front objects when the computation is completed. The Front object in turn creates the Shadow object to carry out the detailed computations, a OneShotTimer object (with its duration equal to the deadline) to monitor the execution time of the computation performed by the Shadow object, and a DeadlineMissHandler object to perform the asynchronous transfer of control in case the Shadow object misses its deadline. The Front object then keeps track of updates from the Shadow object, and reports either the full result or the nominal result to the Control object when either the reportFinal() is called by the Shadow object or the reportNominal() method is called by the DeadlineMissHandler object.

The key requirement of the Shadow Design Pattern is that only the non-time-critical shadow threads consume unbounded amounts of memory in the heap. The time-critical front threads, which are responsible for the approximate solution, can only consume heap memory with known upper bounds on the maximum heap usage and the heap mutation rate (e.g., only performing simple table-lookup or keeping track of intermediate results using a fixed number of data objects), and the approximate solution must be obtainable in time strictly less than the deadline.

3. The Priority-based RTGC

A defining characteristic of a priority-based RTGC is the ability to adjust the priority of RTGC. By being able to adjust the RTGC priority, we can dictate the relative priority of the application threads. For a time-critical thread (that uses a heap), we can set its priority higher than the one assigned to the RTGC, so it will not be interrupted under normal execution. Such time-critical threads will only get interrupted when the amount of free memory goes below a certain threshold. At that point, RTGC preempts the time-critical thread in order to collect the garbage; this is achieved by temporarily increasing the priority of RTGC to a value higher than the one assigned to the time-critical thread.

The Sun Java RTS 2.0 supports a priority-based RTGC. By specifying the values for the runtime parameters RTGCCriticalPriority and RTGCCriticalReservedBytes, the programmer can control the behavior of the Sun RTGC.

3.1 RTGCCriticalPriority

The RTGCCriticalPriority runtime parameter is most significant in the Sun Java RTS 2.0 for ensuring the determinism of time-critical threads. A thread with the assigned priority higher than RTGCCriticalPriority is called the time-critical thread. The RTGC starts running at RTGCNormalPriority (whose default value is the minimum priority for the real-time threads). The auto-tuning mechanism attempts to start RTGC soon enough so that the garbage collection completes before reaching the memory threshold (RTGCCriticalReservedBytes), which will result in bumping up the priority of RTGC to RTGCCriticalPriority.

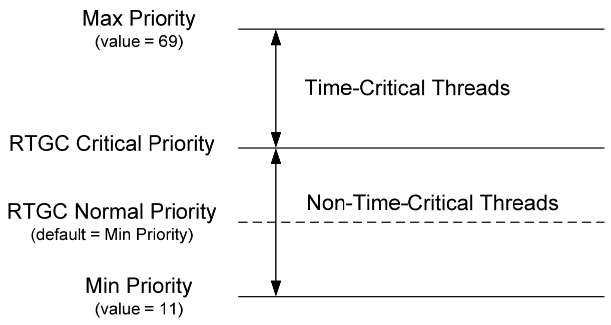

Figure 3. Classification of thread priorities

3.2 RTGCCriticalReservedBytes

To aid the RTGC in ensuring the deterministic behavior of all the time-critical threads, the programmer needs to specify the second runtime parameter RTGCCriticalReservedBytes (the default value is 0). When the free memory becomes less than the value set for the RTGCCriticalReservedBytes, the RTGC runs at the RTGCCriticalPriority, using all CPU cycles not used by the time-critical threads. This prevents all other threads (non-time-critical real-time threads and non-real-time threads) from allocating CPU cycles and memory, and causes them to be blocked. It is important to be aware that time-critical threads with a higher priority can still get blocked by the lower priority RTGC if there is not enough memory for the time-critical threads to run. In general, we want to set the RTGCCriticalReservedBytes just high enough to ensure that the time-critical threads do not get preempted by the RTGC due to lack of free memory. If RTGCCriticalReservedBytes is set too high, the RTGC will run more frequently, thereby preventing the lower priority threads from running. This will reduce the overall throughput. On the other hand, if we set it too low, then the deterministic behavior of time-critical threads can be compromised.

