Real-time Java applied in Automotive Systems

RENZHENG WANG

Master of Science Thesis
Stockholm, Sweden 2004

IMIT/LECS-2004-06

Master Thesis IMIT/LECS/ [2004 - 06]

Real-Time Java applied in Automotive Systems

Master of Science Thesis

In Internetworking

by

Renzheng Wang

Stockholm, June 2004

Supervisor: Robert Senger

Examiner: Vladimir Vlassov

Thomas Sjöland

Acknowledgements

I would first like to thank Dr. Vladimir Vlassov, Associate Professor, at the Department of Microelectronics and Information Technology (IMIT), KTH, my examiner who has always given me kind help and supports throughout this thesis.

I want to thank my co-examiner Thomas Sjöland, who arranged and examined my thesis presentation in KTH.

I also thank Robert Senger, my supervisor in BMW-CarIT, who has always given me a hand when I have had questions or got in trouble with my work.

Thanks to Dr. Thomas Stauner for reading through my thesis draft and giving me many valuable suggestions.

Thanks to Jie Tang, Oliver Noelle, Paul Hoser and all the colleagues in BMW-CarIT who have given me helps and supports in this half a year time.

I am grateful to my parents and my sister, who always encourage me through the phone from my homeland far away; I also thank my girl friend, Gefei, who is always beside me, sharing every piece of joy and bitterness in this half year with me. Without their love and support, I could not get through this thesis in the end.

Abstract

Most of the computer systems today, both hardware and software are all pursuing the best average-case utilization and performance of the system. However, the real-time system is one exception. It has goals fairly different from the conventional systems. In a real-time system, the temporal behavior of the system is regarded as the key issue, such as whether a task's deadlines can be met or not. Therefore in real-time systems, the predictability and worst-case execution time are much more important than the average performance. Such differences make the development work to build a real-time system highly distinct from other conventional systems. From hardware architecture to operating system, from software engineering to the programming language, all these subjects are given new criteria to evaluate when they are applied in real-time systems.

For the fast growing demand on real-time systems in the fields of modern telecommunication, aviation and automotive industry, real-time systems are now expected to take more complex and sophisticated tasks and be more efficient to build. The Real-time Java technologies are hence emerging in recent years aimed to make the Java language more suitable for building real-time applications. Thus many well-known advantages of Java are available to benefit and ease the development work of real-time systems.

This thesis focuses on the feasibility and applicability of real-time Java technologies in the domain of automotive real-time systems. It starts with the survey on the real-time system design theories. Some predominant scheduling theories such as the rate-monotonic scheduling theory and the deadline monotonic scheduling theory are introduced and analyzed here. Then the thesis compares the different real-time Java solutions theoretically by revealing both their conceptual advantages and drawbacks. Finally, a set of benchmark applications and an automotive real-time sample application are designed, implemented and deployed on each chosen solutions. The results are then compared and analyzed. Thus, the state of the art of real-time Java technologies applied in automotive system is evaluated and concluded.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Nowadays Java technologies have become more and more popular in the computer world for its platform independence, better reliability, fully object-oriented structure and flexibility of code reuse. All these good features also make the Java language a good candidate for building real-time applications in many embedded system development domains, such as automotive, avionic and industrial automation. But, to fulfill the requirement of such time-critical and cost-effective systems, some work still needs to be done to improve the predictability and runtime performance of classical Java. This is why the Real-time Java technology came into being during the recent few years. Currently most of the Real-time Java solutions have addressed and solved the main unpredictability in the classical Java virtual machine, which make Java unsuitable for the real-time systems, such as, long-time garbage collection delay, threads priority inversion etc. However, to what degree a specific real-time java solution could achieve its real-time performance to meet the demand of automotive systems is still in need of further investigation. This thesis analyzes, compares and evaluates the applicability of several Real-time Java solutions in the automotive systems, both in theory and in practice.

1.1 Background and Motivation

Nowadays billions of embedded systems, large or small, fast or slow, are widely used in almost every field of our lives, while a lot of others are just being investigated, invented and produced. Since more and more embedded systems have been applied into strictly time-critical or even safety-critical systems, the real-time characteristics of these embedded systems become more and more important. Therefore, in the automotive electronic system domain, a real-time development environment should be chosen for building the next-generation in-car system. Such a development environment should include hardware platforms, communication buses (or networks), operating systems, and programming languages together with its corresponding development tools. Among all these issues, choosing a better programming language plays an essential role because it will directly determine the whole technology set to be used and how the development team will be grouped or even established. This stringent demand formed the initial motivation of this thesis: to evaluate the real-time Java technology applied in the automotive systems. Some background knowledge including real-time system concepts, benefits of bringing real-time Java and the existing problems of typical Java platform will be briefly introduced in this section.

1.1.1 Real-time Conception

What does the term “real-time” exactly mean? There is usually a common misunderstanding, which implies that, a real-time system is just a system either ‘very fast’ or ‘immediately reactive’. But unfortunately, neither of these two vague definitions is correct. According to the definition proposed by Burns and Wellings [1], the term ‘real-time’ means “any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified delay”. This means the execution time of the real-time tasks can be not necessarily very short but should be

predictable and guaranteed within a specific deadline. According to the above definition, two kinds of real-time systems can be classified: hard real-time system and soft real-time system.

Within a hard real-time system, it is “absolutely imperative that responses occur within the specified deadline”, while in the soft real-time system, it is “response times are important, but the system will still function correctly if deadlines are occasionally missed”[1]. This means that in a hard real-time system, each of the task deadlines must be fulfilled; in case one failure occurs in any of such deadlines, it could possibly cause a catastrophic disaster and make the whole system unacceptable. One example of such systems can be the brake control system inside a car. On the other hand, a soft real-time system puts less pressure on such time constraints. All deadline fulfillments are of course preferred, however, a few deadline delays can be accepted. One example of such a soft real time system is the multi-media controller system.

1.1.2 Java For Embedded Systems

Most of the applications running in embedded systems today are developed in C or C++. Several explicit shortcomings exist in these languages, such as, complexity and error tendency brought by manual memory management, poor code portability among different systems, difficulties of choosing and using appropriate libraries, lack of security proving mechanisms etc. A very natural deduction will make people turn to Java, a well-structured object-oriented language. Compared with C and C++, Java has several explicit advantages, such as:

- Java can help developers to improve their productivities because of its high level of abstraction.
- Java’s grammar and semantics are highly refined, so it is not as complex as C++ and is easier to grasp.
- Java has a sophisticated security mechanism to prove system security.
- Java supports dynamic loading of new classes.
- Java is highly dynamic, supporting lots of objects/threads created at run time.
- Java supports component integration and reuse.
- Java language and platforms support application portability.

All the above features make Java a good candidate for the language used in embedded systems to bring more productivity, security, efficiency and portability.

However, Java is not perfectly defined for real-time applications, it still has several drawbacks which prevent it to be a real-time programming language.

1.1.3 Real-time Problems in the Typical Java Virtual Machine

1.1.3.1 Garbage Collection

One of the advantages brought by Java is that Java can relieve programmers from messy memory management work by providing a garbage collector mechanism that can automatically scan and

withdraw the discarded memory blocks. But apparently, such mechanism also brings the unpredictability of the execution time of the application, since the underlying garbage collector thread cannot be detected by the programmers and may become running at any time, preempting the user threads and causing unexpected delay.

So, to make Java a truly real-time language, a certain mechanism must be provided to make the garbage collector more predictable. Normally, this is the most delicate work to be carefully designed and implemented by the real-time Java producers.

1.1.3.2 Threads synchronization and resource sharing

Just like C++, Java also supports the multithread technology for more complex applications to be easily implemented. The monitor mechanism is supplied by Java, denoted by a keyword: synchronized, to solve the resource mutual exclusion problems among the threads. In real-time systems, such a mechanism could introduce other problems, such as: priority inversion.

Priority Inversion can be explained by the following example: Three Java threads H, M and L running together in a Java virtual machine, each of which respectively has the higher, medium and lower priority. Consider this situation: thread L enters a monitor first and will keep this resource until it finishes, then the thread H enters after L, it has to wait for the resource until L releases it; therefore it gives out the CPU and waits until the monitor is available. Here thread M comes and for thread M, nothing special prevents it from running. So according to the priority discipline, thread M will preempt thread L and finish its work first. Finally, the order of the finish time of three threads will be: M, L and H [2].

Priority inversion is apparently not a good situation we expected to see, because the thread with highest priority carries the most critical task and needs to run to end urgently, but when the above scenario appears, it will have to wait for all the threads superior to the monitor holder thread, which could possibly have the lowest priority. In order to make Java more real-time, such problems must be addressed.

1.1.3.3 Dynamic loading

One flexible feature Java provides is that, the classes needed at runtime can be loaded dynamically. This casts another problem that influences Java's predictability, because at any time during one thread running, there may be a new class, which has not been used and needs to be loaded. Since this part of work may include disk accesses, the total loading time can be hard to predict. This is also a potential threat to the real-time Java implementation.

1.1.3.4 Native method call

A very useful mechanism provided by Java is the ability to call the native code written in a platform dependent language like C or C++. It is called Java Native Interface (JNI). Just like the good features mentioned in this section, JNI also makes the real-time Java implementation harder to realize. This is because, when a certain Java thread calls some native code, the control of either the garbage collector

or a certain scheduler will be hardly able to reach such code, and consequently, more unpredictable events are likely to happen, outside the control of the developer.

1.1.3.5 Performance

Once again, as an interpreted programming language, the performance of Java must be considered when being ported into embedded systems for real-time development. Given the specific hardware platform, if a Java application is much slower than the C program that can finish the same task, this real-time Java solution will not be a good choice because it may not fulfill the short time constraints that the C program can, and therefore has to rely on hardware platform update.

1.1.3.6 Other issues

Besides all the aspects mentioned above, there are also some other issues that may cause some problems and need to be taken care of by the real-time Java solution provider, for instance, dynamic calls and type checking, asynchronous thread termination and so on.

1.1.4 Overview of Existing Real-time Java Solutions

As we can see from the previous section, it is really not easy to make Java real-time. While, fortunately, there are already many solutions carried out in this area. The most influential solutions are listed below:

- Specifications:
    - Real-Time Specification of Java[3], produced by The Real-Time for Java Expert Group (RTJEG) under Java Community Process (JCP) sponsored by Sun Microsystems.
    - Real-Time Core Extensions[4], produced by Real-Time Java™ Working Group in J Consortium
- Products:
    - Jamaica[5], real-time Java virtual machine produced by Aicas Company
    - PERC[6], real-time Java virtual machine produced by NewMonics
    - AJile[7] - aJile System: Java direct execution processor, a hardware real-time Java implementation

The above solutions more or less differ one from another. Their similarities and differences will be compared more in detail in the following chapters.

1.2 Summary of Contributions

Since there are already several real time Java solutions available, several questions are rising on the horizon: Is real-time Java today mature enough to be a possible solution for the automotive systems in

the future? If so, which of the real time Java technologies today is more suitable to be used to build real-time applications in automotive systems? This thesis unveils the answers of the above questions.

In general, the differences between these real time Java solutions can be summarized into the following aspects:

- Different focusing real-time domains. Some solutions focus on the hard real-time implementation, while some others provide the soft real-time ones.
- Different implementation strategies used to approach real-time Java. For example, to fight against unpredictable garbage collection work, RTSJ, PERC and Jamaica all have different strategies.
- Different dependencies on the underlying platform, such as some specific processor types or some specific operating systems etc.
- Different API scopes support, such as standard J2SE API, or J2ME CDC/CLDC, or some extra APIs for some specific functionalities
- Different resource requisites, such as, processor speed requisite, ram capacity requisite and so on
- Different complexity and flexibility due to various implementations and development environment provisions
- Different real-time capabilities like predictability, precision, reliability and so on
- Different performance provided, such as binary code size, running speed

In addition, the suitability of these solutions with typical environment for automotive development is an important criterion to measure in this thesis.

Based on the above measuring criteria, this thesis accomplishes the evaluation work by the following work steps:

- Theoretical analysis about the feasibility, complexity and suitability of applying real-time Java into automotive systems

  At this part of work, an overview of the real-time system theory, a requirement analysis for the automotive real-time development domain and a theoretical analysis about the existing real-time Java solutions are provided in this thesis.
- Design and implementation of a benchmark suite for evaluating the common real-time characteristics in the chosen Java virtual machines.
- Design and implementation of a sample automotive application that carries time-critical tasks. This sample application, on one hand, makes a synthetic evaluation of real-time java solutions, and on the other hand shows a way of doing real-time automotive applications in Java.

- Deploy the benchmark applications and the sample application on both non-real-time Java virtual machine and real-time Java virtual machine, collect the test results and evaluate the chosen Java environments by analyzing the results.

All of the above four steps of work will be described in detail in this thesis.

1.3 Thesis Layout

In the second chapter of this thesis, an overview of real-time system design is provided. The overview starts with an introduction to the characteristics of real-time system, and then one important research domain in real-time systems, scheduling analysis, is presented. Particularly, the fixed priority preemptive scheduling theory is discussed in detail for its natural suitability with Java language. Some practical issues concerning the fixed priority preemptive scheduling are discussed after that. At the end of the second chapter, some other important subjects in the real-time system design domain will be briefly overviewed including the hardware architecture, programming language and worst-case execution time estimation etc.

In chapter three, we will look into the current automotive system technologies. The basic characteristics of automotive system are introduced first. Then some predominant technologies used to build in-vehicle real-time systems, including the OSEK real-time operating system and Control Area Network (CAN) bus system, are then reviewed. To address the fast growing demand on building more complex and secure applications in the vehicle, some prospective technologies are listed at the end of chapter three.

In chapter four, theoretical analysis and comparison among the important real-time Java technologies today are deployed. The chosen real-time Java solutions to analyze include Real-time Specification of Java, PERC real-time Java virtual machine and Jamaica virtual machine.

Chapters five and six together describe the processes of creating the evaluation applications step by step. The applications include a set of benchmark applications and a sample automotive real-time application. The methodology, design and implementation issues are presented in a considerable order in these two chapters.

In chapter seven, the applications created in the previous stage of this thesis are deployed on the different Java platforms. One non-real-time embedded Java solution: IBM J9 and one influential real-time Java solution: PERC are both tested for a comparison. The test results are then listed and analyzed.

In chapter eight, the current situation of real-time Java technologies and their applicability in the automotive system domain are concluded based on both theoretical analysis and practical experiments accomplished in this thesis. The limitation of this thesis and future work in this area are also discussed in the final chapter.

The appendix and references are attached at the end of this thesis.

Chapter 2 Real-time System Design

In this chapter, firstly, the basic characteristics of real-time systems will be briefly introduced. Then, we will give an overview on the real-time scheduling theory, with more emphasis on the fixed priority preemptive scheduling, which is closer to the existing real-time Java solutions. For the next section, some practical concern on how to carry through a fixed priority preemptive scheduling analysis will be observed. The last part of this chapter will be remarks of some important issues involved in the embedded real-time system design.

2.1 Basic Characteristics of Real-time Systems

As we have introduced in Chapter 1, there are the two main types of real-time systems: hard real-time systems and soft real-time systems. The main difference between these two types of real-time systems is the different punishments they take for the possible missing deadlines. This section will begin with a further and more legible clarification of the difference, and then will introduce two approaches to a real-time system.

2.1.1 Hard, Soft Real-time and Safety-critical System

To aid the analysis and classification for the real-time systems, we import a time-utility function to distinguish different real-time systems [8]. The function shows the distribution of the system's utility through the time.

The time-utility function for a hard real-time system is depicted in Figure 2-1. The function shows that the system is able to obtain its utility if the task is finished after the start time and before the deadline, whereas, after the deadline, the utility will immediately go to zero, which means the whole system will become useless in that case.

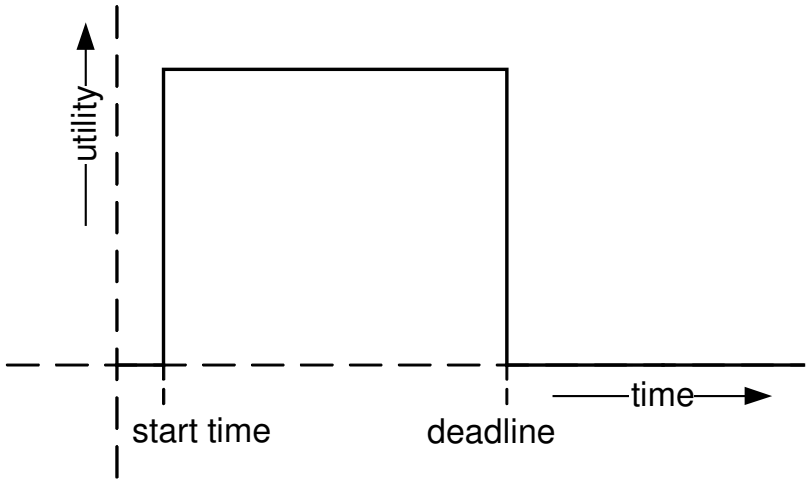


Figure 2-1 Time-utility function chart for hard real-time system[8]

The time-utility chart for a soft real-time system is shown in Figure 2-2. First, the system can surely get the utility when the task is finished between start time and deadline, moreover, it can also obtain some utility while the task finish time is delayed beyond the deadline. That means that the whole system can be still useful when a tolerant number of task deadlines are missed.

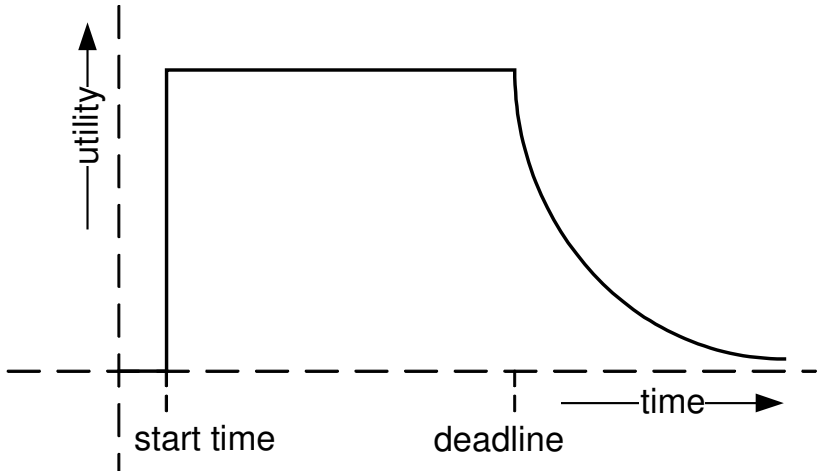


Figure 2-2 Time-utility function chart for soft real-time system[8]

Besides the soft and hard real-time systems, another type of real-time systems, which is also frequently mentioned, is the safety-critical system. A safety-critical system is a real-time system that carries real-time control tasks whose failure can directly lead a life-threatening impact. Such systems therefore have rigorous constraints about the temporal behaviors of tasks. Strictly speaking, this kind of system belongs to the hard real-time systems. But, the time-utility function of safety-critical system (Figure 2-3) shows its different characteristics from the common hard real-time system because of the severe consequence after the time failure.

As Figure 2-3 shows, if a task in a safety-critical system finishes before the start time or after the deadline, not only the whole system will lose all its utility, but also there will be actual damages (negative utilities) occurring to the system. Therefore, in most of the safety-critical systems, special concerns are taken to guarantee the time constraints in system, such as placing redundant hardware to achieve fault tolerance.

Furthermore, the hard real-time tasks and the soft real-time tasks do not always contradict each other. Sometimes, they can coexist in the same system. We call such a system a hybrid real-time system. Normally, within a hybrid real-time system, the hard real-time tasks execute at higher priorities, while, the soft real-time ones execute at lower priorities and make the best efforts on the rest of the resources in the system.

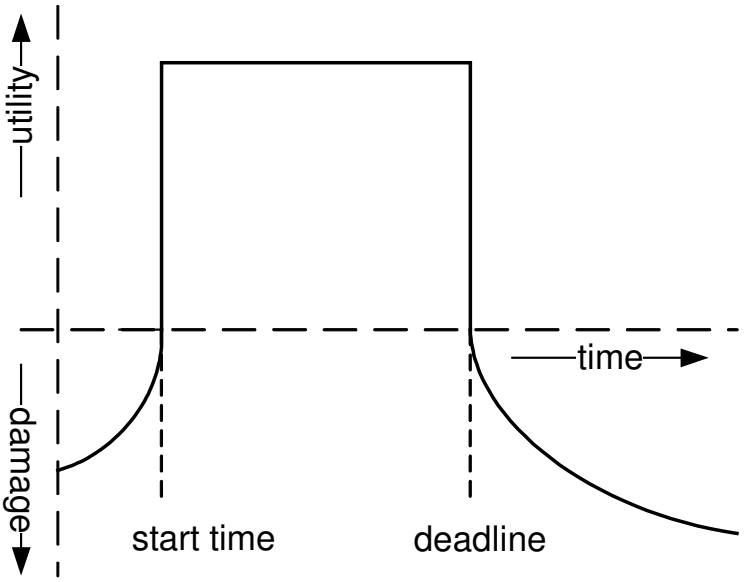


Figure 2-3 Time-utility function chart of safety-critical system[8]

2.1.2 Two approaches to the predictability

A natural question for the real-time application design domain is:

Can a single layer of the platform (such as an operating system or a real-time programming language platform) bring the real-time performance for the entire system?