4. The Incremental RTGC

With a standard garbage collection, the complete activity is executed as a single process. During the garbage collection process, the application threads are suspended. As a consequence, an application program can be paused for a relatively long period of time. The length of this pause can be tolerated in non-real-time applications, but in real-time applications, the length of the pause time is critical to producing correct results.

Instead of running the garbage collection as a single process, it can be executed in a piecemeal fashion; this is known as incremental RTGC. For example, instead of running a garbage collector for 10 ms to complete the whole collection, we can run it ten times with each execution taking 1 ms. This piecemeal execution will result in pausing the application program for 1 ms instead of 10 ms.

The real-time garbage collector included in IBM WebSphere Real Time is called

Metronome, which is an incremental RTGC. A detailed technical description of Metronome can be found in [11]. One key difference between Sun's RTGC and Metronome is the ability to assign a priority to Sun's RTGC. The priority of Metronome is set by the system to be 0.5 higher than the highest priority real-time thread that uses heap memory.

4.1 targetUtilization

To fine tune Metronome, we can specify a number of runtime options. The most important one is the CPU *utilization target*. For example, the following command

```
java -Xrealtime -Xgc:targetUtilization=80 <myapp>
```

makes the application <myapp> run 80% of the time every 10 ms and the remaining 20% of the time is used by Metronome for garbage collection.

Metronome supports two types of garbage collection. The first is called the *heartbeat*, which is an incremental garbage collector. The second is called the *synchgc*, which is a standard synchronous garbage collector that stops the application until it completes the garbage collection. The synchronous garbage collection takes place under extreme conditions, such as when the heap memory is almost exhausted. We need to fine tune the application used in our experiment to avoid triggering synchgc.

5. The Experiment

The Shadow Design Pattern is motivated by the availability of priority-based garbage collectors, such as the one provided by the Sun Java RTS. The key aspect of the design pattern is that only the *Shadow* objects have the priority lower than the priority of the garbage collector (GC), and they are the only ones that may consume an uncertain amount of memory in the heap. The *Front* objects and others with the higher priority will not be interrupted by the GC.

We created a small test program (shown in Appendix A) to test the viability of the design pattern on a Sun BladeTM 2500 workstation (with a 1.6-GHz UltraSPARC IIIi processor with 1 MB of Level 2 cache, and 2GB RAM). For easy reference, we duplicate the description of our experiment we reported in [9] here.

The *Control* is implemented as a RealtimeThread and its run method is defined as follows:

```
public void run( ) {
 for (int i = 0; i < N; i++) {
  DataItem node = new DataItem(i);
  front[i] = new Front(this, i, node);
  frontCnt++;
  }
```

```
          for (int i = 0; i < N) {
               front[i].start();
               /* Point A - Place delay here */
          }
        }
```

We are using an array to keep track of the *front* threads. Every index position of this array is a non-null value as it points to an instance of the *Front* class. When a *front* finishes its computation, it calls the *Control*'s *workDone()* method to report the completion of the assigned task. This will result in setting the corresponding index position to null, thereby turning the used heap memory into garbage.

At Point A in the code, we can place a time delay after a *front* is started. Placing no delay means the program will run all *front* threads simultaneously. This could lead to an OutOfMemory exception when N, the total number of *fronts*, becomes larger than a certain threshold. The reason is that the priority of *Control* is higher than the one for RTGC. As *Control* creates and starts more and more *fronts*, memory gets consumed but there is no garbage to collect because there is no index position in the array that is set to null. In other words, the *front* threads never have a chance to call the *Control*'s *workDone()* method.

If we insert some delay at Point A in the code, then it becomes possible for the *fronts* to call the *Control*'s *workDone()* method to turn both themselves and memory allocated by the corresponding *shadows* into garbage for the RTGC to collect.

A *front* thread performs the computation on a given data item. The actual computation is done by its associated *shadow* thread. When the computation is complete, the *shadow* calls its controlling *front*'s *reportFinal()* method to report the full result, which will, in turn, cause the *front* to invoke the *Control*'s *workDone()* method.

The deadline is set by designating the time duration (RelativeTime that specifies the time duration such as 2 ms) using a *OneShotTimer*. When the time is up, its associated asynchronous event handler *DeadlineMissHandler* calls the *front*'s *reportNominal()* method to report the nominal result, which will, in turn, cause the *front* to invoke the *Control*'s *workDone()* method.

The *front* can get the result in two ways. The first is the full result, that is, the actual computation result received from its *shadow* via the *reportFinal()*. In this case, the *OneShotTimer* object is killed. The second is the nominal result. This result is used when the timeout occurs. In this case, the associated *shadow* object is killed.

With Metronome, these real-time threads (*Control*, *Front*, and all others) can be interrupted by the garbage collection activity. Two questions we need to ask are:

1. Would the Shadow Design Pattern continue to work properly under Metronome?

2. What would be the performance differences? Would the number of timeouts increase or decrease under Metronome?

To answer these questions, we ran the test program reported in [9] on IBM WebSphere Real Time.

5.1 ADAPTING THE TEST PROGRAM FOR METRONOME

We used the following hardware and software for running the experiement on IBM WebSphere Real Time:

- **Hardware**: LS20 Blade, 2 x Dual Core AMD Opteron(tm) Processor 275 @ 2.2-GHz, 8GB RAM. Bios: BKE123FUS-1.25. Model/Type: 885055U

- **Operating System**: IBM Real-Time Linux R1-SR2-GA (r868)

- **Java Real-time System**: IBM RealTime JVM (ibm-ws-rt-sdk-1.0-1.0-linux-i386.tgz)

The fixed-priority assignment of Metronome has occasionally caused null pointer exceptions in the Java program we used in our previous experiments with the Sun Java RTS. We observed this behavior with the following *run* method of the *Front* class in the original code:

```
public void run( ) {
  shadow = ...;
  timer = ...;
  timer.start();
  shadow.start();
}
```

We start the *timer* and then the *shadow*. This sequence works fine with Sun's RTGC because the priority of *front* is higher than the priority of GC, so this run method does not get interrupted by the GC.

With the Metronome GC, this sequence of execution can lead to a problem, that being an occurrence of a NullPointerException. Because the *Front* object is running in a priority lower than the one for the Metronome, it is possible that the set deadline is missed before this run method has a chance to start the *shadow*. Since the deadline is missed, the *reportNominal()* method of the *Front* object is called by the *DeadlineMissHandler*. Here is the portion of the *reportNominal()* method:

```
public synchronized void reportNominal( ) {
  ...
  shadow = null;
  timer = null;
  timeOutHandler = null;
```

```
      ...
    }
```

The variables are set to null so the garbage collector can reclaim garbage later. When the *reportNominal()* is called before the *shadow.start()* is executed, we get a NullPointerException because by the time the statement

```
      shadow.start();
```

of the run method gets executed, the variable *shadow* was already reset to null in the *reportNominal()* method.

The occurrence of this anomaly is not frequent, but such an exception should never happen. Garbage collection activity is certainly one possible cause of the run method being interrupted, but it is not the only cause. It is possible for the method to be interrupted by the normal thread scheduling; this is confirmed in one execution where there was no garbage collection activity, but the NullPointerException still happened.

Switching the order of calls to

```
      shadow.start();
      timer.start();
```

would solve the problem most of the time, but this sequence is not correct from the logical standpoint. It does not make sense to let the shadow start working before we set and start the timer. This sequence is problematic because the Shadow object gets to execute longer than the designated deadline.

A possible solution is to treat the two calls as an atomic operation as follows:

```
    synchronized(this) {
      timer.start();
      shadow.start();
    }
```

In general, we want to avoid the synchronization operation in the code because the operation introduces additional overhead that the system must be able to accommodate.

6. TEST RESULTS

We ran the modified code (that calls *shadow.start()* before *timer.start()*) with Metronome using different values for N (the number fronts), P (the pause time between the creation of fronts), and D (the deadline). In this section we compare the results to those we obtained with Sun 2.0 RTGC. We list the test results first and discuss the results at the end of the section.