The answer is, it depends. The following descriptions about two ways of approaching predictability may be able to answer this question more explicitly.

To achieve predictability of the system, there are primarily two different approaches: One is the layer-by-layer approach (microscopic) and the other one is the top-layer approach (macroscopic) [9].

The layer-by-layer approach requires that, from the lowest layer to the highest layer, each layer of the real-time system must be built to provide the predictability guarantees so that all the upper layers can rely on it. In such a way, the predictability of the whole system can be proved a priori. The system built by this approach can gain more deterministic time guarantee and hence behave more predictably. This layer-by-layer approach is suitable for building hard real-time systems, especially the safety-critical ones. However, if the target system is too sophisticated and complex, this approach may be too complicated to apply.

The top-layer approach can then complement the layer-by-layer approach. It only considers the behavior of the highest layer of the system and provides many mechanisms to prevent the missing of the deadlines. This approach apparently has less complexity, but it normally cannot a priori guarantee the real-time performance due to the unpredictable lower layers of the system. Therefore, the top-layer approach is more suitable for building soft real-time applications on top of a complicated system.

Now we can get a clear answer for the question raised in the beginning of this subsection. An individual tool or application, which is located at on the top of the system, may achieve the soft real-time performance, but due to the lack of predictability in the rest layers of the system, such environment cannot be used for the development of hard real-time applications. For doing the hard real-time applications, all the layers in the system should all support the temporal guarantees to prove the predictability of the overall system. This also indicates that, to build a hard real-time system, the choices made on the hardware platform, operating system, language runtime environment (for example, the Java virtual machine for the running of Java application) and real-time applications all influence the final real-time performance of the system. In the case of a distributed real-time system, special care also needs to be taken about the communication media's predictability. In the later part of this thesis, the layer-by-layer approach will be applied to analyze the predictability of the target system.

2.2 Scheduling Analysis

Because of the particular concern about the timing behavior, real-time application development requires a different software engineering method than those in the non-real-time software development domains. In a real-time application's development life cycle, the typical software lifecycle phases are given more meanings and responsibilities. Timing constraints of a real-time application must be taken good care of during all these phases of the development. From the requirement specification to the architectural design, from the detailed design to the implementation, the temporal issues should be considered and verified throughout the time. During the unit test phase and the system integration phase, the temporal behavior of the system should be inspected and verified more carefully. The result from these phases can be essential to prove the acceptability of the whole system.

In addition to these typical phases, a special analysis phase, called real-time scheduling analysis, is introduced into the real-time application lifecycle. The main purpose of this part of the work is to analyze, test and verify the applicability of the real-time tasks a priori before the design. The scheduling analysis work can greatly improve the reliability of the system and provide the theoretical proof for the practical implementation. Therefore, it plays an essential role in the real-time application development.

First, let us analyze the necessity of the real-time scheduling work by introducing a typical real-time task scenario, which represents the common problem to deal with in the real-time application development. Here we assume that these tasks are in a single processor environment, which indicates that all the tasks share the same processor and therefore a mechanism must be provided for sharing the execution time of this CPU among all the tasks.

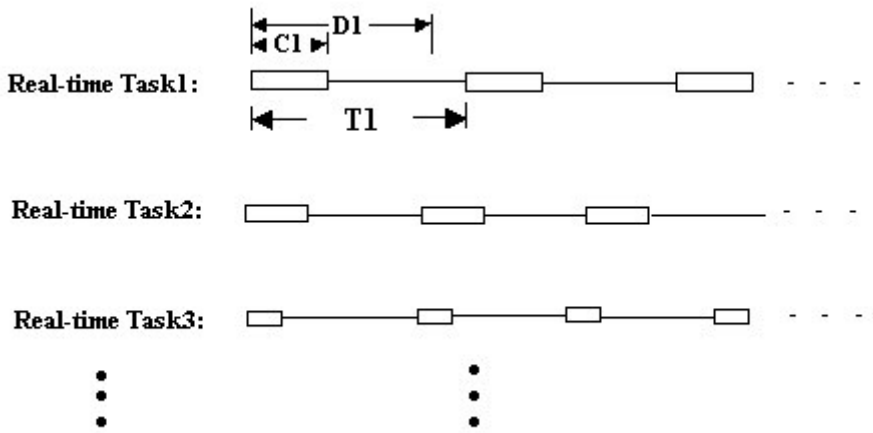


Figure 2-4 Real-time tasks scheduling problem

As shown in Figure 2-4, there are a few real-time tasks. Each of these tasks carries an amount of work to complete and has a predefined deadline to keep up with. For example, real-time task1 has a task release period of T1, a task deadline D1 and a task execution time of C1. In this specific scenario, the period for each task can be constant or uncertain.

To classify the real-time tasks further, we introduce three terms in the real-time system domain, which represent three different releasing behaviors. They are *periodic*, *aperiodic* and *sporadic*. The tasks, which have constant periods of release time, we call them *periodic* tasks. For *aperiodic* and *sporadic* tasks, both of them belong to the non-periodic tasks[10]. The differences between these two types of non-periodic tasks can be described as follows: *Aperiodic* tasks are those whose task invocation frequencies are unbounded, meaning that it is possible that, in a short period of time, several releases of one aperiodic task occur simultaneously. Aperiodic tasks make it theoretically inevitable that there may be more than one instance of the same task process coexisting in the system at some time. Whereas, *sporadic* tasks are those that have a maximum release frequency (that is to say, there exits a minimum interval between each two consequent releases of one sporadic task) such that only one instance of a particular sporadic process can be active at a time.

Since we have a certain set of these real-time tasks, periodic or non-periodic, we have to somehow manage the CPU time to share among these tasks, so that, any of these tasks' deadlines won't be missed. Given the specific conditions of all these tasks, such as periods (for periodic ones), execution times and deadlines, a scheduling method should be chosen to prove the possibility of running these real-time tasks on the target platform before implementation, and try to obtain as much schedulability and as less complexity as possible. For example, for the set of real-time tasks shown in Figure 2-4, suppose real-time task1 is periodic and has a period of 100 milliseconds, its deadline and task execution time respectively are 100 ms and 50 ms; real-time task2 is also periodic and has 50 ms period, 40 ms deadline and 30 ms execution time. In such a case, a simple scheduling analysis can be performed to prove the unfeasibility of this task set. Because, the total CPU utilization U_{total} should

be not less than U_{task1} (the expected utilization for task1)+U_{task2} (the expected utilization for task2), which is $\frac{50ms}{100ms}+\frac{30ms}{50ms}$ equals to 1.1; apparently, the CPU utilization in a single processor platform cannot exceed 1.0. As we can see, such analysis can reject an inappropriate task set in advance, without leaving the failure unchecked until the implementation, and thus can greatly help the whole development process. Furthermore, a real-time system without being proved by scheduling analysis can only rely on being verified by mass scale tests, which is certainly costly and can only provide empirical probability statistics. All these reasons can clearly show the necessity and great benefits of doing scheduling analysis in the real-time system development.

Based on the different task priority assignment strategy, the real-time scheduling methods can be classified into two categories:

1. Fixed priority scheduling:
2. Dynamic priority scheduling

Furthermore, if we classify the scheduling methods once again by the preemptability of the underlying runtime environment, we thus get three more specific scheduling method categories (the method of dynamic priority non-preemptive is seldom used):

1. Fixed priority non-preemptive scheduling
2. Fixed priority preemptive scheduling
3. Dynamic priority preemptive scheduling

For the first category of scheduling methods, within a fixed priority non-preemptive scheduling environment, each real-time task has its own fixed priority that will not change at runtime; the feature of non-preemptive requires that a task in the system, no matter whether its priority is higher or lower, is never allowed to preempt another task that is running on the processor. This kind of scheduling does not provide so much flexibility. If there is any demand to make the original task set expand after the design or implementation, this will lead to a lot of redesign work and in the worst-case, the whole net task set may have to be rescheduled again. Nevertheless, by applying such static scheduling, a real-time system can achieve very high predictability. Therefore, it is most suitable in the device that has limited computation ability and only handles a small set of real-time tasks with relatively simple logic.

On the contrary, the dynamic priority preemptive scheduling allows the priorities of the tasks in the system changing at runtime, and a task with higher priority can always preempt the lower priority task, which is executed on the processor. The scheduling method of Earliest Deadline First (abbreviated as EDF) belongs to this category. The general idea of this method is to check tasks' deadlines at runtime and rearrange their priorities according to their deadline. The task with the earliest or to say the most stringent deadline will get the highest priority, and therefore can preempt other tasks to run first. Such scheduling method brings much flexibility and almost exists independently with individual task sets. Moreover, such scheduling method can help the system achieve very high CPU utilization. However, such flexible algorithm also induces a problem that the runtime behaviors of the tasks are very difficult to adjust in case some exceptional purpose requires it.

Also, the predictability provided by the dynamic priority preemptive scheduling is not so reliable as the fixed-priority scheduling can provide.

The fixed priority preemptive scheduling lies in between the above two scheduling methods. The priorities of tasks stay constant during the runtime, and once a higher priority task is ready to run, it should always be able to preempt the executing lower priority tasks. Such scheduling method mixes good features of the above two scheduling methods and can therefore build a reliable real-time system with also sufficient flexibility.

Java, as an advanced object-oriented language, supports prioritized multithreads to be used in the user applications, and provides a preemptive runtime environment for the multiple threads running in it. Though the specification of the Java virtual machine does not explicitly oblige that the higher priority thread should be more eligible to run than the lower ones, most of the real-time Java solutions today implement their platform in this way. Therefore, among the above three scheduling methods, the most suitable one that can be well adopted in real-time Java platforms is the preemptive fixed priority scheduling. The following sections of this chapter will look into this scheduling theory more in detail.

2.3 Fixed priority Preemptive Scheduling Theory

In this section, first, some fixed priority preemptive scheduling knowledge basis will be introduced, and then, some important improvements made by the mature rate-monotonic scheduling theory: generalized rate-monotonic scheduling theory are presented; at last, the deadline-monotonic scheduling is briefly introduced.

2.3.1 Background Knowledge: Liu and Layland's Research in 1973

Early at 1973, Liu and Layland performed a successful research on the scheduling algorithms in the hard real-time environment[11]. Some of their results became the basis of many scheduling theory popularly used today, especially for the fixed priority preemptive scheduling domain, such as Rate Monotonic Scheduling (RMS) and Deadline Monotonic Scheduling (DMS). In this section, let us first examine some essential assumptions made in the beginning of their paper defining the problem domain, and then introduce the interesting conclusions made and proved in their research.

Assumptions:

I. *In a single processor, multiple real-time tasks are running in a preemptive way and share the processor time. The time for the processor to switch between tasks can be neglected.*

II. *All tasks are periodic, which means each task has its own constant period to be invoked.*

III. *The deadline of each task is the same as that task's period. That is to say, each task has to be finished before the next release of it.*

IV. *All tasks are independent to one another, which means that a certain task does not depend on the initialization or execution results of any other tasks.*

V. *The execution time for each task is constant, which means, the processor time needed by one task to finish its job without interference will not change in any period.*

From the assumptions above, Liu and Layland drew and proved the following important Theorems [11]:

Theorem 2-1 A critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks.

This theorem is quite useful when analyzing multiple periodic tasks. It forms a simple and worst case scenario, a so-called *critical instant* or a *critical time zone*, for the scheduling works to focus and analyze on. If the task set is proved to be schedulable in this time zone, it can be concluded that this task set can be scheduled in any other time period too.

Theorem 2-2 If a feasible fixed priority assignment exists for some task set, the rate-monotonic priority is also feasible for that task set.

This theorem proves the optimality of rate-monotonic scheduling given the assumptions mentioned previously.

Theorem 2-3 For a set of m tasks with fixed priority order, the least upper bound to processor utilization is $U = m \times (2^{1/m} - 1)$.

Note: here the processor utilization of a task set can be calculated as: $U = \sum_{i=1}^{m} \frac{C_i}{T_i}$, *where m denotes the number of tasks,* C_i *denotes the execution time of task i and* T_i *denotes for the period of task i.*

(One can refer to Liu and Layland's paper for the detailed proof of this theorem).

From this theorem, we can easily get the following practical conclusions:

For two real-time tasks, the upper bound of processor utilization will be $2 \times (2^{1/2} - 1)$, which is 0.83.

For three tasks, the factor would be 0.78.

If the number of the tasks turns to be a large one, the upper bound processor utilization will be close to $\lim_{m \to \infty} m \times (2^{1/m} - 1) = \ln 2$, which is about 0.69 in value.

In practice, a simple and efficient test can be deployed here to check the schedulability of a real-time task set.

I. If the processor utilization factor U= $\sum U_i = \sum \frac{C_i}{T_i} > 1$, the task set is not schedulable. A new revised task set needs to be defined, or more powerful hardware platform is needed.

II. If the processor utilization factor U= $\sum U_i = \sum \frac{C_i}{T_i} < 0.69$, this set of tasks is schedulable.

III. If the processor utilization factor U= $\sum U_i = \sum \frac{C_i}{T_i} < m \times (2^{1/m} - 1)$ where m denotes the number of the tasks, this set of tasks is schedulable.

IV. If the utilization is greater than $m \times (2^{1/m} - 1)$, an exact schedulability analysis is required to find a specific, more precise solution.

Though this quick schedulability test is rather rough, it is still useful for classifying the schedulability problems quickly.

The theorems founded by Liu and Layland are still not sufficient to solve many practical problems because of the strict assumptions that are made in front. But the results drawn in their paper still contributes a lot to the research in the hard real-time scheduling domain during the following thirty years. Most of the later researchers build their theories based on these useful results. Later in this chapter, some important improvements made for rate monotonic scheduling theory will be introduced. These improvements make rate monotonic theory and its close relative, deadline monotonic theory more and more mature in the scheduling analysis domain. Till now these two scheduling theories are widely used in the fixed priority preemptive system to analyze and design hard real-time applications.

2.3.2 Exact Completion Time Analysis for RMA

In 1986, Joseph and Pandya [12] founded a way of making exact analysis of schedulability of a real-time task set using fixed priority scheduling algorithm, and proved that this exact schedulability test is both sufficient and necessary. The idea of this exact analysis can be described as below:

Given the same assumptions as described in 2.3.1, a real-time task set contains several tasks to be scheduled. According to Liu and Layland's research result about the critical time zone, first, a leading theorem can be drawn as follows [13]:

Theorem 2-4 For a set of independent period tasks, if a task τ_i meets its first deadline $D_i \le T_i$, when all the higher priority tasks are started at the same time, then it will meet all its deadlines in the future with any other task start times.

For a certain task τ_i under its critical instant phasing, factor $W_i(t) = \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_i} \right\rceil$ describes the cumulative time for this task, where C_i denotes execution time of a task τ_i, and T_i denotes the period of that task.

According to the above theorem, it is not difficult to find that, τ_i will meet its deadline if $W_i(t) = t$ at some time t, where $0 \le t \le D_i$. Equivalently, a task will meet its deadline if and only if there is a t, $0 \le t \le D_i$, at which $\frac{W_i(t)}{t} \le 1$. The smallest t that satisfies this inequality is the worst-case completion time of t in any of its execution period. So the following theorem can be summarized:

Theorem 2-5 Let a periodic task set $\tau_1, \tau_2, \tau_3 \ldots \tau_n$ ***be given in priority order and scheduled by a fixed priority-scheduling algorithm. If*** $D_i \le T_i$***, then*** τ_i ***will meet all its deadlines under all task phasing if and only if:***

$$\min_{0 \le t \le D_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \le 1$$

The entire task set is schedulable under the worst-case phasing if and only if:

$$\max_{1 \le i \le n} \min_{0 \le t \le D_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \le 1$$

To compute and test using this theorem, one can simply create a sequence of times $S_0, S_1, S_2 \ldots$ where $S_0 = \sum_{j=1}^{i} C_j$, $S_{n+1} = W_i(S_n)$. If for the first n, $S_n = S_{n+1} \le D_i$, then τ_i is schedulable, and S_n is the worst-case completion time. If the tests show that, none of S_n can be found to fulfill the condition $S_n = S_{n+1} \le D_i$, task τ_i is not schedulable. This is what we often call a completion time test.

An example is put here to explain this completion time test method more clearly.

Real-time Task	Execution Time: C	Task Period: T
A	10ms	30ms
B	10ms	40ms
C	12ms	52ms

Table 2-1 Exact completion time test example

Given a set of real-time tasks A, B and C. Their execution time and task period can be found in Table 2-1.

According to the rate-monotonic scheduling rules, the priorities of tasks A, B and C are set in a descending order because of their ascending periods. Thus, we have: $P_A > P_B > P_c$.

Let us use the disciplines drawn from Liu and Layland's paper to do the preliminary schedulability test:

$U = U_a + U_b + U_c$ = 10/30 + 10/40 + 12/52 = 0.33 + 0.20 + 0.23 = 0.81 < 1

According to *Theorem 2-3*, when m= 3, the least upper bound processor utilization is 0.78 < 0.81. So we need a more exact analysis to check the schedulability of this task set.

For task A: $S_{a0} = 10,\ S_{a1} = W_a(10) = 10 = S_{a0}$. So, task A is schedulable.

For task B: $S_{b0} = 10 + 10 = 20;\ S_{b1} = W_b(20) = 10 + 10 = 20 = S_{b0}$. So, task B is also schedulable.

For task C:

$$S_{c0} = 10+10+12 = 32; S_{c1} = W_c(32) = 2\times10+10+12 = 42;$$
$$S_{c2} = W_c(42) = 2\times10+2\times10+12 = 52;$$
$$S_{c3} = W_c(52) = 2\times10+2\times10+12 = 52 = S_{c2}.$$

That is to say, at the worst case, task C will meet its deadline exactly by the end of its period. So, task C is schedulable too. Consequently, the whole set is proved schedulable.

2.3.3 Generalized Rate Monotonic Scheduling Theory

To relax the restrictions and limitations in Liu and Layland's research and bridge the gap between rate monotonic theory and the industrial development. Sha and Rajkumar founded the Generalized Rate Monotonic Scheduling theory (GRMS) and fixed an amount of practical problems[13].

2.3.3.1 Task Synchronization Problem and Solutions

First, according to the assumption *IV* made in Liu and Layland's research (2.3.1), tasks should be independent and never interact with each other. But for practical uses, this condition can hardly be fulfilled. So, GRMS puts the task synchronization issues into discussion.

First of all, to keep the consistency of the system, a mutual exclusion mechanism must be provided, such as semaphores, locks, monitors and so on.

For any of the mechanisms mentioned above, the priority inversion problem is inevitable. Once the priority inversion problem exists, the bounded execution time of a task involved in synchronization cannot be estimated or is too pessimistic to analyze the schedulability. To settle this problem down, two approaches were raised in GRMS: the priority inheritance protocol and the priority ceiling protocol[14]. In these two protocols, the priority ceiling protocol is derived from the priority inheritance protocol and provides several advantages in comparison to it. We will introduce the ideas behind these two approaches and compare their similarities and differences. Additionally, the reason why we pay so much attention to these two mechanisms is that, some existing real-time Java solutions claimed that, they had priority inversion mechanism supported in their system to prevent priority inversion, while, some of others support priority ceiling for the same purpose. To better evaluate these real-time Java solutions, this thesis will analyze these two protocols more in detail in order to attain a deeper understanding of them.

Priority Inheritance Protocol

The following criteria define the priority inheritance protocol:

1. The mutual exclusion resource is guarded by a binary semaphore[1] in order to keep the system's consistency. The semaphore has two states: locked and unlocked. When no task accesses the resource, the semaphore is in an unlocked state. If one task gets the lock and accesses the resource, the semaphore will be set in the locked state. Meanwhile, if there is another task applying for the access of the same resource, it has to be blocked and wait in the queue for that semaphore. The queue is organized in a prioritized way so that the task that has the highest priority will be invoked first after the semaphore is unlocked. Tasks with the same priority will be invoked in a First Come First Serve way (FCFS).
2. The priority of task T will be raised if a lock it holds blocks another task with a higher priority. Task T will then be given a higher priority, which is the same with the blocked task (This is why we call it a priority inheritance protocol). The temporary priority raised period will be from the time the higher priority task is blocked until task T finishes its access to the mutual resource (The period of accessing the mutual resource is also called the critical section of this task to that mutual resource). After the priority inheritance period, the task's priority will fall back to its original value.
3. Priority inheritance is transitive. For example, suppose that tasks T1, T2 and T3 respectively have the higher, medium and lower priority. If T3 blocks T2, and T2 blocks T1, T3 will inherit the priority of T1 via T2.
4. The operations of priority inheritance and resumption must be indivisible to keep the consistence of the runtime system.

Now let us look at the priority inversion scenario mentioned in the first chapter in the priority inheritance mechanism.

Three tasks H, M, L respectively has the higher, medium and lower priority. Task L applies to access a mutual resource first, since no previous task is accessing the resource at the time, the semaphore on this resource is still unlocked. So L is given the lock and starts to access the resource. Before L finishes its work in its critical section, task H is invoked and starts to run. It preempts L from running because of the higher priority it has. After H runs for a while, it also applies to access the same mutual resource that L accessed. Because L still holds the lock, task H is blocked and yields the processor. According to the priority inheritance protocol, task L's priority is raised to the same priority with H, and L goes on its work in its critical section. Meanwhile, task M is invoked and starts to run. Since its priority is less than the temporary priority that L has, it has to wait L to finish its work on the mutual resource. After L finishes the resource access, it gives up the lock and returns to its original priority, and at the same time, task H is awakened from the queue of the resource. It then preempts L, locks the semaphore and works on the resource. After H finishes its work, task M runs and finishes its work. At last, task L awakes again and finishes at the end.

As we can see here, priority inversion is effectively prevented by the priority inheritance protocol. But two problems still remain in the system using the priority inheritance mechanism.

---

[1] We just choose semaphore as a sample mutual exclusion mechanism; in practice, this can also be monitor, rendezvous or any other effective mutual exclusion mechanism.

First, the priority inheritance protocol cannot prevent the deadlocks. For example, if task T1 and T2 both need to access mutual exclusion resource S1 and S2, and T1's operation order is: lock S1, lock S2, release S2, release S1, while T2's operation order is: lock S2, lock S1, release S1, release S2. We can easily prove that a scenario can possibly occur where T1 holds the lock of S1 and T2 holds the lock of S2, and both of them wait for the other resource and they are both blocked. A deadlock is then formed.

Though the problem of deadlock can be solved by, for example, imposing a rule of totally ordering resource accesses. Still, a second problem exists. That is, a chain of blocking can be formed. For instance, tasks H, M, L still represent the tasks with descending priorities. H needs to access resources S1 and S2. But before it starts, task L grasped the lock of S1 and was preempted by M, which entered a bit later and got the lock of S2. If H is invoked at this time, it has to wait for L's critical section on S1 and M's on S2 after that. Priority inheritance cannot help in this case. Thus, a blocking chain is formed.

Priority Ceiling Protocol

To solve the two problems existing in priority inheritance systems, the priority ceiling protocol is thus invented. The general idea about this protocol is to ensure that when a task T preempts the critical sections of other tasks and wants to enter its own critical section, it has to have a higher priority than the priority ceilings of all the preempted critical sections to get the permission. The priority ceiling of a mutual resource denotes the priority of the highest priority task that may use the resource. Within the priority ceiling protocol, a task T is allowed to start a critical section only if T's priority is higher than all priority ceilings of all the mutual resources locked by other tasks, otherwise, it will be blocked and wait in the queue of the resource it applied for, and the preempted tasks that cause the block of T will inherit T's priority.

It is easy to prove that the priority-ceiling protocol can also prevent the priority inversion. Now we will examine how the priority-ceiling protocol prevents the deadlocks and also avoids the blocking chain problems.

For the deadlock problem, we use the example that described the situation. We assume that the priority of task T1 P1 is greater than that of task T2 P2, and for resources S1 and S2, no tasks other than T1 and T2 will use them. So they both have a priority ceiling of P1. Suppose that T2 starts first and applies to lock S2, since there is no other critical section existing at the moment, T2 gets the lock of S2 and executes its operations inside its critical section for S2. T1 then starts to run and preempts T2, when the time T1 needs for the lock of S1, priority ceiling protocol will compare the priority of T1 (P1) with the priority ceiling of the preempted locked resource S2, which is also P1. Because the priority of T1 is no higher than P1, T1 is thus blocked and T2 inherits the priority of T1 and continues its operations in the critical section. After T2 finishes accessing S2, S1 and unlocks them, its priority falls back to original so that T1 will get back to run. As we can see, deadlock is successfully prevented by the priority ceiling protocol.

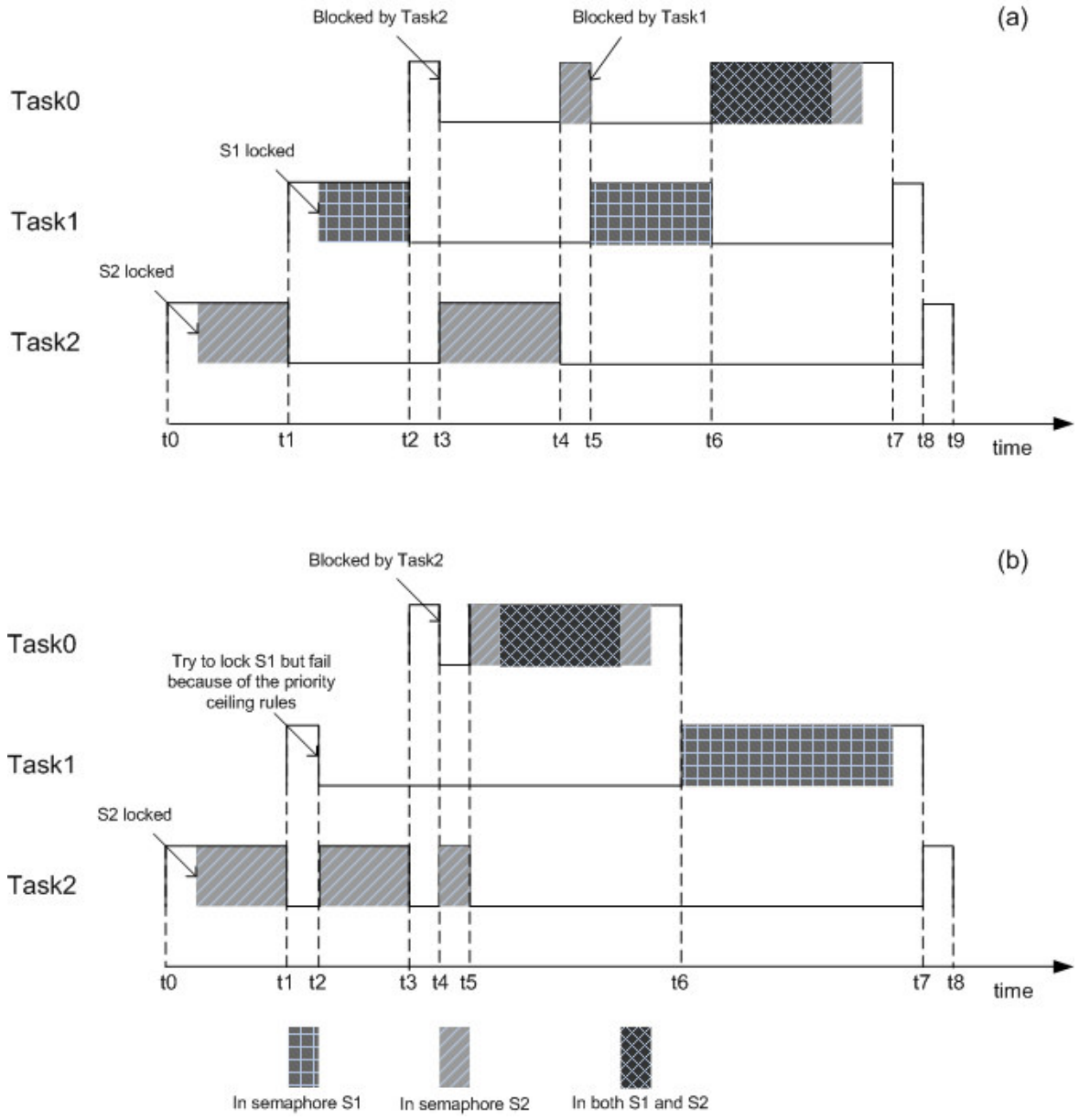


Figure 2-5 Example to show how priority ceiling protocol solve the blocking chain problem

Figure (a) shows the problem in priority inheritance protocol; Figure (b) shows how the same circumstance under priority ceiling protocol

For the blocking chain problem, consider the previously mentioned example. In that case, the priority ceiling protocol comes into play when task M enters and tries to lock S2. Since S1 has been locked by task L, and S1's priority ceiling will be the priority of H, which is higher than M's priority, thus M will be blocked and L will inherit its medial priority. When task H starts and asks for the lock of S1, it

just needs to wait for one critical section of task L in S1, and then it can get the resources it needs to run on. Figure 2-5 describes more clearly how priority-ceiling protocol solves the blocking chain problem. For a more strict proof of these properties of priority ceiling protocol, one can refer to the published paper[14]

After the introduction to the two protocols, we will now turn back to the scheduling analysis and discuss the schedulability issues after the synchronization among tasks is considered. Differences of priority inheritance protocol and priority ceiling protocol will be shown in this section.

According to the results of Sha, Rajkumar and Lehoczky's study on priority inheritance and priority ceiling protocols[14], these two protocols have been proved to have the following scheduling properties.

Theorem 2-6 Under the priority inheritance protocol, give a task T0 for which there are n lower priority tasks {T1, T2… Tn}, task T0 can be blocked for at most the duration of one critical section in each of the blocking sets $\beta_{0,i}$ ***(where*** $\beta_{0,i}$ ***refers to the set of the longest critical sections of Ti that can block T0)***

Theorem 2-7 Under the priority inheritance protocol, if there are m semaphores which can block task T, T can be blocked by at most m times.

Given the above two theorems, a worst-case blocking duration for one task can be calculated. For example, if there are four semaphores that can potentially block task T and three lower priority tasks, T can be blocked for a maximum duration of three longest critical sections of the three lower priority tasks.

Theorem 2-8 Under the priority ceiling protocol, a task Ti can be blocked for at most the duration of one element of Betai (where Betai refers to the sets of the longest critical sections that can block Ti)

As we can see, the worst-case blocking time of a task under the priority ceiling protocol can also be calculated, and the bound time is well optimized rather than the pessimistic result calculated under the priority inheritance protocol.

Given the above theorems, the following extended theorems for Rate Monotonic Scheduling theory can be drawn.

Theorem 2-9 A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm if the following conditions are satisfied:

$$\forall i, 1 \le i \le n,\ \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \ldots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1)$$

Theorem 2-10 A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm for all tasks phases if

$$\forall i, 1 \le i \le n,$$

$$\min_{(k,l)\in Ri}\left[\sum_{j}^{i-1} U_j \frac{T_j}{lT_k}\left\lceil\frac{lT_k}{T_j}\right\rceil + \frac{Ci}{lT_k} + \frac{B_i}{lT_k}\right] \le 1$$

In both of the above theorems, C_i and T_i denote the execution time and task period of task τ_i. U_i is the utilization of task τ_i. B_i is the worst case blocking time for τ_i, and $R_i = \{(k,l) \mid 1 \le k \le i, l = 1,....,\lfloor T_i / T_k \rfloor\}$.

Theorem 2-10 can also be described in a more convenient way by importing a parameter W_i. W_i here denotes the window (time interval) starting from the release of task τ_i, which we attempt to insert the computation time of tasks into. A formula for W_i can be set up:

$$W_i^{n+1} = B_i + C_i + \sum_{j\in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil C_j \qquad (2.1)$$

Thus, a similar exact completion time test as illuminated in 2.3.2 can be used here too.

2.3.3.2 Aperiodic and Sporadic Events Handling

After the task synchronization issues considered, GRMS also addresses another restriction in the assumptions in Liu and Layland's research. That is, how to handle the aperiodic and sporadic tasks as well as the periodic tasks.

There are several algorithms proposed for solving the above problems. Each of them has their own advantages and drawbacks. A brief overview on these algorithms will be presented here.

Two common and relatively simple approaches among them were proposed in the early stages. They are background processing and polling tasks. The idea behind the background processing approach is that the arrived aperiodic events will be pending in the system queue until the processor gets idle time after executing periodic tasks. While the polling strategy is to construct a periodic task, which keeps polling aperiodic events in a fixed rate; when the polling task comes into execution and there are no pending aperiodic tasks pending, it will yield its execution time to the other periodic tasks and suspend until the next period. These two strategies are simple to implement, but their drawbacks are evident. The background processing approach won't have any guarantees on the aperiodic events being served in time when the utilization of processor is high, and the polling approach will give a long average response time if its period is set long; if its period is set too short, it will be a waste of execution time. Therefore, these two approaches are only suitable to serve for the aperiodic or sporadic events which are soft real-time or do not have any time constraints at all.

To cater for the demand of serving hard real-time aperiodic and sporadic events, later on, three recommendable approaches were raised: priority exchange server, deferrable server and sporadic server. The main idea of the first two approaches is to preserve an amount of processor time for aperiodic events handling and replenish this execution time periodically. Their difference lies on the priority assignment for the handling server. For the priority exchange server, as its name indicates, it will exchange its priority with the coming periodic task if there are no aperiodic events coming. In an extreme case, the priority of a priority exchange server will decrease from the highest priority it has to the bottom priority when there is no event arriving within the whole period. For the deferrable server, it keeps its high priority throughout the time. Once an aperiodic event arrives, it will be served as a high priority task and could possibly preempt the current periodic task.

The sporadic server approach, raised by Sha et al in 1989[15], gives a more sophisticated strategy with special concern about the possible burst releases situation of the aperiodic events (several aperiodic tasks arrive simultaneously). The main idea in this approach is that, a high priority task for servicing aperiodic tasks is created, and the sporadic server preserves its server execution time at its high priority level until an aperiodic request occurs. The server replenishes its execution time after some or all of the execution time is consumed by aperiodic task execution.

However, none of these three approaches can guarantee the deadlines of aperiodic events, since theoretically there can a particular time period when an arbitrarily large amount of events arrive and cause the execution time insufficient to meet all of their deadlines. But in case of sporadic events, there is always a minimum interval between two events' arrival, the schedulability could therefore be analyzed in a worst-case to guarantee their deadlines being met. It has been proved that a periodic task set that is schedulable with a task, T, is also schedulable if T is replaced by a sporadic server with the same period and execution time. For more details about the sporadic server such as performance and implementation issues, one can refer to the doctoral thesis of Brinkley Sprunt[16].

2.3.3.3 Earlier deadline issue in rate monotonic theory

According to the assumption *III* that made in Liu and Layland's paper (2.3.1), the deadline of a periodic task should always be the same with its period. However in the practical point of view, this assumption can be hardly fulfilled in the development in real world. Quite a few realistic tasks demand a shorter deadline than their release periods. Though, to address this problem, deadline monotonic scheduling analysis is more eligible, there is still a different approach to fit this situation into the rate monotonic scheduling analysis theory. Let us give a glimpse of this approach and look into the deadline monotonic theory in the next section.

Suppose that a task t's deadline D is before the end of its period T. Let E=T-D. That is, the task t has a deadline earlier than E. We can consider this in another way that the task has an end of the period deadline but has an extra blocking time by lower priority tasks for a duration of E. So the effect can be modeled as if task t's utilization is increased by E/T. Combining the theorem 4, a following theorem can be drawn:

Theorem 2-11 A set of n periodic tasks scheduled by the rate-monotonic algorithm will always meet its deadlines, for all task phases, if:

$$\forall i, 1 \le i \le n, \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i + B_i + E_i}{T_i} \le i(2^{1/i} - 1) \qquad (2.2)$$

The completion time test introduced in the earlier sections can be extended here to add parameter E into the formula and get a more accurate schedulability test.

2.3.4 Deadline monotonic scheduling theory

In practical industrial real-time systems, some tasks in the system can possibly have such characteristics: the deadline of the task is less than its period. For the reason that rate monotonic scheduling theory always demands the period of a task to be equal to its period, so RMS is not suitable any more in this case. A similar scheduling theory, deadline monotonic scheduling theory was then proposed by Leung in 1982[17]. The main idea of this theory is that the priorities of tasks can be assigned according to their deadlines instead of their periods; the task that has a shorter deadline will be assigned a higher priority. Particularly, when the all tasks' deadlines are equal to their periods, the deadline monotonic scheduling will have the same effect as the rate monotonic scheduling.

The closeness in concepts between RMS and DMS makes their properties and analysis methods quite similar. So in this section, we only bring some interesting improved results given in Deadline Monotonic scheduling theory.

First of all, for the deadline monotonic scheduling theory, *Theorem 2-1* drawn from RMS in the previous section about critical instance can be also adopted here. In fact, the theorem is applicable to any fixed priority preemptive scheduling theory. And for *Theorem 2-2*, it can also be proved that if a feasible priority assignment exists for some task set, the deadline monotonic priority assignment is feasible for that task set.

Similar to the exact analysis in RMS, deadline monotonic scheduling also has a sufficient and necessary schedulability test. And it can be described as a practical algorithm written in pseudo-code below (the algorithm did not put the blocking factor into consideration):

Algorithm

foreach τ_i *do*

$t = \sum_{j=1}^{i} C_j$

continue = *TRUE*

while [*continue*] *do*

if $\left[\frac{I_i^t}{t} + \frac{C_i}{t} \le 1 \right]$

continue = *FALSE*

/* *NB* τ_i *is schedulable* */

else

$t = I_i^t + C_i$

endif

if $[\, t > D_i \,]$

exit

/* *NB* τ_i *is unschedulable* */

endif

endwhile

endfor

Algorithm 2-1 Schedulability algorithm in Deadline-Monotonic Scheduling[10]

where

$$I_y^x = \sum_{z=1}^{y-1} \left\lceil \frac{x}{T_z} \right\rceil C_z$$

In addition, the algorithm defined above can also be used to guarantee the deadline of any sporadic events in the system. Since every sporadic event should have a minimum interval between two releases, so, in a worst-case, one sporadic handler can be taken as a periodic task with the period equal to the minimum interval. The priority of a handler can be assigned according to the deadline of the corresponding sporadic event. Therefore, even without the aid of a sporadic server, the sporadic events with hard deadline still can be scheduled using deadline monotonic scheduling.

2.4 Practical Concerns about Fixed Priority Preemptive Scheduling

Besides the issues of task synchronization, non-periodic events handling and shorter deadline than period mentioned in the previous sections, there are still many practical issues not addressed in practice. It seems that the gap between scheduling analysis and development work in the real world is still large.

As the general theoretical background introduced in the previous sections, this section will do more analysis on the practical issues of priority preemptive scheduling, and take them into consideration when scheduling tasks, so that the theoretical analysis work of this thesis could help and conduct the rest of the thesis.

PLATFORM REQUIREMENTS

The implementation of preemptive priority based scheduling raises a set of explicit demands to the underlying runtime environment.

1) The runtime environment should provide a preemptive multi-tasks scheduling mechanism with a sufficient range of priorities.

A sufficient range of priorities here means that the range of priorities in a specific platform should be well enough to fulfill the requirements of most real-time applications running on it. For instance, POSIX OS standard demands its implementations to have 65 unique priorities: 0 to 64; While, in real-time specification of Java, the scheduler is required to have 28 unique levels of priority.

2) The context switch overhead must be bounded

In practice, multiple tasks in a single processor will be scheduled in different queues so that they can get the processor time for running according to the corresponding scheduling algorithm. The time overhead to move one task from one queue to another, for example from the delay queue to the run queue, cannot be neglected. Though, this overhead is always related to the number of the tasks in queues, as a real-time application platform the underlying system should have a way of predicting the time cost for context switch before runtime. Thus, during the scheduling analysis the worst-case context switch time can be taken into calculation and help to get the realistic result.

3) Predictable tick driven scheduler overhead

The most common strategy of proving timing behavior of both periodic tasks and non-periodic tasks is to use a tick driven background scheduler, which is in charge of invoking periodic tasks and polling non-periodic events. The execution time for one operation of such a scheduler can be trivial, but considering its constant behavior, this overhead should be counted in the scheduling analysis as well. Hence, the overhead caused by tick driven scheduler should be predictable.

In summary, the key attributes of the underlying platform are summarized and listed in the following Table 2-2:

Notation	Description
C_{sw}^{P}	*Cost of context switch away from a periodic task - may be a function of maximum size of delay queue*
C_{sw}^{R}	*Cost of context switch to a task currently in the run queue*
C_{CLK}	*Clock interrupt handler cost (no tasks being moved)*
T_{CLK}	*Clock interrupt handler period*
C_{PER}	*Cost of moving one task from the delay queue to the run queue*
C_{sw}^{S}	*Cost of context switch away from a sporadic task – when it suspends waiting for its next release*
C_{SP}	*Cost of releasing a sporadic task (i.e. putting it on the run queue)*
C_{INT}	*Cost of an interrupt handler that just releases a sporadic task*

Table 2-2 Key attributes of the real-time platform

Suppose that the above parameters in the platform can be predicted or calculated by some means. The next step of work will be applying these parameters into the rate monotonic scheduling formula we got in the previous sections.