Table 1. Pause time = 0 ms

Deadline (ms)	N (# of fronts)	Sun Result (# of timeouts)	IBM Result (# of timeouts)
20	100	79 ~ 100	0 ~ 99
	200	200	0 ~ 196
	500	500	0 ~ 499
	1000	1000	0
	1500	OutOfMemory	OutOfMemory
50	100	28 ~ 96	0 ~ 100
	200	142 ~ 200	0 ~ 200
	500	500	0 ~ 500
	1000	1000	0
	1500	OutOfMemory	OutOfMemory
100	100	0 ~ 60	0 ~ 77
	200	35 ~ 200	0 ~ 189
	500	500	0 ~ 428
	1000	1000	0 ~ 34
	1500	OutOfMemory	OutOfMemory
500	100	0	0
	200	0	0
	500	184 ~ 434	0
	1000	998 ~ 1000	0 ~ 660
	1500	OutOfMemory	OutOfMemory

Table 2. Pause time = 5 ms

Deadline (ms)	N (# of fronts)	Sun Result (# of timeouts)	IBM Result (# of timeouts)
20	100	0	0 ~ 9
	200	0	0
	500	0	0
	1000	0	0 ~ 5
	1500	OutOfMemory	0 ~ 2
50	100	0	0
	200	0	0
	500	0	0
	1000	0	0
	1500	OutOfMemory	0
500	100	0	0

	200	0	0
	500	0	0
	1000	0	0
	1500	OutOfMemory	0 ~ 77

Table 3. Pause time = 50 ms

Deadline (ms)	N (# of fronts)	Sun Result (# of timeouts)	IBM Result (# of timeouts)
20	100	0	0
	200	0	0
	500	0	0
	1000	0	0
	1500	1 ~ 5	0
50	100	0	0
	200	0	0
	500	0	0
	1000	0	0
	1500	0	0
500	100	0	0
	200	0	0
	500	0	0
	1000	0	0
	1500	0	0

Because any real-time thread can be interrupted by Metronome, we observe more frequent garbage collection activities as expected. However this does not lead to worse results (i.e., an increase in the number of timeouts). Even though the frequency of garbage collection activities increases, the occurrence of timeouts actually decreases under Metronome.

The data in Table 1 indicates that the number of timeouts reaches 100% for Sun's RTGC as the number of fronts increases for the same deadline; we reach the 100 percent timeout ratio sooner for the shorter deadline. For example, when D = 20 ms, we reach 100% timeouts when N = 200, but if D = 50 ms, we will not reach the same 100 percent timeouts until N = 500. When the deadline is shorter, the fronts do not have enough time to finish the task, and since they cannot be interrupted by the garbage collector with a lower priority, there will be fewer fronts we can run concurrently. With the shorter deadline and larger number of fronts, the system workload increases, and thus, results in more frequent timeouts.

Contrast these to the results of the IBM WebSphere Real Time. For the same set of parameters, we see the number of timeouts varies. For example, when D = 20 and N = 200, Sun's RTGC result is 100% timeouts, but Metronome's result ranges from 0 to 196.

The fronts will be interrupted by Metronome, but the interruption is short enough for the fronts to complete the task before the deadline. So compared to Metronome, the frequency of timeouts is much higher under Sun Java RTS, especially when the values for the pause time and the deadline are small (e.g., P = 0 and D = 20).

Tables 2 and 3 show the comparative results for P = 5 ms and 50 ms, respectively. As the values for the pause time increase, the differences between the two garbage collectors disappear. The reason is simple. Longer pause time implies less workload, and therefore, more "relaxed" time for the system to complete the given tasks. One marked difference we see in Table 2 is the OutOfMemory exception when N = 1500 for Sun's RTGC, but no such exception for Metronome; this is again due to their core difference. When there is a longer pause time (P = 5 ms), the incremental RTGC will have a chance to collect garbage. In Table 3, we see that there are no OutOfMemory exceptions with Sun's RTGC. When the pause time is increased to 50 ms, the intervals between the creations of fronts get long enough for the garbage collector to be triggered and reclaim garbage.

7. DISCUSSIONS AND CONCLUSION

To study the adaptability of our proposed Shadow design pattern to different types of real-time garbage collectors, we ran experiments with the IBM Metronome RTGC and compared the results against those we collected from the experiments with the Sun Java RTS 2.0. The results we obtained generally conform to our expectations. Although the use of the Metronome RTGC resulted in a few timeouts, we needed more care in producing the correct implementation of the design pattern with the Metronome as illustrated by the anomaly described in Section 5.1.

Our preliminary results shows that the Shadow Design Pattern works correctly under the two RTGCs. They work quite differently, so there is no generic algorithm which we can use to determine the right values for P, D, and N. We have to find those parameter values through empirical means for each type of RTGC. Note that the comparison of the two RTGCs is done strictly within the context of our Shadow Design Pattern. We do not make any claim on the relative merits of the types of RTGCs on the general cases.

One major threat to the validity of our experiment is that we do not know the relative speeds of the Virtural Machines (VMs); for example, our result would be skewed if Sun's VM ran significantly slower than the IBM's. More experiments are needed to run a benchmark suite such as the virtualization benchmark suite from Standard Performance Evaluation Corporation (SPEC) on both VMs to obtain a baseline throughput figure and make sure that timeouts are really caused by GC activity. Another threat is that the IBM experiment was run on a dual core, which may give Metronome an unfair advantage over Sun's RTGC since Metronome can do parallel GC (and does so by default). We need to re-run the experiments on comparable hardware platforms.

8. ACKNOWLEDGMENTS

The authors thank the IBM Metronome team and the Oregon Linux Center for their expert advice and allowing us to use their computing resources for running experiments. The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

9. REFERENCES

[1] Bollella, G., et al. Programming with non-heap memory in the real time specification for Java. In Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (Anaheim, CA, USA, 2003). ACM Press, New York, NY, 2003, pp. 361–369.

[2] Dibble, P.C. The Real-Time Java Platform Programming, Prentice-Hall, 2002.

[3] Wells, A. Concurrent and Real-Time Programming in Java, John Wiley & Sons, 2004.

[4] Kwon, J., Wellings, A., and King, S. Ravenscar-Java: A high integrity profile for real-time Java. In Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande JGI (Seattle, Washington, USA, November 2002). ACM Press, New York, NY, 2002, pp. 131-140.

[5] Laukkanen, M. Real-time Java—Memory Management. Seminar on Real-time Java, Department of Computer Science, University of Helsinki, April 2001.

[6] Pizlo, F., Fox, J., Holmes, D. Vitek, J. Real-time Java scoped memory: Design patterns, semantics. In Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (Vienna, Austria, May 2004), 101-112.

[7] Potanin, A., Noble, J., Zhao, T., and Vitek, J. A High Integrity Profile for Memory Safe Programming in Real-time Java. In Proceedings of the 3rd Workshop on Java Technologies for Real-time and Embedded Systems (San Diego, CA, USA, October 2005).

[8] Cook, T.S., Drusinsky, D., Michael, J.B., Otani, T.W., and Shing, M. Design of Preliminary Experiments with the Sun Java Real-Time System, Technical Report NPS-CS-06-010, Naval Postgraduate School, Monterey, CA, 2006.

[9] Otani, T.W., Auguston, M., Cook, T.S., Drusinsky, D., Michael, J. B., and Shing, M. A Design Pattern for Using Non-Developmental Items in Real-Time Java. In Proceedings of the 5th International Workshop on Java Technology for Real-time and Embedded Systems (Vienna, Austria, September 2007), pp. 135-143.

[10] Auguston, M., Cook, T.S., Drusinsky, D., Michael, J.B., Otani, T.W., and Shing, M., Experiments with Sun Java Real-Time System - Part II, Technical Report NPS-CS-07-005, Naval Postgraduate School, Monterey, CA, 2007.

[11] Bacon, D.F., Cheng, P., Rajan, V.T. The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems. In Meersman, R., Tari, Z. (eds.) Proceedings On The Move (OTM) Federated Conferences, OTM Workshops 2003. LNCS 2889, Springer, Berlin, 2003, pp. 466-478.

10. APPENDIX

In this appendix, we present the sketches of the Java code for our experiment.