CONTEXT SWITCH OVERHEAD

After examining the behavior of each task from its invocation to its suspension, we find that two context switches are involved. One is to move a task from the delay queue to the run queue and activate it, and the other is to move the task away from the run queue back to the delay queue. Thus, we can modify the formula (2.1) into:

$$W_i^{n+1} = B_i + C_i + C_{sw}^{P} + C_{sw}^{R} + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil (C_j + C_{sw}^{P} + C_{sw}^{R}) \qquad (2.3)$$

TICK DRIVEN SCHEDULER OVERHEAD

In a tick driven scheduler, the scheduler acts as a periodic task with a constant period T_{tick}. We use C_{tick} to denote the bounded execution time of each period T_{tick}. Then to add tick overhead into scheduling analysis, we can extend the formula 2.1 into:

$$W_i^{n+1} = B_i + C_i + C_{sw}^{p} + C_{sw}^{r} + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil (C_j + C_{sw}^{p} + C_{sw}^{r}) + \left\lceil \frac{W_i^n}{T_{CLK}} \right\rceil C_{CLK} \qquad (2.4)$$

For the realistic requirement of the real-time applications, there are some other variations of the scheduling formula. We concluded them as follows:

RELEASE JITTER CONCERN

One periodic task is considered to have exact constant period all along in the above analysis. However in practice, this is not always true. The invocation of a periodic task can be delayed by the tick driven timer or the handling overhead of a periodic event, we say that this task suffers from a release jitter. During the design of the real-time application, the release jitter overhead must be considered and kept in a bounded time. Let J_i represent the worst-case release jitter suffered by task τ_i. R_i represents the response time for task τ_i. We can get:

$$R_{i_{TRUE}} = R_i + J_i$$

Considering a release jitter can practically make the time between two releases of one task narrower than its real period, we can get such a variation of formula 2.2:

$$W_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n + J_i}{T_j} \right\rceil C_j \qquad (2.5)$$

ARBITRARY DEADLINES

In case that the deadline of a task is greater than its period, none of the previous discussion can handle this situation. Since when deadline is less than (or equal) to the period, it is only necessary to consider a single release of each task: the critical instant, when all higher priority tasks are released at the same time. But when deadline is greater, more releases other than one must be considered. We assume that the release of a task will be delayed until any previous releases of the same task have completed. For each potentially overlapping release we define a separate window W(q), where q is just an integer identifying a particular window (i.e. q =0,1,2, . . .). Hence, we get the following formula on W(q):

$$W_i^{n+1}(q) = (q+1)C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n(q)}{T_j} \right\rceil C_j \qquad (2.6)$$

Other practical issues such as inconstant computation time, multi-deadline tasks, internal deadlines, offsets and phases execution, one can refer to Alan Burns research in 1994[18].

2.5 Other Important Issues in Real-time System Design

In this section, we will consider more issues that are also important for the real-time system design other than the scheduling analysis. First, some remarks on the hardware architecture issues will be brought out; and then the programming language concern in the real-time system development will be discussed; the last part of this section will be the analysis on how to estimate the worst-case execution time in a high-level programming language[9].

2.5.1 Hardware Architectures in Real-time System

Within a common computer system, the hardware architecture is usually optimized for a high utilization in the average case. But, this conventional situation just contradicts with the criterion of the real-time system scheduling, which mainly puts more emphasis on the worst-case execution time analysis. The optimization techniques used in the conventional hardware architecture, which can influence the real-time system design, include parallel processing (pipelining), caching and direct memory access (DMA) etc. With such technologies used in the underlying hardware platform, the temporal behavior of the upper system will be very hard to predict or can only be estimated over-pessimistically[9].

Pipelining Impact

To improve the throughput of a processor, many modern computer architectures adopt the pipelining technology, which pipes the machine instructions to achieve executing them on processors in parallel. Using the pipelining technique can greatly improve the processor average performance. However, for the well-known breaking problem within pipelining technology caused by branch instructions, the worst-case execution time of such a hardware platform is very pessimistic. To approach more exact estimation of the worst-case execution time in a pipelined hardware platform, special analysis tools containing the detailed knowledge of underlying hardware architecture must be provided. However, such tools can hardly be found for most of the industrial products.

Cache Impact

Fetching data from the off-processor sources normally represents the performance bottleneck in the hardware architecture. Therefore, the cache technology was introduced to reduce such operations. However, to what degree caching can improve the overall performance during a specific period of time largely depends on the cache-hit percentage of that period. So, to estimate the worst-case execution time more accurately, a complex analyzing tool, which is able to emulate the cache behavior, is required.

Direct Memory Access Impact

Direct Memory Access (DMA) technology was invented to release the central processor from the burden of handling mass memory manipulations. But in case that DMA controller can operate in a cycle stealing mode or a burst mode, it induces much unpredictability into the system. In practice, such impacts must be considered and put into estimation for the worst case execution time.

All the above discussions indicate that the technologies used in modern hardware architecture bring many difficulties into the real-time system design domain. They make the system's temporal behavior very hard or even impossible to predict. Therefore, in order to successfully design and implement real-time systems, especially the hard real-time ones, choosing and analyzing the hardware architecture of the underlying platform are absolutely necessary.

2.5.2 Programming Language Issues in Real-time System Design

Normally, a high level programming language used in the real-time system development domain should fulfill the requirements below (as discussed in [9]):

- It should provide deterministic temporal behavior of programs.
- It should support strong type checking, exception handling and thus provide a secure and reliable runtime environment for the programs.
- It should support multiprogramming and provide effective mechanism for task synchronization.
- It should support good hardware access; especially, the ability to access the peripheral device drivers is essential to the real-time system implementation.
- It should support large-scale real-time application development.
- The programs written in it should be convenient to maintain.

The high level programming languages that are currently used in practice include implementation languages such as C/C++ and real-time languages such as ADA and PEARL.

The Java language fulfills most of the above requirements except for the first one and fourth one. To cater for the first requirement, many unpredictable features of Java, as mentioned in Chapter 1, must be addressed; and for the fourth requirement, JNI seems to be a way of solving this problem, but how well JNI is integrated in a specific real-time Java solution will still take effect on the Java language's ability of accessing low level hardware devices.

2.5.3 Worst-case Execution Time Analysis for High Level languages

As we mentioned earlier in this chapter, the scheduling analysis needs to know the execution times of the programs a priori. Such a requirement demands the real-time developers to deploy a worst-case execution time analysis for their programs in early stages of the development cycle.

There are some certain rules to analyze the worst-case execution time for a high level language[9].

First, the source code of the high level language should be parsed into basic temporal units such as straight-line code blocks, condition sentence, loops and method invocation and return etc. The temporal behaviors of these basic units should be analyzed and estimated. If possible, the assembly language or machine code representation of these basic units should be calculated to get more accurate results.

Second, each of the condition blocks and loop blocks are calculated in such a way: a condition block's worst-case execution time is equal to the longer branch; a loop block's worst-case execution time is equal to the execution time of the loop body multiplied by the maximal number of iterations.

At last, the nested structures in the compiled source code should be searched recursively until the innermost basic unit level is reached. Thus, the execution time of all the function blocks including the outermost one is then obtained.

In case of Java, for the first phase, some special issues should be paid more attention to, such as memory allocation, dynamic loading etc. As for the third phase, more concern about thread manipulation should be considered, such as thread startup time, thread context-switch time and so on.

Chapter 3 Overview of Automotive System Technologies

A vast increase in automotive electronic systems, coupled with related demands on power and design, has created an array of new engineering opportunities and challenges

-Gabriel Leen & Donal Heffernan

As being investigated and analyzed by Gabriel Leen and Donal Heffernan[19], there is a rapid growth of electronic systems inside the vehicle systems, which replace more and more traditional in-car components from mechanical units into electrical ones. In current BMW 7 series, there are more than sixty Electrical Control Units (often abbreviated as ECU), which are connected by five different types of buses (shown in Figure 3-1).

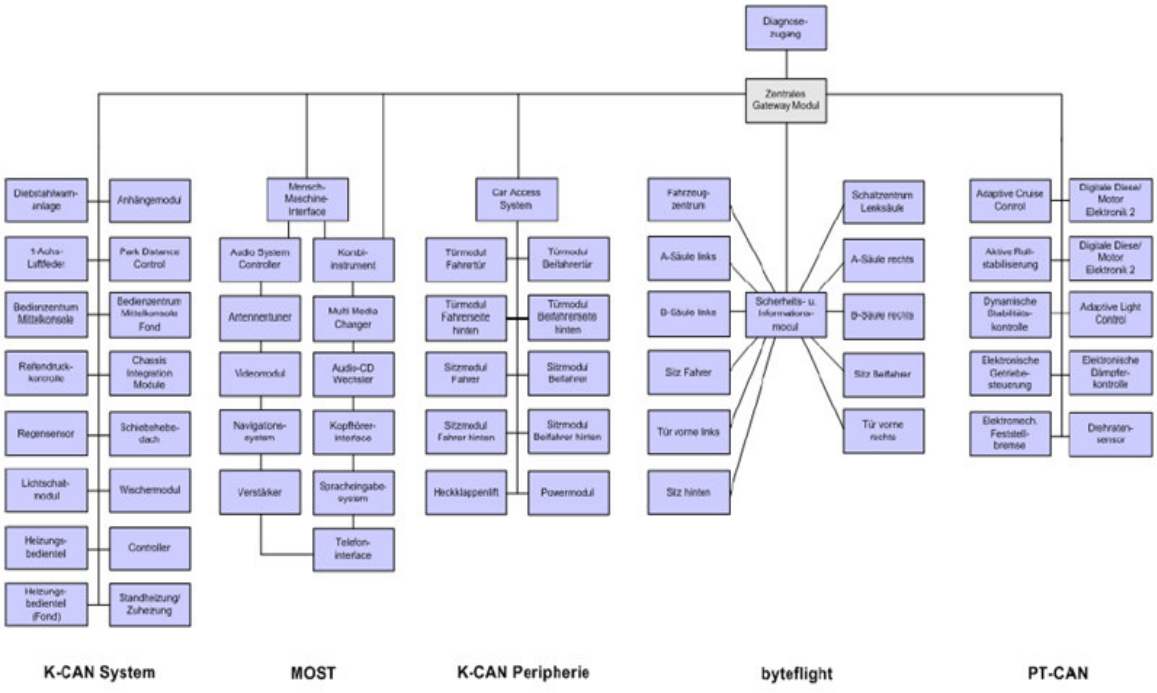


Figure 3-1 Current BMW 7 series on-board supply system structure. Boxes are ECUs[20]

Even for some critical units like anti-lock brake system (ABS), engine management system, are now being created using modern electronic, computer and communication technologies. On the other hand, such rapidly growing demands also make the software design tasks much more complicated than before. One of the challenges is to build more and more sophisticated hard real-time (sometimes safety-critical) software in the vehicle.

In section 3.1, the typical in-vehicle network architecture will be briefly introduced.

In section 3.2, we will present a typical real-time operating system (RTOS), OSEK, which is widely used on the powerful ECUs carrying real-time tasks. Both the advantages and the shortcomings of this RTOS will be discussed.

In section 3.3, the Control Area Network (CAN) bus system will be inspected, because the CAN bus has been widely used to build the in-car real-time communication network, and it is also chosen to be the underlying bus system for the sample automotive application developed in this thesis.

In the last section of this chapter, some of the latest technologies and concepts, which may be brought into the future automotive systems will be briefly prospected.

3.1 Typical In-Vehicle Network Architecture

The typical architecture of the ECU network inside the vehicle can be shown in Figure 3-2, an amount of distributed ECU being connected by several specific types of buses. First, the processor speeds of the ECUs vary a lot depending on their functionalities, ranging from several kilohertz up to several hundred million Hertz. The bus systems are also chosen based on the functionality requirements and the various bus properties, such as the bandwidths or the real-time features etc.

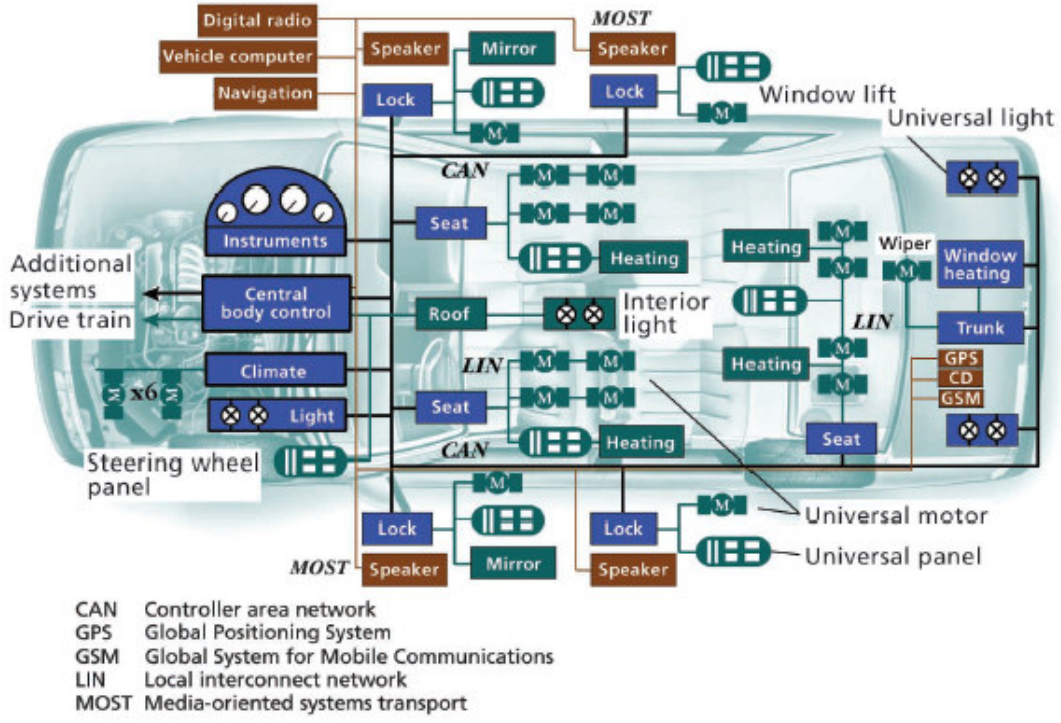


Figure 3-2: Typical Electrical Control Unit network architecture[19]

In such a distributed in-vehicle network, the gateway ECU, which acts as a central ECU that connects different buses, normally takes on more important tasks.

The typical tasks that are carried by the gateway ECU contains:

- Collecting data received from buses, which are detected by the sensor and sent by the end node ECU;
- Inspecting the status of each end ECU on the bus; make the appropriate reaction according to their changes;
- Exchanging data between two bus systems, sometimes even more than two bus systems are connected to the gateway ECU and take part in the message exchanges.
- Sending messages to the end ECU if necessary, in order to adjust the behaviors of some in-car components.

Therefore, the gateway ECU plays an essential role in the automotive system. Both of the hardware platform and the software architecture should be carefully chosen.

3.2 Typical Real-time Operating System in Automotive system: OSEK

As mentioned in the previous section, some ECU in the vehicle takes more complicated tasks and therefore has more powerful hardware support. To ease the software development on such ECUs, especially some hard real-time tasks, a Real-Time Operating System (RTOS) normally will be installed to provide more convenient development interfaces and real-time support. One typical RTOS product, which is widely used in the market today, is OSEK [21] (an abbreviation of a German term with the meaning “Open Systems and the Corresponding Interfaces for Automotive Electronics”).

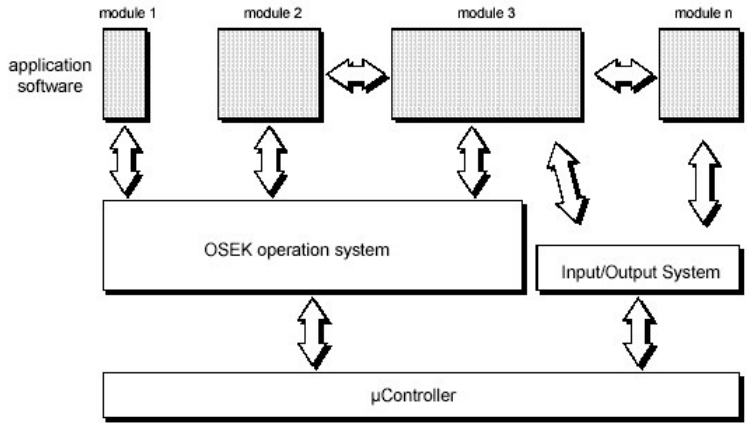


Figure 3-3: OSEK OS Overview[21]

Two specifications and one language compose the OSEK system, where OSEK OS is the operating system specification; OSEK COM is the communication specification, and the OSEK Implementation Language (OIL) is a modeling language for describing the configuration of an OSEK application and operating system. Both of OSEK OS and OSEK COM provide application program interface (API) standards for automotive real-time application development.

An overview of architecture of OSEK OS can be shown in Figure 3-3:

An OSEK COM's layer model is presented in Figure 3-4 to show conceptual model of OSEK COM and its positioning within the OSEK architecture.

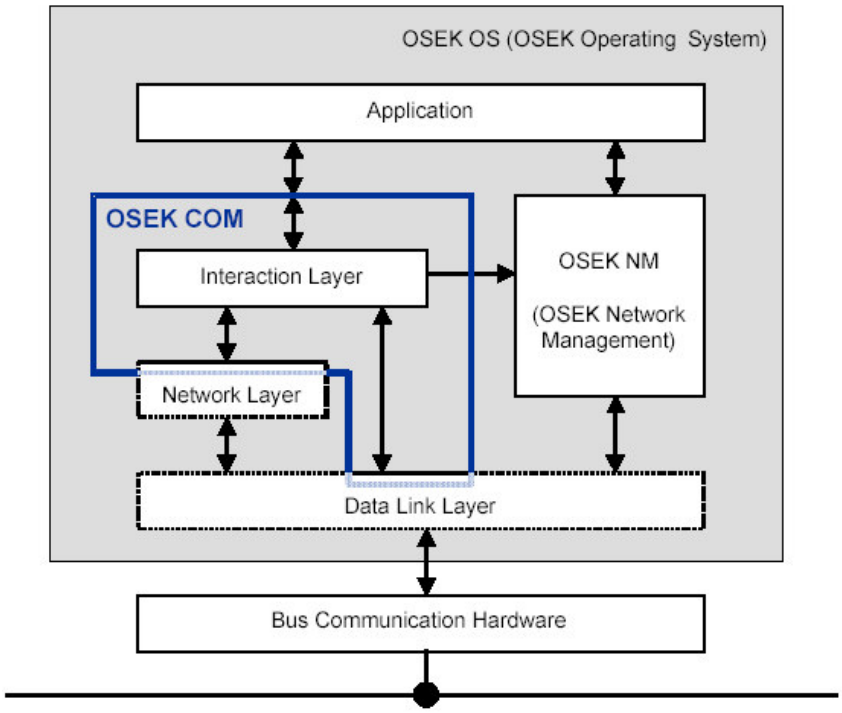


Figure 3-4 OSEK COM's layer model[21]

In an OSEK OS, eight kinds of services are provided, they are:

1) Task management and Scheduler: controlling tasks such as activating or terminating a task; providing scheduler for the tasks. The scheduler in OSEK is priority based and event-driven. The preemptability of the scheduler is configurable.
2) ISR (Interrupt Service Routine) management: providing the possibilities to use service calls in ISRs. ISRs always have higher priorities than tasks in the application.
3) Resource management: using priority ceiling protocol to synchronize system resources, which are shared among tasks.
4) Counters: keeping the number of the ticks elapsed, providing the time basis for the alarm.
5) Alarms: controlling tasks according to the timer (triggered by counters). Thus, counters and alarms together provide the time dependent services, whereas the rest of the services are all event-driven.
6) Events: serving as a task signaling mechanism; serving for tasks suspending and waking during the synchronization

7) Communication: providing services including sending and receiving messages between tasks. Messages can be queued or not queued (This part of the services needs to refer to the CPU internal communication in the COM standard)

8) Error Handling and hook routines: providing a mechanism with system callback routines to handle the errors that might occur in a program; providing routines for initialization or debugging.

The main features of the OSEK system can be summarized as follows:

- OSEK is a static operating system which has no dynamic handling of system resources
- Generates kernel individually for every single application
- Does not have dynamic memory management nor any kind of IO services
- Configuration is part of the application development
- Provides several strategies to set up a real time environment, such as
    - Prioritized Tasks;
    - ISR based event mechanism and messaging service
    - Counters and alarms for timing
    - Mutual exclusion guaranteed resource management…etc

OSEK is a highly concise real-time operating system with limited services provided. All the development work should be written in C so that some low level delicate tasks could be done. So the developers still have to face the hardware driver, task switch logics and a lot of memory management work. All these works are very low-level and error-prone, and will consequently decrease the productivity of the whole development process.

Nowadays, many new concepts and technologies, such as OSGi framework [22], have emerged to help automotive system developer's build more complex applications. Such tendencies has brought prospects as well as challenges into this area, that the in-car ECU need to have a platform with support of at least one advanced and powerful OO language, and support of distributed application development in order to build more complicated automotive applications with more efficiency.

3.3 Control Area Network (CAN) in Automotive Systems

CAN is a serial communication protocol which supports distributed real-time control systems and provides a high level of security[23].

It has the following distinguished properties, which make CAN an ideal candidate for building in-car real-time communication systems.

- Prioritization of messages
- Guarantee of latency times
- Configuration flexibility

- Multicast reception with time synchronization
- System wide data consistency
- Multi-master
- Error detection and signaling
- Automatic retransmission of corrupted messages as soon as the bus is idle again
- Distinction between temporary errors and permanent failures of nodes and autonomous switching off of defect nodes

The layered structure of one CAN node is shown in Figure 3-5.

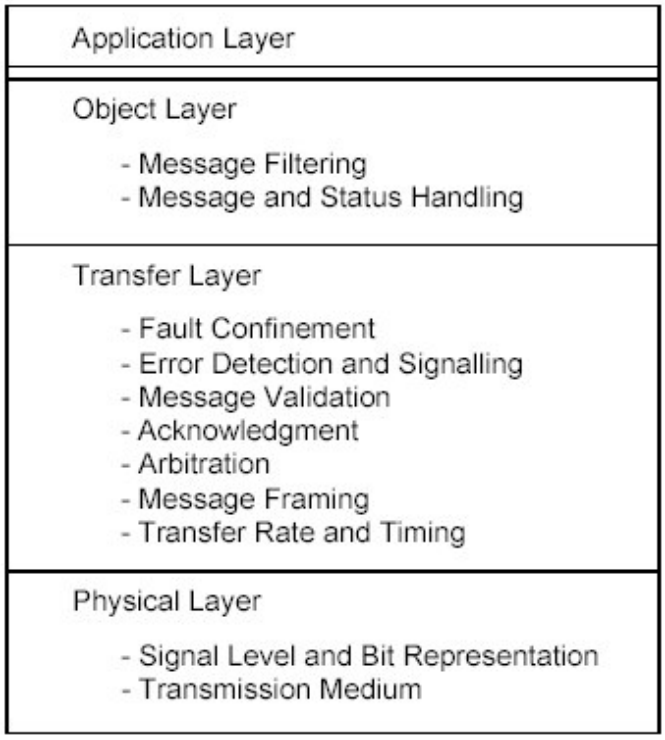


Figure 3-5 Layered structure in a CAN node[23]

3.3.1 Data Frame of CAN

The data frame of CAN is composed of seven different fields as shown in Figure 3-6.

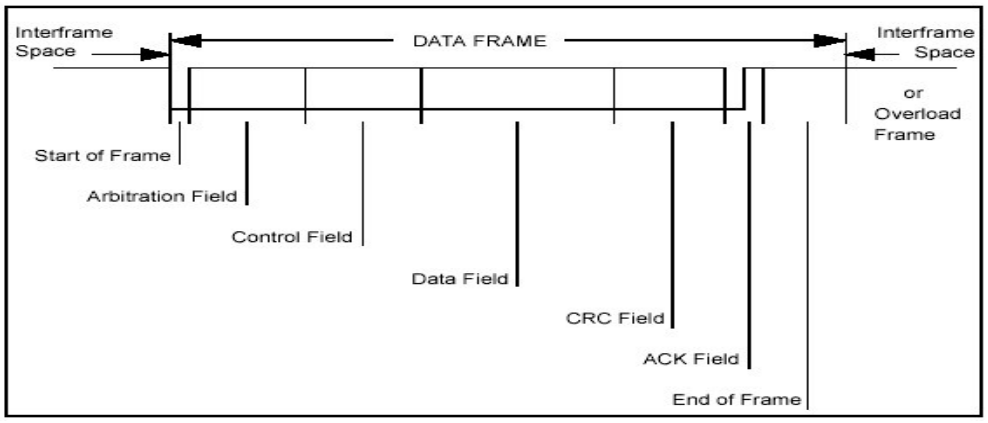


Figure 3-6 CAN Data frame[23]

The most significant field in the data frame during message routing is the Arbitration field. The value of the Arbitration field serves two purposes:

1. Describes the meaning of the data as an identifier

2. Controls bus arbitration

The first part of the Arbitration field is the Identifier of the CAN frame. This ID, instead of containing some destination address, contains a code identifying the meaning of the data. During the sending, the CAN station will send the frame ID to the bus in a multicast manner. All the stations, which desire to know the content identified by this ID, will set filters in their bus interface chips to match the code of this ID and whenever this ID is available on the bus, they will receive it and notify the processing element of the station to handle it. While, for the other stations that are not interested in this message, they simply filter out the ID and ignore this message when it appears on bus.

For the rest of the fields, 'Start of Frame' is used to synchronize all the stations when one of the stations starts transmission first; The control field specifies the number of bytes in the data field; The CRC field contains a 15 bit CRC check, and the ack field is used to acknowledge correct reception of a message.

3.3.2 CAN Bus Arbitration Mechanism

One feature of CAN bus that must be mentioned is its bus arbitration mechanism. This mechanism successfully handles the collision situation on the bus. Once there is a collision, it does not avoid collisions, instead, it uses the collision to compare the frame identifiers and leave the frame with the highest priority on the bus.

CAN makes use of a wired-OR (or wired-AND) bus to connect all the stations. Two values of a bit represent whether it is *dominant* or *recessive*.

When the bus is free, any stations on the bus may start to transmit a message. If two or more stations start transmitting messages at the same time, the bus access conflict is resolved by bitwise arbitration using the ID field of the data frame. During arbitration every transmitter compares the level of the bit

transmitted with the level that is monitored on the bus. If these levels are equal the unit may continue to send. When a 'recessive' level is sent and a 'dominant' level is monitored (see Bus Values), the unit has lost arbitration and must withdraw without sending one more bit. Under such mechanisms, the frame with the highest priority ID can always win the arbitration and be sent onto the bus.

3.3.3 Real-time Concerns about CAN bus

As mentioned earlier in this section, CAN has several built-in features, which make it very suitable to build the in-car real-time network. However, to be a 'real' hard real-time communication protocol, which can provide timing guarantee for the messages on it, CAN is still not the perfect one for its event-triggered bus concept lying behind, which lacks enough predictability. For CAN a maximum busload of 50% is often recommended for non-critical applications [24]. And for real-time critical applications on CAN bus, a maximum busload of approximately 20 - 30% is suggested in [25]. To achieve more predictability in order to help the work of hard real-time scheduling, the protocol built on a time triggered concept is preferred, such as the Time Triggered CAN (TTCAN). Furthermore, to keep both the flexibility and simplicity of the events triggered bus and the high predictability of the time triggered bus, new communication systems with hybrid architecture have been deduced from the specification of future bus protocols, such as FlexRay[26]

3.4 Future Prospects for Automotive Technologies

In recent years, many new technologies and concepts have emerged to cater for the rapidly increasing demand in the automotive software development domain, such as dynamic real-time operating system technologies and OSGi framework.

3.4.1 Dynamic Real-time Operating Systems

Since static real-time operating systems like OSEK cannot satisfy the fast growing demand of building large, complex real-time applications, several dynamic RTOS products come into being in recent years, such as VxWorks and QNX Neutrino. For the reason that this thesis adopts QNX as the operating system in the test-bed environment, we will take it as an example to illustrate the mainstream technologies and concepts involved in the dynamic real-time operating systems today.

QNX RTOS

The real-time operating system developed by QNX Software Systems represents a new concept of the next-generation real-time operating systems. It delivers the open systems POSIX API in a robust, scalable form suitable for a wide range of systems -- from tiny, resource-constrained embedded systems to high-end distributed computing environments[27].

The main features of QNX RTOS include:

- Scalable system size

QNX RTOS offers the customer the option to scale a microkernel OS simply by including or omitting the particular processes that provide the functionality required. In such a way, the customer can use a

single microkernel OS for a very wide range of applications. Rather than changing operating systems products, customers can easily scale the needed system upon a microkernel OS by adding additional components such as file systems, networking, graphical user interfaces, and so on.

- POSIX implementation

POSIX is the acronym for Portable Operating System Interface. It is a proposed operating system interface standard aiming to support application portability at the source-code level. Some of its standards such as POSIX 1003.1, POSIX Real-time Extensions and POSIX Thread Standard can greatly contribute to the development in the real-time embedded system domain. So, implementing the POSIX standard API could be a rather important criterion when choosing a real-time operating system in the future.

- Highly concise microkernel architecture

The scalable feature of QNX mentioned above is mainly contributed by the concise microkernel architecture. As shown in Figure 3-7, in the QNX RTOS, most of the conventional in-kernel components, such as file system, process manager etc, are taken out of the kernel to make a highly concise and flexible approach. The microkernel of QNX only provides a few fundamental services, such as thread services, signal services, message passing services, synchronization services, timer services etc.

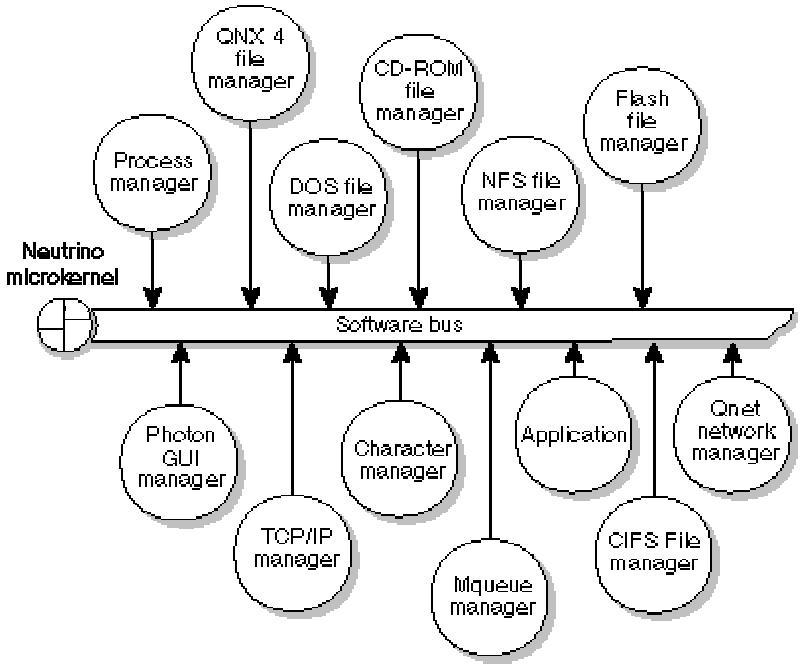


Figure 3-7 QNX RTOS microkernel and system architecture [27]

- Message-based inter-process communication

Real-time and other mission-critical applications generally require a dependable form of IPC, because the processes that make up such applications are so strongly interrelated. The QNX RTOS's message-passing strategy can help bring more reliability and predictability to the applications.

- Support of cross development

To help the development work on the QNX RTOS, a cross-hosted development environment is also provided by the QNX group. Such convenient tools will greatly improve the efficiency of the real-time development work.

3.4.2 OSGi

The Open Services Gateway Initiative (OSGi) [22] was founded in March 1999 for creating open specifications for the network delivery of managed services to local networks and devices. With more than eighty member companies today, OSGi has become a leading standard for the next-generation Internet services to homes, small offices and in-vehicle systems.

The OSGi specifications provide an open standard for remote programmable devices. The scopes of the specifications include software downloading, application life cycle management, programming environment security, device driver management, configuration management, user management and a remote administration model. The OSGi specification contains Java APIs and a clear and concise definition of their semantics.

The technical work required to generate OSGi Specifications is conducted within approved expert groups. One of the three OSGi expert groups, which deal with OSGi architecture, is Vehicle Export Group(VEG). "The Vehicle Expert Group is working on tailoring and extending the generic OSGi Service Platform core specifications for use in in-vehicle environments. The VEG receives much of the input from automotive, telematics and transport member companies to ensure the specifications produced are well suited to their target environment." [28]. This shows the close connection between the OSGi and modern automotive technologies.

The following entities are involved in the OSGi solution:

- Service Platform

An application server that is connected to both the local networks and the wide area network (sometimes, Internet)

- Service Provider

The provider of the specific value-added services, which are deployed on the Service Platform

- Gateway Operator

It offers the Service Provider a safe environment to execute and is responsible for the integrity of this environment. It fully controls the Service Platform through the management application such as installing or removing applications etc. In an automotive OSGi implementation in vehicle, the Gateway ECU often acts as the Gateway Operator.

- Internet Service Provider

To provide the Internet access

- Local networks and devices

They are connected to the Service Platform by local wired or wireless networks so that all the services in there can be accessed. In such a way, the resource limited embedded devices can be connected to the wide area network.

Common stack architecture in an OSGi implementation (in the gateway) can be shown in Figure 3-8.

Some OSGi terminologies are introduced here:

- Bundle

The OSGi service implementation and deployment package. Normally, a bundle is implemented as a JAR file, which contains service classes, Java libraries or native libraries etc.

- Framework

The actual service container, or to say, bundle container, which provides environment for the deployed services and manages them.

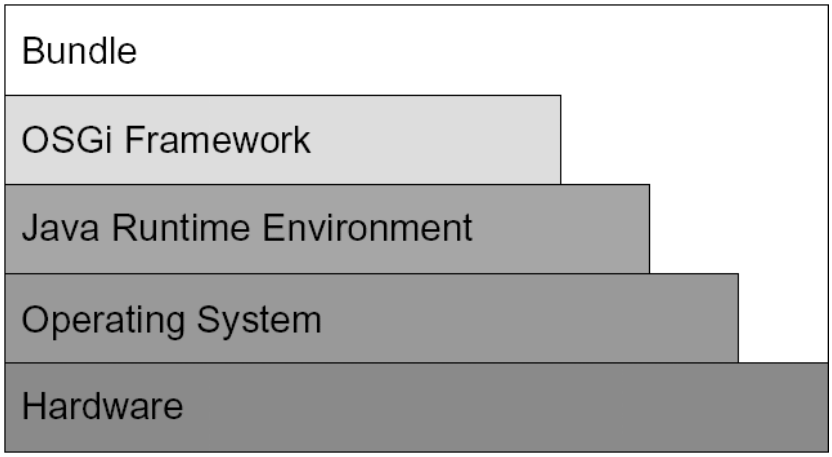


Figure 3-8 Common OSGi implementation architecture

Although, for the time being, the OSGi specification is merely applied in the non-real-time application development domains, such as in-car multimedia services, vehicle diagnoses service etc. It is evident that there will be a strong demand to integrate the automotive applications and networks, both non-real-time and real-time, in order to provide more sophisticated and convenient in-car services. Therefore, the expansion of OSGi technologies used in automotive systems raises big challenges as well as great prospects for the real-time Java technologies.

Chapter 4 Different Real-time Java Techniques Comparison

To make a Java program run in a predictable way, there are many committees and companies being devoted to it. These entities have made a lot of efforts and raised several different approaches to improve the real-time behavior of Java. All these solutions can be roughly classified into three categories:

- Hardware support approaches

  For example: aJile System [7]

- Language extension specifications and their implementations

  For example: Real-time Specification of Java[3], Real-time Core Extensions[4]

- Other software solutions with their own architectures and implementations

  For example: the PERC Java Virtual Machine[6], the Jamaica Virtual Machine[5]

Considering the goals of this thesis and the limited time of achieving them, this thesis cannot traverse all these solutions and their technical details. So, due to the determined target hardware platform (Transmeta processor) and the fixed operating system (QNX) in this thesis[1], we only choose the compatible ones among all real-time Java solutions as the candidate solutions and deploy the evaluation on them. This part also shows the difficulty of importing a certain real-time Java solution into a specific industry domain. The compatibility with the hardware platform, the operating system or even some specific hardware driver, all these issues must be taken into consideration to choose an appropriate solution.

After considering all the above issues plus the hardware requirement and product maturity concern, two real-time Java products and one conventional embedded Java product are chosen primarily in this thesis. They are PERC Java Virtual Machine from NewMonics (real-time), Jamaica Virtual Machine from Aicas (real-time) and J9 Virtual Machine from IBM (Non-real-time).

However, as more and more experiments done to these three Java virtual machine products, we find that the Jamaica virtual machine tends to be more instable on the test-bed environment. It fails on supporting several important experiments of this thesis. Hence, we can only remove it from the listed target platforms we will test on. But still, the special methodology and theory existing in Jamaica are worth to study. So, for the test deployment and experimental evaluation of this thesis, the PERC virtual machine and IBM J9 will be the final two platforms to test on. As for the theoretical study on different real-time Java technologies, we still give a brief introduction to Jamaica virtual machine and hope that it could be more mature and convenient in the future to fulfill the requirement of complex real-time application development in automotive systems.

Both the PERC virtual machine and the Jamaica virtual machine belong to the third category, which is the individual software solution. In this chapter, the main features and technologies of these two

[1] For more information about the evaluation test-bed in this thesis, the reader can refer to Chapter 7.

approaches are introduced respectively. In addition, since the Real-time Specification of Java plays a very important role in the real-time Java domain, though its implementations cannot be evaluated in this thesis because of their incompatibility with the target hardware platform and operating system, we will still bring an overview on its main features in order to set up a reference solution to compare with the later two approaches.

Since the garbage collection mechanism brings so many impacts on the real-time performance of Java, some background knowledge on garbage collection will be briefly introduced right after the foreword, and then in each of the following sections on the different real-time Java solutions technique description part, we recalled the garbage collector issue and compare the different strategies for memory allocation and garbage collection. Some other important issues, such as the strategy for thread synchronization, will also be mentioned for each real-time Java solution and get compared so that in such a horizontal way a theoretical evaluation can be accomplished more clearly. In the last section of this chapter, we will make a discussion on the code reuse issues raised in practice.

4.1 Garbage Collection

As mentioned in the first chapter, automatic garbage collection mechanism of Java raised a great challenge to the real-time Java implementers. To provide a real-time feature, a specific solution must choose a way to conquer the unpredictability caused by garbage collector. In order to examine and evaluate different solutions on garbage collection better, some preliminary knowledge about the garbage collector is necessary and should be observed first.

4.1.1 Garbage Collector Technique Overview

A traditional garbage collector is well known for its long running time and unpredictability of invocation. Typically a garbage collector is realized as an independent thread with a sufficiently high priority, which cannot be preempted by the other user threads. Thus, once it is invoked, it will not stop until all the collection work in this cycle finishes. Generally, one cycle of GC work contains the following four phases as follows (shown in Figure 4-1):

Root Scanning

Check through the non-heap area i.e. all the registers and the stack, to find all root references and the objects they are referring to.

Mark

Mark all the reachable objects in the heap recursively, these objects will be regarded as alive and kept staying continuously in the memory

Sweep

In this phase, all the unmarked dead objects will be swept and reclaimed.

Compact

In order to prevent the heap area from being fragmentized and unable to allocate any large object, in this phase, the garbage collector will move the living objects together and free more large blocks of memory for further allocation

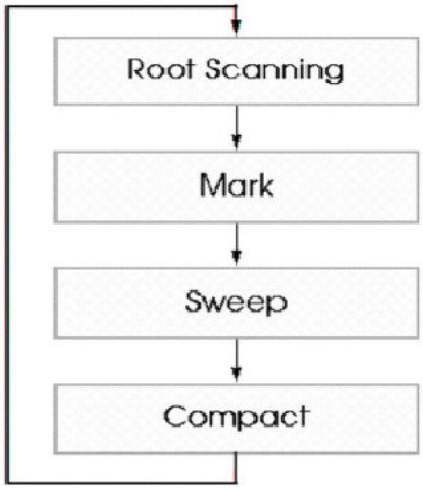


Figure 4-1 Typical GC phases[29]

Apparently, among the four phases, the work in 'Compact' tends to be the most dangerous and time-consuming phase and may cause much unpredictability. In a typical Java virtual machine this part of the work will not be interrupted until finishing in order to keep the memory space consistent. A real-time garbage collector must make the time period of this part of the work bounded and predictable, while at the same time, reclaiming enough memory for applications to use. This task becomes the trickiest problem that the real time Java implementers, who provide a real-time garbage collector, have to face.

Now, let us look at the real time Java solutions one by one.

4.2 Real Time Specification of Java

The Real-Time Specification of Java (RTSJ) was carried out by the Real-time Java Experts Group (RTJEG) in the year 2000. It aims to make a common standard for the real-time Java implementers to follow. It addressed several issues that determine the real-time performance of Java and correspondingly provided strategies. Generally, there are seven main areas covered by RTSJ, which construct the seven main features of it.

4.2.1 Main Features in RTSJ

These features include[3]:

(Existing features in Java that are not deterministic)

- Thread Scheduling and Dispatching
- Memory Management
- Synchronization and Resource Sharing

(Additional features that should be provided to support the real-time software development)

- Asynchronous Event Handling
- Asynchronous Transfer of Control
- Asynchronous Thread Termination
- Physical Memory Access

These features are generally guided by a set of core requirements for real-time Java concluded on a workshop, which was sponsored by the National Institute of Technology (NIST) during 1998 to 1999. The final workshop report, published in September 1999[30], defines nine core requirements as follows:

1. The specification must include a framework for the lookup and discovery of available profiles.
2. Any garbage collection that is provided shall have bounded preemption latency.
3. The specification must define the relationships among real-time Java threads at the same level of detail as is currently available in existing standards documents.
4. The specification must include APIs to allow communication and synchronization between Java and non-Java tasks.
5. The specification must include handling of both internal and external asynchronous events.
6. The specification must include some form of asynchronous thread termination.
7. The core must provide mechanisms for enforcing mutual exclusion without blocking.
8. The specification must provide a mechanism to allow code to query whether it is running under a real-time Java thread or a non-real-time Java thread.
9. The specification must define the relationships that exist between real-time Java and non-real-time Java threads.