• **Control**

```
public void run( ) {
  for (int i = 0; i < repeatCnt; i++) {
     DataItem item = new DataItem(i);
     Front front = new Front(this, i, item);
     dataStore[i] = front;
     //other bookkeeping tasks
  }

  RelativeTime delay = new RelativeTime(50, 0);

  for (int i = 0; i < repeatCnt; i++) {
     dataStore[i].start();
     try {
        RealtimeThread.sleep(delay);
     } catch (InterruptedException e) {
     }
  }
}

public synchronized void workDone
                 (int id, DataItem result) {
     dataStore[id] = null;
     //remove it, so it gets garbage collected

     //other bookkeeping tasks
}
```

• **Front**

```
public void run( ) {
  PriorityParameters scheduling
       = new PriorityParameters(
            PriorityScheduler.instance().
                getMinPriority());


  shadow = new Shadow(this,dataItem, scheduling);

  DeadlineMissHandler timeoutHandler =
                     new DeadlineMissHandler(this);
  timer = new OneShotTimer(new RelativeTime(
                     controller.getDeadline(),0),
                     timeOutHandler);
  timer.start();
  shadow.start();
}

public synchronized void reportFinal
                  (DataItem result ) {
  if (isActive) {
       isActive = false;
       timer.stop();

       //we got a full result from the shadow
       //so stop this OneShotTimer object

       timer = null;
       shadow = null;
       timeOutHandler = null;
       control.workDone(id, result);
  }
}

public synchronized void reportNominal( ) {
  if (isActive) {
       isActive = false;
       shadow.quit();

       //this kills the shadow by setting its
       //'isActive' to false.

       shadow = null;
       timer = null;
       timeOutHandler = null;
       control.workDone(id, nominalResult);
  }
}
```

```
public synchronized void reportProgress
                    (DataItem result) {
  //bookkeeping tasks
}
```

- **Shadow**

```
public void run( ) {

  while (isActive && i < 100) {
      //do work
  }
  front.reportFinal(result);
}
```

- **DeadlineMissHandler**

```
public void handleAsyncEvent( ) {
           front.reportNominal();
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52
   Naval Postgraduate School
   Monterey, CA

3. Research Office, Code 09
   Naval Postgraduate School
   Monterey, CA

4. Mr. Richard Ritter
   Missile Defense Agency
   Washington, DC

5. COL Scott LeMay, USAF
   Missile Defense Agency
   Washington, DC

6. Mr. Michael Young
   Missile Defense Agency
   Washington, DC

7. Ms. Denise Spencer
   Missile Defense Agency
   Washington, DC

8. Mr. Steve Hill
   Missile Defense Agency
   Washington, DC

9. Mr. Jan Young
   Missile Defense Agency
   Washington, DC

10. Ms. Genell Hausauer
    Missile Defense Agency
    Washington, DC

11. Mr. Wilson Varga
Missile Defense National Team
Crystal City, VA

12. Mr. Erik Stein
Missile Defense National Team
Crystal City, VA

13. Mr. Tim Trapp
Missile Defense National Team
Crystal City, VA

14. Dr. Butch Caffall
NASA IV&V Facility
Fairmont, WV

15. Dr. Doron Drusinsky
Naval Postgraduate School
Monterey, CA

16. Dr. Bret Michael
Naval Postgraduate School
Monterey, CA

17. Dr. Thomas Otani
Naval Postgraduate School
Monterey, CA

18. Dr. Man-Tak Shing
Naval Postgraduate School
Monterey, CA

19. Mr. Greg Porpora
IBM Federal Software

20. Dr. David Bacon
IBM T.J. Watson Research Center
Hawthorne, New York

21. Dr. Perry Cheng
IBM T.J. Watson Research Center
Hawthorne, New York

22. Mr. Theodore Tso
IBM Linux Technology Center

Medford, Massachusetts

23. Mr. Paul McKenney
IBM Linux Technology Center
Medford, Massachusetts

24. Dr. Greg Bollella
Sun Microsystems Software
Palo Alto, CA

25. Mr. Jay Magnino
IBM

26. Mr. Fred Weindelmayer
Naval Surface Warfare Center
Dahlgren, VA