Table 4-1 can show how RTSJ satisfies all these requirements except for the first one, which is beyond RTSJ's primary scope. Additionally, the feature of 'Access to physical memory' provided by RTSJ is not motivated by the requirements but it is added into RTSJ to cater for some practical demands existing in industrial real-time software development.

In RTSJ, the authors decided to support real-time programming in Java by defining features and semantics with extended APIs. In such a way, RTSJ can lead the programmers managing thread execution and reducing the overall unpredictability of execution for certain thread types. The extension upon standard Java API explicitly causes some impacts on the real-time development such as code reuse problems or flexibility issues. Whether it is necessary to impose such an API extension

and how to tradeoff between keeping standard API and extending standard API will be discussed at the end of this chapter, after both of the candidate real-time Java solutions have been introduced.

RTSJ Features	NIST core requirements								
	1	2	3	4	5	6	7	8	9
Scheduling	N/A		S					S	
Memory Management	N/A	S	S						
Synchronization	N/A		S				S		S
Asynchronous event handling	N/A			S	S				
Asynchronous transfer of control	N/A								
Asynchronous thread termination	N/A		S			S			
Physical memory access	N/A								

Table 4-1 RTSJ features with NIST core requirements[31]

We now briefly look into each of RTSJ's features and explain their usages as well as the help they brought to the real-time development.

4.2.2 Scheduling

RTSJ requires its implementation to have at least 28 unique priorities in the system and also provide a fixed-priority preemptive thread dispatcher. This shows that the scheduling method in RTSJ has been confined to the fixed-priority preemptive one, which we have discussed in detail in Chapter 2. The reason why RTSJ chooses 28 as the least number of unique priorities in the system is that, according to some earlier research done by L. Sha et al[32], system with 32 unique priorities one can expect close to the optimal schedulability, and the RTSJ implementation, as part of the underlying system, should better leave some priorities for the system usage.

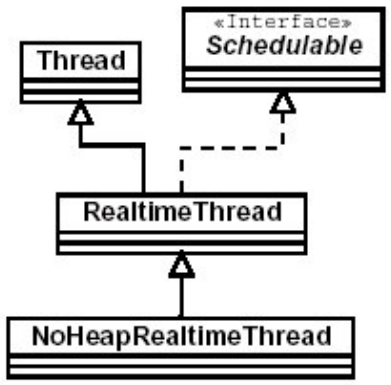


Figure 4-2 RTSJ Real-time Thread class Hierarchy[38]

The scheduling part of RTSJ also defines a set of APIs for implementers to follow. Three classes: *Scheduler*, *SchedulingParameters* and *ReleaseParameters* should be provided to control the schedulable threads or event handlers. For Java threads that have temporal constraints, RTSJ defines two types of real-time thread: *RealtimeThread* and *NoHeapRealtimeThread*. The difference between these two threads is that the prior one can access the objects on the heap but the latter one cannot. Since RTSJ does not require a real-time garbage collector implemented, the difference indicates that one *RealtimeThread*'s temporal behavior could be interfered by the garbage collection work, while the *NoHeapRealtimeThread* can run even when the garbage collection is running in the background. The two types of real-time threads therefore provide alternatives for the developers to choose between the real-time tasks with loose time constraints and the tasks with stringent deadlines. In addition, the regular threads in RTSJ do not have any guarantees for their temporal behaviors. The thread class architecture of RTSJ is shown in Figure 4-2.

4.2.3 Memory Management

Facing the unpredictable garbage collector problem, RTSJ chooses a way to keep the traditional garbage collector implementation and create other memory areas rather than the heap for the real time memory allocation. Thus, it becomes possible for the real time thread to preempt the garbage collector thread because there is no memory interference between the two threads. Figure 4-3 and Figure 4-4 show the different threads running states between the traditional Java virtual machine and the virtual machine under RTSJ.

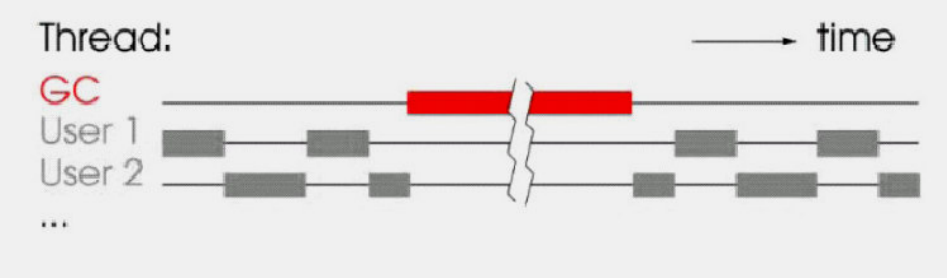


Figure 4-3: Garbage collector and threads in typical Java virtual machine[29]

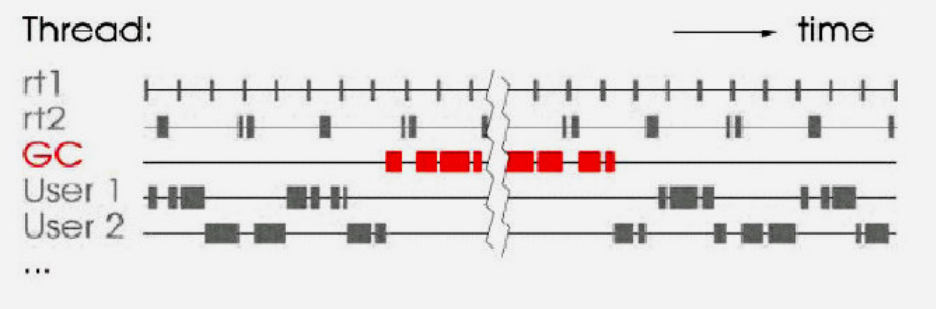


Figure 4-4: Garbage collector and threads running in RTSJ[29]

As we can see in Figure 4-4, the real time threads (*NoHeapRealtimeThread*) rt1 and rt2 can aways preempt the garbage collector and complete their real-time tasks in time. But the other user threads, which are using the common Java thread class, will still run in an unpredictable mode. This implies that, the existing codes with usage of standard Java library will not get the real time feature by just putting themselves onto the RTSJ defined virtual machine. The drawback here seems not trivial, since many companies will prefer a real-time Java solution that could allow them to reuse the existing codes to achieve real-time. However, as we have analyzed in the Chapter two, the real-time software development requires all the phases, especially scheduling analysis, design and implementation, to consider the temporal issues in every detailed part of the application. Without such careful design, the existing Java code may contain many inappropriate structures and workflows to make it very hard or even impossible to guarantee its predictability. Therefore, the solution from RTSJ forces the developer to fit the application into their structure and reconsider the temporal behavior in the application from the top down. This may be more appropriate than to just move the whole application to a new environment to run and pray for it.

Back to the memory management strategy of RTSJ, RTSJ introduces the notion of *memory area,* which denotes the region of memory that one can allocate for object creation. Besides the common heap memory area for Java, there are three new types of memory area defined in RTSJ, they are physical memory, immortal memory and scoped memory (shown in Figure 4-5). Physical memory is the memory area for developers to create particular important objects, such as objects to put into nonvolatile RAM. Immortal memory is a memory pool that all the objects allocated within it will live until the whole program terminates, never reclaimed. Immortal object allocation is suitable to be used in the hard real-time systems for more predictable memory behavior.

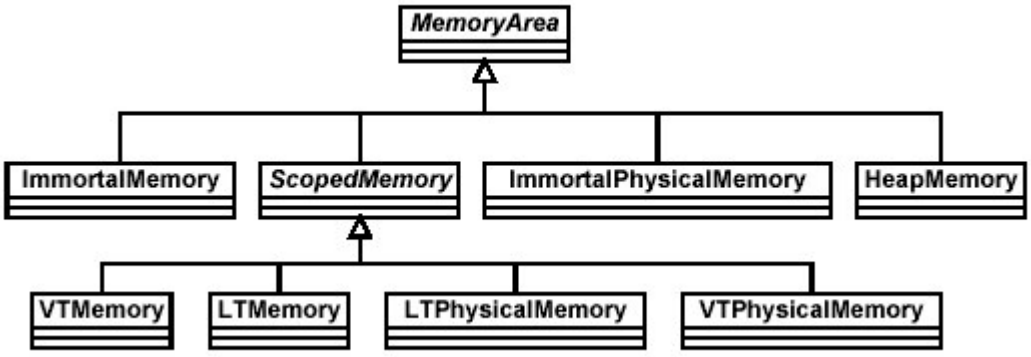


Figure 4-5 Hierarchy of classes in RTSJ memory model[38]

Scoped memory is an important concept brought by RTSJ. It lets the developer allocate and manage their objects in a memory area, where the lifetimes of objects allocated in it are bounded within a syntactic scope. When the system enters such syntactic scope, each *new* operation can allocate memory from the scoped memory for objects. When this syntactic scope ends up, the system will destroy all the objects allocated in it and reclaim their memory.

4.2.4 Thread Synchronization

For the synchronization and resource sharing aspect, RTSJ requires the priority inheritance strategy implemented as a must by default to avoid the priority inversion problem. The priority ceiling emulation policy is also specified by RTSJ for the implementations that want to support it. In addition, it also provides the possibility that allows the implementers to override the default synchronization policy to add more synchronization mechanisms.

RTSJ requires the implementations providing the fixed upper bound time for the program to enter a synchronized block with the monitor in an unlocked state.

RTSJ also defines the mechanism for the communication between threads, especially between the real-time threads and the non-real-time ones. Through a unidirectional queue, it can be ensured that the real-time thread will not be blocked by the non-real-time ones during the communication.

4.2.5 Asynchronous Event Handling

In RTSJ, the asynchronous event handling mechanism is carried out by two classes: *AsyncEvent* and *AsyncEventHandler*. The *AsyncEvent* represents any type of events, including the events occurred outside JVM such as hardware interrupt, and the events created inside JVM for some specific conditions reached. Whenever the *AsyncEvent* fires, the system will dispatch an *AsyncEventHandler* for it. An *AsyncEventHandler* is a thread-like schedulable object. When the event fires, the *run()* method of it will be invoked just like a thread. As for the implementation of such *AsyncEventHandler*, much fewer resources than for the actual thread can be realized. Thus, the system is able to handle a large amount up to tens of thousands of *AsyncEvents* with must fewer actual threads.

The Timer class defined in RTSJ is also a special kind of *AsyncEvent*. There are two forms of Timers: *OneShotTimer* and *PeriodicTimer*, which the former one only fires once at a specific time and the later one fires periodically with a specific interval. Each fire of these timers will be handled by an *AsyncEventHandler* and hence can be scheduled and guaranteed the temporal constraints.

By defining the asynchronous event handling mechanisms, RTSJ forms a convenient and effective way to schedule and handle both periodic and non-periodic tasks.

4.2.6 Asynchronous Control Transfer

The Asynchronous Control Transfer (ACT) technology is introduced into RTSJ to solve such practical problems: the computation time of an algorithm is highly variable. Given the time bound for the computation, there may be such circumstances that demand the control transferred from computation to the result transmission as soon as possible in order to catch the deadline. The ACT mechanism can then help in such cases. The main idea to approach ACT in RTSJ is to import an *AsynchronouslyInterruptedException*(AIE) into the thread, which requires ACT. The methods that allow ACT should explicitly declare to throw the AIE and when the control goes into such methods, an outside call of this thread's *interrupt()* method can immediately lead to an AIE thrown in that method and achieve fast ATC. If the *interrupt()* method is called when the control goes into a method that doesn't declare to throw AIE, the system will create an AIE and set it to a pending state. Till the

next time control enters method with the declaration, this AIE will be thrown. In addition, an AIE will also be set to a pending state when the control is in, returns to or enters a synchronized block.

4.2.7 Asynchronous Thread Termination

Rather than the current unsafe approach of terminating a thread, such as method *Thread.stop()* and *Thread.destoy()*, the asynchronous thread termination part of RTSJ aims at asynchronously terminating a thread safely and efficiently. By combining the asynchronous control transfer mechanism and the asynchronous event handling strategy, this goal can be easily and effectively achieved.

4.2.8 Physical Memory Access

RTSJ introduces the physical memory access technology into Java so that more direct and convenient memory operations can be conducted for special purpose. Two kinds of physical memory accesses can be done by the classes defined in RTSJ. One is through class *RawMemoryAccess*, you can construct an object to represent a range of physical addresses, and then access the physical memory with byte, short, int, long, float and double by using the *set<type>()* and *get<type>()* methods. The second way of accessing physical memory is to use *VTPhysicalMemory*, *LTPhysicalMemory* or *ImmortalPhysicalMemory*. These three classes allow programmers to create objects representing a range of physical memory addresses where Java objects can be allocated.

4.3 PERC

As an independent Real-time Java solution, PERC virtual machine tries to provide a real-time system development environment in a different way rather than RTSJ. First, it chooses to implement a real-time incremental garbage collector in the system to avoid the unpredictability caused by memory manipulation. It also provides a set of individual APIs and tools for the developer to check and control the behaviors of the virtual machine either outside the application or right in the program. Furthermore, PERC supports all the J2SE libraries except for the graphic interface packages (which means neither *awt* nor *swing* is available) and supports JNI technology as well as the direct memory access feature[33]. The following sections will look into their approaches in detail.

4.3.1 Real-time Garbage Collector in PERC

To achieve the real-time feature in Java, instead of creating different memory areas to evade interference of GC as defined in RTSJ, PERC tries to implement a predictable real-time garbage collector. Their real-time garbage collection solution is to use an incremental two-space copying strategy. Figure 4-6 and Figure 4-7 together describe a typical garbage collection scenario in PERC virtual machine.

As shown in Figure 4-6, at runtime, the memory space of the Java heap in PERC virtual machine will be divided into several regions with the same size (the number of regions can be configured before the virtual machine starts). Whenever the garbage collector runs, two of these regions will be selected

out to form the *to* and *from* spaces for the copying garbage collection. For the rest of the regions, the typical mark and sweep work will be conducted as described earlier this chapter. Note that, the compact work is not applied here mainly because the memory defragment task is carried out in the chosen two regions by the two-space copying strategy. In other words, for each pass of garbage collection, only two regions of memory can be defragmented and the whole heap memory can be kept utilizable by choosing different copying regions each time.

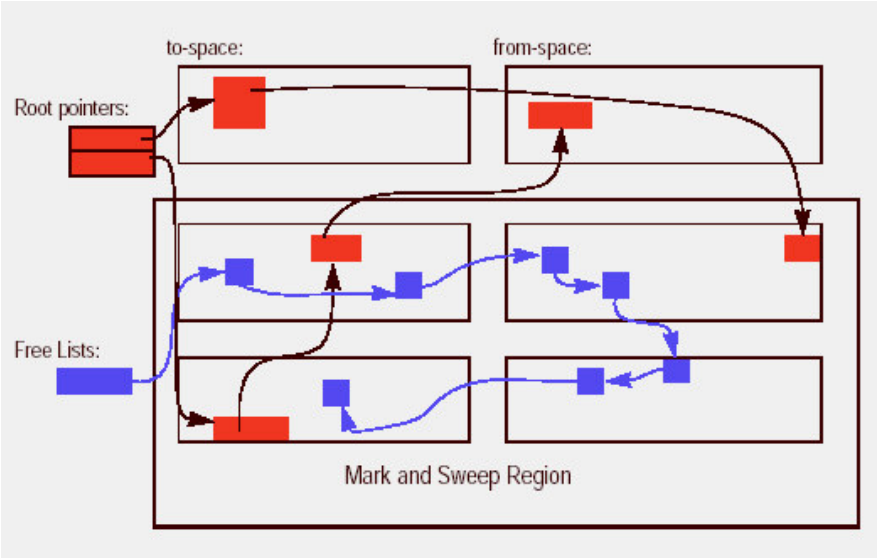


Figure 4-6 Java heap in PERC virtual machine[33]

The two-space copying strategy can be explained by Figure 4-7. After the mark phase of the garbage collection, all the in-use objects in the from-space, such as A, B and C in the figure, will be copied to the *to-space* one by one. Two benefits lie in this copying behavior: one is that after the copying, all the memory in *from-space* will be reclaimed so that a big free block is available in heap; the other benefit is that, compared to the common compact operation in the typical garbage collector, this copying approach always keeps the valid objects in the memory so that this work can be preemptive and resumed without destroying the integrity of the system.

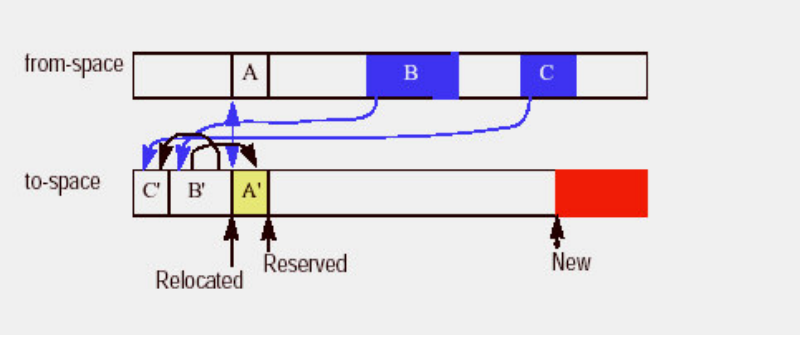


Figure 4-7: PERC GC – two-space copying strategy[33]

To summarize, the real-time garbage collector in the PERC virtual machine has the following features:

- **Preemptive**: As we have discussed, the two-space copying work can be preempted by other threads when necessary.
- **Incremental**: This means, the garbage collection work can be divided into many small incremental work and therefore can reduce the GC impact on the schedulability and predictability of the tasks.
- **Accurate**: the PERC real-time garbage collector uses an accurate way to conduct the GC work. (Here, accurate is in contrast to the conservative garbage collector which references objects less accurately in the mark phase)
- **Defragmenting**: This feature benefits from the two-space copying strategy.
- **Paced**: In the PERC real-time garbage collector approach, the incremental GC work can be paced slowly in some cases to save the CPU time for the stringent high priority real-time task. There is a pacing agent inside the virtual machine, which controls the GC pacing as well as performs enough GC work for later allocation. The behavior of the pacing agent can be configured before the runtime by setting some general temporal attributes of the real-time tasks. PERC API also provides the possibility of communicating with pacing agent at runtime. In addition, the pacing agent follows some basic principles of rate-monotonic theory to adjust the GC operations.

4.3.2 Virtual Machine Management API and Improved Timer Services

PERC virtual machine provides a set of API for the developers to control and manage the behaviors of the virtual machine as they requires[34]. The services in these VM management APIs include:

- Query and modify the maximum number of heap allocation regions
- Query and modify the priority and frequency of the real-time garbage collector
- Determine how much CPU time to be used for a particular Java thread
- Determine which thread should hold the lock of a particular synchronization monitor and which threads should be waiting for the particular synchronization monitors.
- Query the RTOS native priority of a particular PERC thread
- Query the duration of a PERC thread's tick period and time-slice duration.
- Query how much time the PERC virtual machine has been idle and how much CPU time has been consumed at each priority level of threads (to assist rate-monotonic scheduling analysis).

4.3.3 Other Features of PERC

Other features provided by PERC can be summarized as follows:

- For thread synchronization issues, PERC implements priority inheritance mechanism to avoid priority inversion problem.

- PERC virtual machine fully supports J2SE 1.3 libraries (except for GUI packages).
- Besides supporting JNI with real time guarantees, PERC also provides an alternative native interface: PNI (PERC native interface) and the API for accessing physical memory.
- To improve the runtime performance of the Java applications, PERC provides JIT(Just In Time) compiler and AOT(Ahead Of Time) compiler besides the basic interpreter.
- Several tools are provided for debugging and profiling the applications.

4.3.4 Remarks on PERC Real-time Java Solution

The PERC real-time Java approach mainly focuses on providing a convenient and efficient way of developing embedded Java program with real-time demands. It is suitable to be used in the soft real-time application development area (as they often call it a firm real-time). This is because some important strategies in their solution to help the real-time development are based on empirical or experimental knowledge, such as the GC pacing, runtime adjusting virtual machine parameters feature and so on. And the necessary theoretical scheduling analysis work in the hard real-time domain is not well supported and hence is difficult to be applied within PERC virtual machine. Furthermore, PERC is more suitable for building relatively large applications with a relatively powerful hardware environment. Thus their efforts on the J2SE libraries support can be well utilized.

Table 4-2 shows the philosophies from PERC technical group about the different real-time Java solutions' properties due to the different real-time requirement.

VM Properties	**Traditional Java virtual machine (Non-real-time)**	**Mission-Critical Java (real-time)**		
		Soft Real-time	**Hard Real-time**	**Safety Critical**
Library Support	J2SE	J2SE	Special Subset	More restrictive subset
Garbage Collection	Exhibits pauses in excess of 10 seconds	Real-time GC	No	No
Manual Memory Disposition	No	No	Yes	No
Stack Memory Allocation	No	No	Yes	Yes
Dynamic Class Loading	Yes	Yes	Yes	No
Thread Priorities	Unpredictable priority clusters and priority aging	Fixed preemptive, distinct	Fixed preemptive, distinct	Fixed preemptive, distinct

Priority Inversion Avoidance	None	Priority Inheritance	Priority Inheritance and Priority Ceiling	Priority Ceiling
Asynchronous Transfer of Control	No	Yes	Yes	No
Approximate Performance	*x*	0.9*x*	2-3*x*	2-3*x*
Typical Memory Footprint	16+Mbytes	16+Mbytes	64Kbytes-1Mbyte	64-128 Kbytes

Table 4-2 Differentiation between Real-time Java technology standards proposed by PERC producer[35]

Particularly, the PERC development teams claim that, they are now implementing the hard real-time Java solutions and will try to integrate them within the PERC virtual machine in the coming months.

4.4 Jamaica

Jamaica real-time Java solution is provided by Aicas Company. It aims to implement a hard real-time Java virtual machine by providing a hard real time garbage collector and at the same time guaranteeing that the execution time of all the primitive Java operations are bound and predictable [36]. Some of its main techniques to achieve real time Java can be summarized as below:

- A hard real time garbage collector

  Instead of keeping the garbage collector as they were like in RTSJ, the Jamaica virtual machine implements its own garbage collector with hard real time guarantee. This is obtained by dividing the GC work into small tasks, which are running incrementally during the small interval of the user threads (such an interval is called synchronization point in their publications) as shown in Figure 4-8.

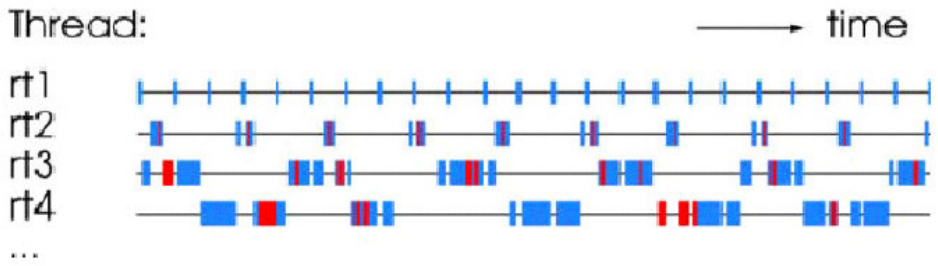


Figure 4-8: Threads Running in Jamaica (Red blocks represent the incremental GC work)[29]

The general techniques in this garbage collector are:

- Saving root references for constant time of root scanning
- Coloring and write barrier strategy for separated marking and sweeping work
- Fixed size heap blocks to avoid memory defragment work
- Automatically inserted synchronization points with the frequency related with allocated memory prove a sufficient garbage collection work being done.

- Priority Inheritance strategy used for avoidance of priority inversion
- JNI supported with real time guarantee, as well as an alternative native interface provided: JBI
- Dynamic linking supported
- Tools provided for real time and embedded system development

Several technologies of performance improvement provided, such as, Ahead Of Time compiling, smart linking, configuration and profiling etc.

4.5 Non-real-time Code Reuse Issue Discussion

Since several real-time products claims that they have such advantages in their system that all the existing Java codes can be reused and gain the real-time performance by simply switching to their real-time Java environments, a question is then raised: *Is it possible that, the existing codes can get real-time performance without any modification by just changing the underlying platform into a real-time one?*

To answer this question, we should remind and summarize some of the results we get in Chapter 2:

First, the essential processes required for building a real-time system can be summarized as follows:

1. Defining the time constraints

   This part of work is apparently the most important and influential part that directly determines the possibilities and difficulties of the consequent works.

2. Analyzing the runtime behaviors of the task set

   This part of work is mainly aimed at the applications with multiple tasks, which require a certain scheduling mechanism provided by the underlying platform.

3. Choosing appropriate scheduling theories and algorithms
4. Applying scheduling theories, such as assigning priorities to tasks according to the chosen scheduling algorithm.
5. Verifying schedulability of the task set by deploying the scheduling analysis

All the above processes are necessary to the real-time system development for its definition, analysis, implementation and verification.

As for the existing code, which is designed and implemented without these real-time issues considered, it may have its runtime behavior analyzed during the design, and may be assigned some deadlines afterwards. But without the third, fourth and fifth processes done, it is impossible for this code to achieve the real-time behavior seriously by just executing in a real-time platform.

After the discussion, we can now give a clear answer for the raised question before: It is not feasible to get the existing non-real-time code real-time by exporting them to a real-time environment.

Therefore, from this point to review the real-time Java solutions again, we know that, the strategy to provide extra semantics and APIs for the real-time development, such as RTSJ has done, is appropriate and will not cause much inconvenience. As for the approaches, which claim to provide real-time feature for existing Java code, it is not so easy to achieve that as it seems to be.

Chapter 5 Evaluation Methodology

In the previous three chapters, we have gathered enough background knowledge of this thesis: real-time system characteristics and real-time scheduling methods; technologies widely used in automotive systems and their future prospects; theoretical analysis on the different real-time Java solutions.

In this chapter, we will approach an effective method to evaluate the chosen real-time Java solutions based on the given background knowledge. There are two goals for the evaluation work in this thesis, one is to reveal the real-time performance provided by the chosen Java environments, the other one is to test the feasibility and flexibility of developing real-time Java automotive applications conducted by the fixed priority preemptive scheduling theory.

The evaluation work is then divided into two subtasks:

First, design a set of benchmark applications in order to test the important parameters of each solution, such as memory allocation time, thread startup time and so on.

Second, design a sample application, which carries the typical logic of the real-time application in automotive systems. The design work here will refer to some of the results got from the benchmark tests and try to practice the real-time scheduling analysis to help and guide the design.

The detailed behavior and functions of the sample applications and benchmark application will be the output of this part of work. To get a clear clue of the work and arrange the tasks in a reasonable way, this thesis adopts a purpose-driven methodology, which means all the works are driven by the purpose of this thesis. That is: to evaluate the applicability of the real-time Java solutions in the automotive systems.

Therefore, the work to define both sample application and benchmark application will be arranged as Figure 5-1 shows.

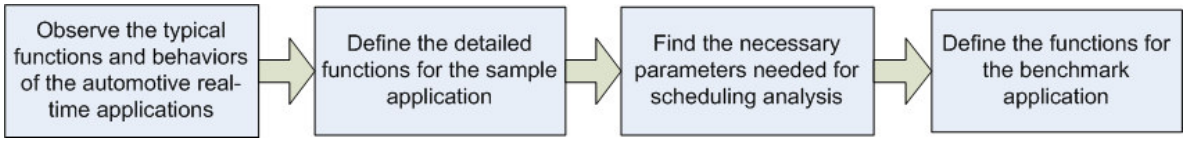


Figure 5-1 Flow Diagram to show the methodology used when defining the functions of sample application and benchmark application

5.1 Survey on the Typical Automotive Real-time Application

The evaluation work in this thesis mainly concentrates on examining the applicability of the candidate real-time solutions in the automotive real-time systems domain. So, to get a clear scope for the requirement of evaluation, we can follow the purpose-driven way to observe what the typical automotive real-time application behaves, where the real-time Java platform should be involved and

what functions the real-time Java is supposed to provide. After such work has been done, we can then reasonably define the functions of the sample application in this thesis.

5.1.1 Survey of Running Environment of Automotive Real-time Applications

As mentioned in Chapter 3, the CAN bus has several natural well-defined properties, which can be used to provide a real-time communication channel among in-car ECUs. Therefore, nowadays, the CAN bus is widely used in the in-car electronic systems to connect ECUs, which are involved in the time-critical tasks, such as Engine ECU or Brake ECU. Normally, one real-time task will be carried out by more than one ECU, Figure n shows the typical network environment, where a real-time application runs.

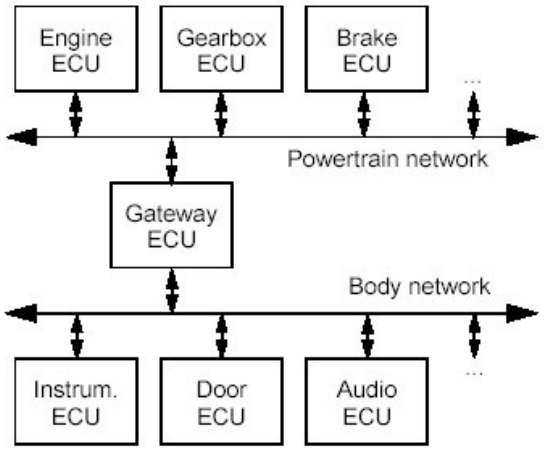


Figure 5-2 Typical CAN network in vehicle and Real-time tasks running environment[37]

Under such typical in-car network, a typical real-time task's behavior is described as the following steps:

1. The Engine ECU examines the status of engine every 100 milliseconds and send a message containing this information onto the CAN bus.
2. The Gateway ECU listens and receives these messages from the bus, and at the same time receives other similar messages that carry status information of other parts of the vehicle.
3. In the Gateway ECU, every time when some new Engine status message arrives, a specific handler program will handle this message and do some necessary calculation to check if the engine is currently in a healthy state. In some more complicated cases, this handler program could combine several status messages and check the synthetic status for a set of in-car components.
4. If after the calculation, it indicates that the engine has been in or would possibly be in an unsafe state, some appropriate operations must be managed and deployed by the handler

program within a predefined deadline, such as sending warning message to Displayer ECU to notify the driver or sending control message to the Brake ECU to slow down the car etc.

5. The related ECUs get the above emergency messages and make the corresponding reactions as informed by the messages. In such a way, a potential danger can be avoided.

As we can see, most of the logic and actions in the workflow of this real-time task resides in the Gateway ECU. It acts as a data collector, a status inspector and also an emergency routine operator. This essential role of the Gateway ECU determines that, it has more powerful hardware support than the other end node ECUs. So some more complex software can run on it such as a real-time operating system or other real-time application platforms.

A Java virtual machine normally runs on top of a specific operating system[1] and hence demands a relatively more powerful hardware platform than other embedded programming languages. Therefore, importing Java into the Gateway ECU could be a most natural idea. In this thesis, we will follow this idea and build up a test bed environment as the typical environment shown in Figure 5-2 for the evaluation work. More detail about the test bed environment, one can refer to Chapter section of this thesis.

Since the physical environment (including the target device and the network topology) for deploying real-time Java has been chosen, let us observe behaviors of typical automotive real-time tasks more in detail so as to reveal the functions and responsibilities that the Gateway ECU has.

5.1.2 Survey of Typical Automotive Real-time Tasks Behaviors

As described in the previous subsection, most of the real-time tasks are carried out by more than one ECU. The hardware platforms of these ECUs can vary a lot depending on their different functionalities. For some important ECUs, such as Gateway ECU acting as a central gateway connecting different buses, their processor speed can be several Million Hertz. While, for some simple ECUs that only take the responsibility of obtaining data from the sensor and put them onto the bus in a fixed rate, it is enough for them to be running in a processor with just several Kilo Hertz. Apparently, not all of the ECUs involved in a real-time task can be equipped with a real-time Java environment. Therefore, for the scope of this thesis, we must inspect the functions, which 'Java' ECUs are supposed to take and try to separate them from the entire real-time task. After such a work done, we can then confine and concentrate on the subtask that 'real-time Java ECU' takes.

As mentioned in section 2.1.2 in this thesis, to build a hard real-time system, we need to adopt the layer-by-layer approach to ensure the predictability. A rough partition of layers can be shown in Figure 5-3. Here we assume that, the gateway ECU has been equipped with one specific real-time Java environment. Then, we got four layers:

- Function layers: Represent the specific real-time task's logic and workflow

[1] Here, we assume that there is no special hardware support for Java code.

- Compiler and Interpreter layer: This layer is introduced here to provide a running environment for Java. This should be replaced by an individual Real-time Java solution during the evaluation.
- Communication Layer: This layer abstracts all the underlying communication media. It can be roughly taken as a composition of the Data Link layer, Network layer and Transport layer of the ISO/OSI reference model. All the message routing, message composing, decomposing and the frame management are handled in this layer.

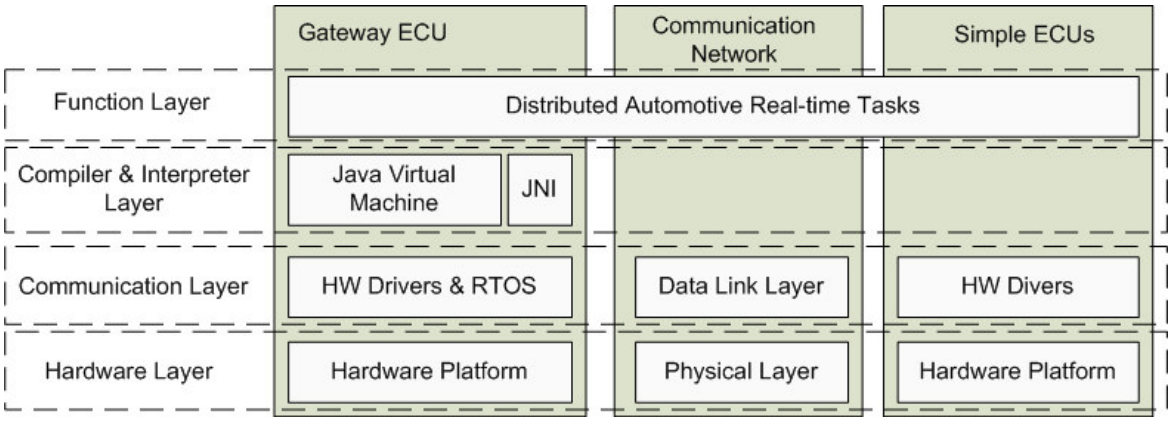


Figure 5-3 A layered view on the implementation of an automotive real-time task

- Hardware Layer: This layer represents all the necessary hardware needed in this environment. Including the CPU, RAM, I/O devices and so on.

Now we can inspect the predictability of these layers bottom up.

The hardware layer's predictability is not difficult to guarantee. Normally, with the help of profiling tools, people can calculate the needed processor cycles out by analyzing the machine codes (instructions) of the applications, and then easily get the execution time by dividing the frequency of the processor. It is always true that the hardware is much more predictable than the software.

The predictability of the communication layer largely depends on the protocols applied in the network. By adopting some real-time featured protocols, such as CAN, TTCAN, TTP[1] or Flexray, the message transfer time on this layer can be bounded and predicted.

For the compiler and interpreter layer and the function layer of the Gateway ECU, their predictability will be focused and examined later in this thesis. The function layer's predictability in simple ECUs can be provided by means of low level programming or with the help of some light weight RTOS.

After the above analysis, we can draw the following conclusion that, except for the function layer and compiler and interpreter layer of the Gateway ECU, all other parts, which are involved into a distributed automotive real-time task, can achieve timing predictability by deploying some appropriate technologies. Thus, we can separate the entire real-time task into several subtasks:

[1] TTP denotes for Time Triggered Protocol.

1. Simple ECUs, in a fixed rate, retrieve status information of in-car components and send them onto the network.
2. The messages go through the communication layer. From simple ECUs, passing the network to the Gateway ECU.
3. The Gateway ECU receives these messages, does some necessary calculation and checks the status of the whole set of in-car components. In case that some inappropriate situations are found, the Gateway ECU will react by sending corresponding messages to the related ECUs in the vehicle.

As the first and second subtasks in the above list have been proved to be able to provide predictability. We can therefore split the task, and also the time constraints. Then, each of the subtasks will have their own deadline and can be treated separately.

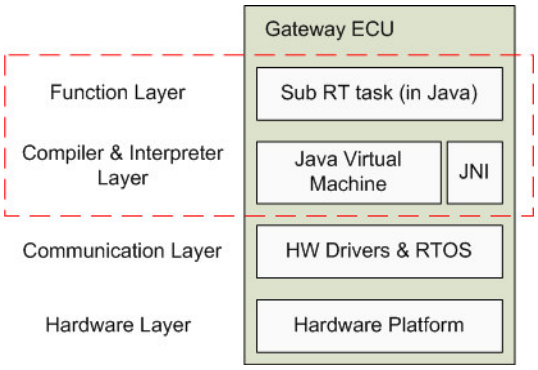


Figure 5-4 Separated View of Gateway ECU and its subtask

In such a way, we can eventually confine the position and functions of the target real-time Java platform. As shown in Figure 5-4, the field encircled by the dashed represents the main area this thesis focuses on. Both the sample automotive real-time application and the benchmark applications are built to run within this field. The rough functions or to say behaviors of the sample application could be:

- Handle the periodic messages events from the network
- Calculate and check the new states of the specific set of in-car components
- Send emergency messages in case of improper situation observed

5.2 Define Functions for Sample Automotive Real-time Application

After the behaviors of the sample application are confined, we need to provide a detailed context and realistic task to specify certain functions of the sample application so that for the next step, both the scheduling analysis and the sample application design can be launched based on it.

The functions of the sample application will be specified by the following three parts: specification of the incoming messages; the workflow of each message handler and the time constraints.

5.2.1 Incoming Messages

The specification of the incoming messages can be organized as a table. There are altogether four incoming messages for the sample application to handle. Their name, description, period of sending are shown in Table 5-1

Message Name	Message Description	Send Period
Vehicle Speed Message	Contains the newest speed of the vehicle	100ms
Tire Pressure Message	Contains the newest values of all tires' pressures	500ms
Steering Wheel Angle Message	Contains the newest angle value of the steering wheel	200ms
Obstacle Distance Message	Contains the newest distances from the vehicle to the obstacles around the vehicle	100ms

Table 5-1 Incoming Messages Information

5.2.2 Workflows for Each Message Handler

The workflow of handling 'Vehicle Speed Message' is described by Figure 5-5.

For this part of the sample application's work, the handler only needs to update the newest vehicle speed value in the memory. But, for the consideration of mutual exclusion, it has to make the update through the Java monitor of the speed object.

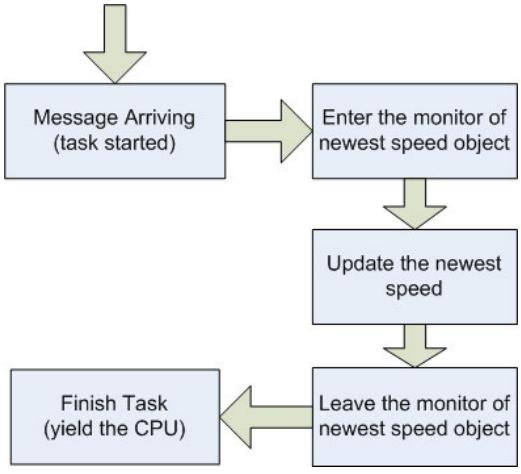


Figure 5-5 Workflow of handling Vehicle Speed Message

The workflow of handling 'Tire Pressure Message' is described by Figure 5-6.

After the arrival of 'Tire Pressure Message', the handler should check the status of all tires' pressure values, if any of them exceeds the predefined upper bound value or goes below the lower bound value, the handler will send the corresponding message onto the bus for proper reaction.

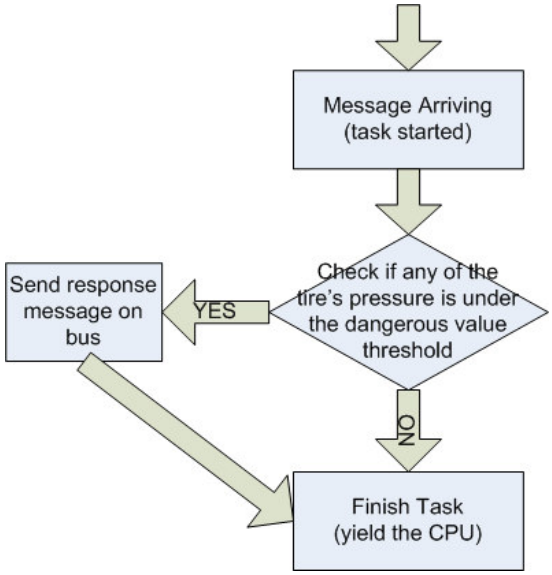


Figure 5-6 Workflow of handling Tire Pressure Message

The workflow of handling 'Steering Wheel Angle Message' is described by Figure 5-7.

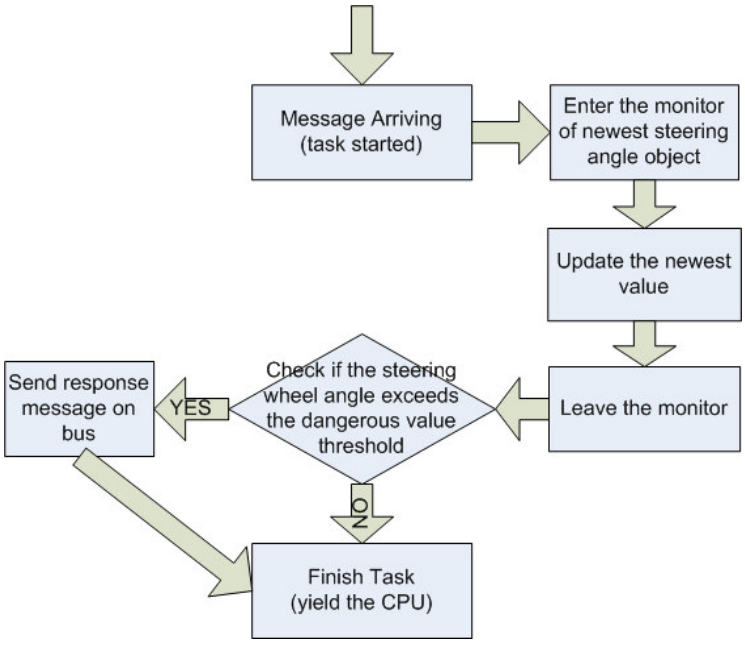


Figure 5-7 Workflow of handling Steering Wheel Angle Message

After the 'Steering Wheel Angle Message' arrives, its handler will be invoked to run. First, it will enter the monitor of the steering angle object, which stores the newest value of steering wheel angle in the memory. When the handler gets the lock, it then updates the object with the newest value it gets from the message. After leaving the monitor, the handler begins to check whether the angle's value is

too large and exceeds a predefined threshold. If it does, a corresponding message will be sent on the bus to make the reaction.

The workflow of handling 'Obstacle Distance Message' is described by Figure 5-8.

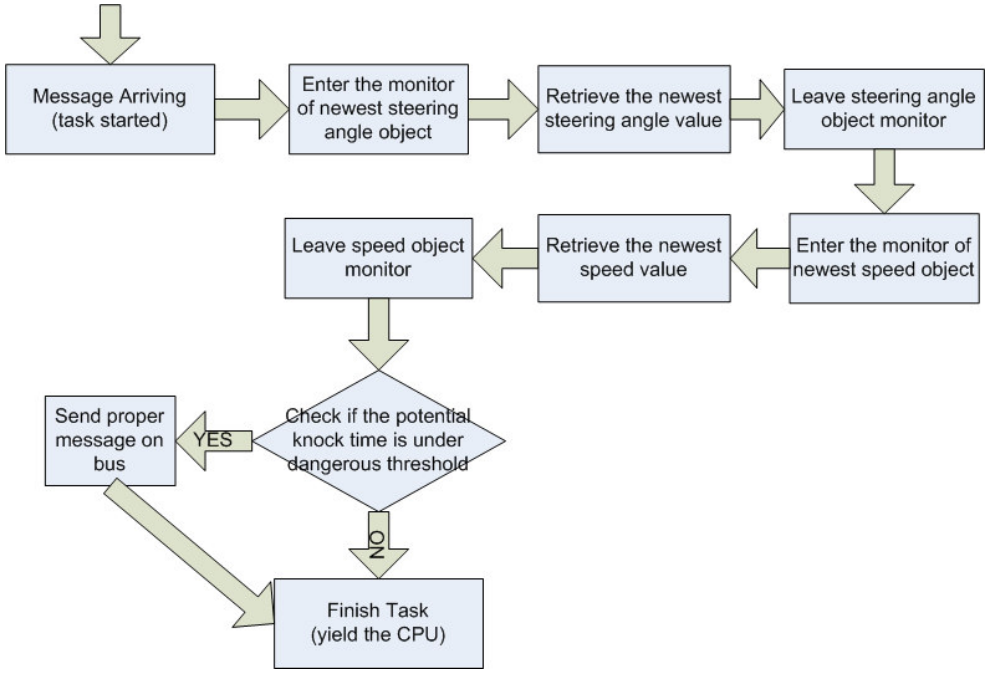


Figure 5-8 Workflow of handling Obstacle Distance Message

The handler of Obstacle Distance Message first needs to retrieve the steering angle value from the memory through the monitor of the steering angle object. And then, it goes into the speed monitor to get the newest value of speed. After enough information is gathered, the handler calculates the potential knock time of the vehicle to the obstacle according to three parameters: vehicle speed, steering angle and obstacle distance. If the time result indicates a dangerous state, the handler will send out a proper message onto the bus, for example a control message to stop the vehicle.

5.2.3 Time Constraints in the Sample Application

The following table shows the deadlines of each message-handling task:

Message Handling Type	Period (ms)	Deadline (ms)
Speed Message Handling	**100**	**100**
Tire Pressure Message Handling	**500**	**200**
Steering Wheel Angle Message Handling	**200**	**150**
Obstacle Distance Message Handling	**100**	**70**

Table 5-2 Messages handling deadlines

The start time phase of each message is randomized. But once the first message of each type is sent, the consequent messages of that type will follow it being sent in the predefined fixed rate.

5.3 Finding Necessary Parameters needed by Scheduling Analysis

Given the requirement for the sample application, the design work could consequently follow. However, before working on the design of the application, we will first try to make a scheduling analysis for the whole task set, try to reveal the necessary parameters we need for the analysis. These necessary parameters can then guide the benchmark application design, which means that an important purpose that the benchmark application takes is to provide the necessary parameters for the scheduling analysis. At the same time, through comparing the different values of the same parameter in different Real-time Java environments, the real-time performances of these RTJ solutions can be evaluated.

First, we need to fix the scheduling algorithm we will use. Since several attributes of Java determine that only the preemptive fixed priority scheduling algorithms can be applied, we will only focus on and choose the algorithm from this category.

When choosing between the Rate-monotonic scheduling theory and deadline-monotonic scheduling theory, we find that the tasks' deadlines shown in Table 5-2 are not all equal to their period; three of them have the deadlines shorter than their periods. Therefore, we believe that, the deadline-monotonic scheduling theory is more suitable to be applied in our case.

Before introducing the fixed priority preemptive scheduling theory into analysis, we assume that all the real-time Java platforms we evaluate can support this scheduling theory and can therefore provide bounded and predictable values for all necessary parameters required during the analysis.

Now let us examine the necessary parameters step by step.

PRIORITY ASSIGNMENT

First of all, all the four handlers can be taken as the periodic tasks. Their priorities will be assigned according to the deadline-monotonic theory, which means the task with earlier deadline will be assigned a higher priority. The final priority assignment is shown in Table 5-3. Here, the priority of each task is set relatively to the highest priority of the four, denoted by P. The value of P will be determined respectively under each real-time Java environment to be evaluated.

Task No.	Task Description	Period (ms)	Deadline (ms)	Priority
1	**Handling Speed Message**	**100**	**100**	**P-1**
2	**Handling Tire Pressure Message**	**500**	**200**	**P-3**
3	**Handling Steering Wheel Angle Message**	**200**	**150**	**P-2**
4	**Handling Obstacle Distance Message**	**100**	**70**	**P**

Table 5-3 Priority assignment of the tasks in the sample application

THREAD SYNCHRONIZATION CONSIDERATION

Considering three message handlers need the use of two mutual exclusive resources, the blocking issues must be counted. So, we can first try the formula (2.1) on our task set:

$$W_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil C_j$$

Here, $Task_i$'s response time R_i can be given by the smallest W_i^n, where $W_i^n = W_i^{n-1}$

The parameters to examine in this formula:

T_j is the period of each task, its value for each task can be found in Table 5-3.

B_i is the worst-case blocking time for $Task_i$, we have assumed this blocking time would be bounded in each of the real-time Java solution, though this depends on the specific priority-inversion avoidance mechanism applied in each solution. We will examine this characteristic in each test Java environment in the benchmark tests.

C_i is the worst-case execution time for $Task_i$ despite of higher priority tasks preemption or blocking issues. We will follow the routines concluded in 2.5.3 to examine the value of this parameter.

CONTEXT SWITCH OVERHEAD CONSIDERATION

In practice, the overhead of two Java threads' context switch cannot be neglected. According to the discussion in the Chapter 2, by importing two parameters: C_{sw}^P and C_{sw}^R, we can extend our formula into:

$$W_i^{n+1} = B_i + C_i + C_{sw}^P + C_{sw}^R + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil (C_j + C_{sw}^P + C_{sw}^R)$$

where Table 2-2 can be referred for C_{sw}^P and C_{sw}^R's definitions.

Here, $Task_i$'s response time R_i can be also given by the smallest W_i^n, where $W_i^n = W_i^{n-1}$

Observing that C_{sw}^P and C_{sw}^R always appear as the sum value of the two in the formula. We can combine them into one parameter $C_{sw} = C_{sw}^P + C_{sw}^R$ for convenience and test its value in the benchmark application. Thus, the formula can shorten into:

$$W_i^{n+1} = B_i + C_i + C_{sw} + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil (C_j + C_{sw})$$

RELEASE JITTER CONSIDERATION

Though we have assumed that, each of these four tasks can be taken as a periodic task, there will still be some time that one of their releases is delayed. Especially, if we consider the Gateway ECU is not the generator but the receiver of the messages, the arrival time of the messages could be delayed because of the jitter that occurred in the communication media. Therefore, we need to bring in the release jitter parameter into the formula we use.

$$W_i^{n+1} = B_i + C_i + C_{sw} + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n + J_i}{T_j} \right\rceil (C_j + C_{sw})$$

where, J_i denotes the maximum release jitter delay that $Task_i$ may meet.

Here, $Task_i$'s response time R_i can be given by $J_i + W_i^n$ where W_i^n is the smallest one that satisfies the equation $W_i^n = W_i^{n-1}$

Since there can be many possible reasons that can lead to a release jitter and some of them are out of scope of this thesis, we will try to determine the value of J_i in an empirical way during the evaluation.

To make a summary of all the necessary parameters for the scheduling analysis for sample application (including those we have known), we get the following table:

Parameters	Description
T	Period between two consequent task releases
D	Deadline relative to the release of task
J	Maximum release jitter delay
****B***	Maximum blocking delay
******* C_{sw}	Context-switch overhead (Equal to $C_{sw}^P + C_{sw}^R$)
****C***	Worst-case execution time of the task (without blocking or preemption)
P	Priority of the task (determined by specific scheduling theory applied)
R	Response time of the task (calculated)

Table 5-4 Necessary parameters for the scheduling analysis

(Parameters with * should be part of the benchmark's criteria)

5.4 Define the Functions of Benchmark Application

There are altogether two criteria that can guide the benchmark application's functions. The first criterion is the parameters required by the scheduling analysis work for the sample real-time application, which we have concluded in the Table 5-4; the second criterion is the most important features for real-time Java that are listed in the Real-Time Specification of Java, as we have introduced in 4.2.1.

After combining these two criteria together, we get a list of functions that the benchmark application should provide, as shown in Table 5-5.

Function Name	Description
Memory allocation time test	Test the time for one real-time Java platform to allocate a certain size of memory (for the worst-case execution time of tasks)
Thread startup time test	Test the time for a certain real-time Java platform to start a new thread
Priority Inversion test	Test if an effective mechanism to avoid priority inversion is provided by a certain real-time Java platform
Thread context switch time test	Test the time for a certain real-time Java platform to switch the context from one thread to another
JNI access time test	Test the time for a certain real-time Java platform to access the native code through Java Native Interface
One-shot Timer test	Test the precision of the one-shot timer in a certain real-time Java platform
Periodic Timer test	Test the precision of the periodic timer in a certain real-time Java platform
Asynchronous event handling test	Test the performance of a certain real-time Java platform to handle asynchronous events

Table 5-5 Functions to be provided by the benchmark application

Some comments on these functions:

The test results from the tests in the benchmark should only be considered as an experimental reference for the scheduling analysis work. For more accurate exact data, a large scale and long period of tests must be taken and some probability and statistics methods have to be used to analyze on the results.

Since the benchmark application also takes the responsibility of testing the real-time Java solutions in a general way, some of the test results are not aimed to serve the scheduling analysis, such as the timer test, asynchronous event handling test and so on. The results from these tests are more reflecting the overall performance of the candidate solutions.

Chapter 6 Design and Implementation

The implementation work in this thesis can be divided into three parts. Besides the sample automotive real-time application and the benchmark application suite, to better deploy and analyze the final evaluation, a remote graphic controller application is implemented to help the control and result generation. So, in this chapter, all these three parts of implementation work will be introduced, including their design and the detail logic of codes in some essential parts.

6.1 Benchmark Application Design and Implementation

The functions that the benchmark application should provide have been concluded in Table 5-5 in Chapter 5. According to these functions, we can visualize the behaviors of the benchmark application by a use-case diagram (see Appendix A).

Before we go into the design of each individual test in the benchmark, we find that a high-resolution timer is necessary for the benchmark because all the tests need a time precision within a sub-millisecond level.

6.1.1 High Resolution Timer for the Benchmark

In RTSJ, an explicit requirement has been raised that, a class called '*HighResolutionTime*' should be provided in its implementations. In such a way, a nanosecond accuracy of time can help the real-time application developer to create more powerful and serious real-time application. However, none of the real-time Java solutions to be tested in this thesis support this feature, we have to make another approach to this high-resolution timer utility.

In the conventional Java program, the timing behavior can be done by calling static method: *System.currentTimeMillis()*. This method returns the time difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. The problem of using this method is that it cannot be used to measure some sub-millisecond time interval, furthermore the time precision that one can get from this method depends on the tick value of the underlying platform. For example, in the Windows 2000 operating system, the tick period is 1 millisecond and in most of the Linux OS implementations, this parameter is 10 milliseconds in value; for the real-time operating system VxWorks, it has a tick period of 500 microseconds, which is half a millisecond. Therefore, we cannot rely on this method to generate time periods with high precision.

To obtain more accurate time than the underlying system, we have to turn to some platform dependent APIs, which provide the way to retrieve CPU frequency and the CPU cycles that are used by the test period. The target operating system QNX RTOS, supports such APIs in its native libraries. Then, a natural idea will be: making use of these APIs through Java Native Interface and getting high-resolution time by simple calculation. This thesis adopts this idea and develops a QNX high-resolution timer in this way. The following sequence diagram can briefly describe the logic to retrieve high-resolution time in this thesis (Figure 6-1).

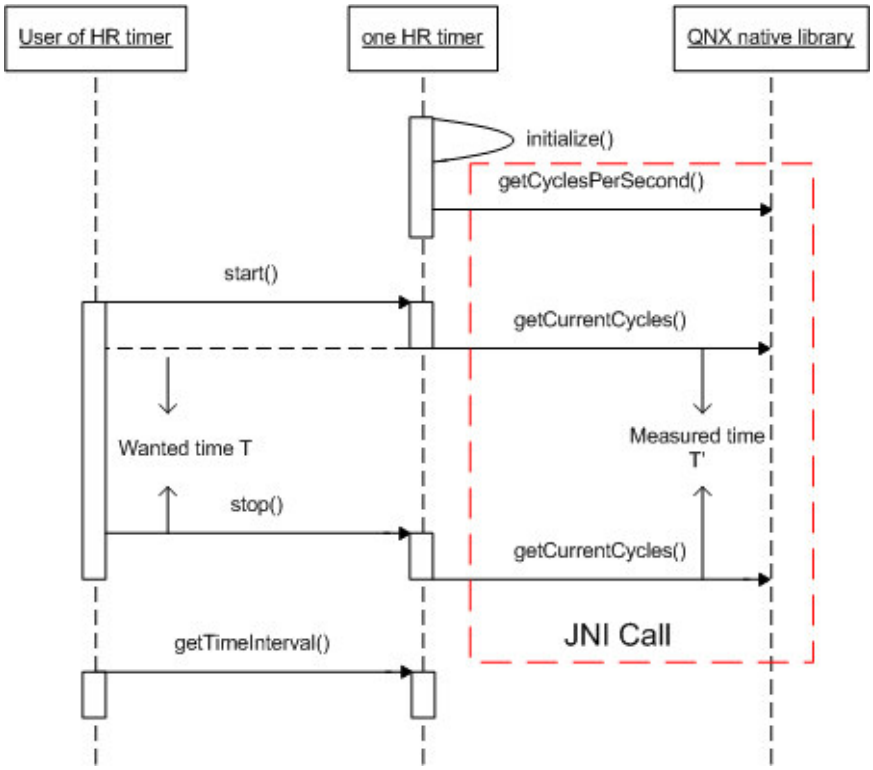


Figure 6-1 Sequence Diagram of retrieving high resolution time in benchmark application

A consequent issue is that, to measure the time of a sequence of codes, we need to invoke the high-resolution timer two times: to start it and to stop it. The time overhead of the two JNI accesses cannot be neglected, especially for measuring some small time costs.

From Figure 6-1, we can see that, the time we want to measure (T) and the actual value we can get (T') have the equation: $T = T' - T_{JNI}$. From experiments done on the target Java platforms, the one time JNI access time costs at most tens of microseconds and normally only takes several microseconds. Therefore, to test the time period in the millisecond level, this timer overhead can be neglected, otherwise, the average timer cost should be measured and reduced from the test results. In a special case, if the purpose of the test is to examine the worst execution time of some operations, the reduction can be cancelled so that the final result won't be under-estimated by reducing the average cost.

Thus, the high-resolution timer is prepared, and we can continue the benchmark implementation with the help of it.

6.1.2 Memory Allocation Time Test

The reason of including a memory allocation time test is because, a Java program has to manipulate with memory very often as in an object-oriented language. That is also part of the reason why Java decides to provide a garbage collector to release the programmer from complicated memory

operations. But for a real-time system, the memory allocation time must be predictable so that the worst execution time can be calculated. The following class diagram can show the structure of this part of design.

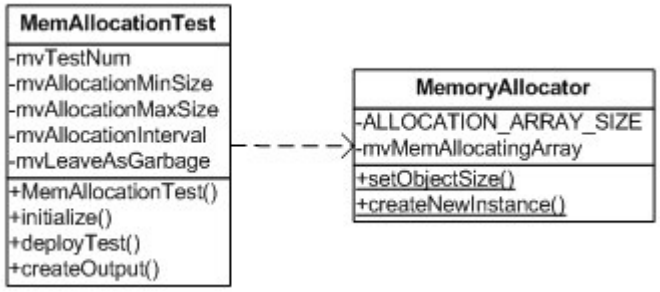


Figure 6-2 Class Diagram for the Memory Allocation Time Test

The main idea to create various sizes of objects for allocation is to define a member variable in a class with the type of byte array. The length of this byte array can be adjusted through the static method in the class, and as we know, the member variable will only be initialized when the object is being constructed. So, in such a way, we can get any specific size of object as we want.

One more thing to mention is that, any class in Java has to inherit from the class Object, and hence every class has the 8 bytes of basis as being a Java object. In addition, each reference member takes 4 bytes and an instantiated Java array takes at least 16 bytes to store its necessary parameters like length, start address etc. Therefore, for an exact memory allocation simulation, the design of the *MemoryAllocator* class takes these issues into consideration. As Figure 6-2 shows, *MemoryAllocator* has a member byte array variable *mvMemAllocatingArray* for the use of various sizes of memory allocation. A static variable *ALLOCATION_ARRAY_LENGTH* is to adjust the initial size of this array. Notice that an object of *MemoryAllocator* class has a size comprised of 8 bytes object shell; 4 bytes for the reference of the byte array; 16 bytes for initialized array shell; *ALLOCATION_ARRAY_LENGTH* of bytes for array body. Therefore, given the expected memory size n bytes to allocate, *ALLOCATION_ARRAY_LENGTH* should be equal to n–4–8–16 = n–28. This logic is handled in the static method: *setObjectSize().* A consequent call to the method *createNewInstance()* can then allocate the memory block with the required size.

For the convenience of the test, there are several parameters for user to adjust, such as, allocation times, interval between two allocations, whether leaving the created objects as garbage or not and so on.

6.1.3 Thread and Synchronization

In this part of the benchmark, there are altogether five tests. Four of them rely on the quantitative results. They are thread startup time test, thread entering unlocked monitor time test, thread notification context switch time test and thread yielding context switch time test. The other test is mainly designed for revealing the characteristics of the virtual machine: priority inversion test.

The purposes and design criteria of these tests will be explained one by one in the following subsections.

6.1.3.1 Thread Startup Time Test

Thread definition in the Java language supports the developers to create more than one thread at runtime. These threads can run concurrently in the application sharing the same memory region. Whenever a new thread is created and started, there will be one more branch of the code being executed independently from the existing ones. Multi-threaded programming is very useful for designing and implementing systems with a large size and sophisticated logic. Consequently, whether the startup time of a thread is steady and predictable is fairly important for the real-time application developers. Therefore, the benchmark application in this thesis includes this test.

This test application mainly measures the time period from a thread object's *start()* method is invoked untill the first sentence of method *run()* is reached. The thread number to test and the interval between two threads' start are configurable to make the test more flexible.

6.1.3.2 Thread Notification Context-switch Test and Thread Yielding Context-switch Test

There are two thread context switch tests in the benchmark applications of this thesis: thread notification context-switch test and thread yield context-switch test. They are basically classified by the ways of causing threads' context switch. The prior one is through the current running thread notifying a waiting thread; the later one is through the running thread voluntarily giving up the CPU and so awaking the other waiting threads.

The main thought of using thread notification to cause context-switch is that:

In the beginning of the test, a thread with a high priority starts to run in the system. It then suspends itself through invoking the *wait()* method provided by Java *Object* class. The main thread in the system which has the normal priority then takes over the CPU time, it does two things at this time, first, it triggers the *startTimer()* method of the *HighResolutionTimer* class described earlier to start timing, and then, it grasps the monitor of the high priority thread and *notify()* the thread which is waiting. Thus, the previous high priority thread will be brought from the *suspended* state to the *runnable* state (Figure 6-3 shows the basic thread state transfer logic). When this high priority thread backs to execute, it immediately stops the timer and thus, a thread context-switch is accomplished and timed. Figure 6-4 can show the execution sequence of the notification context-switch test.

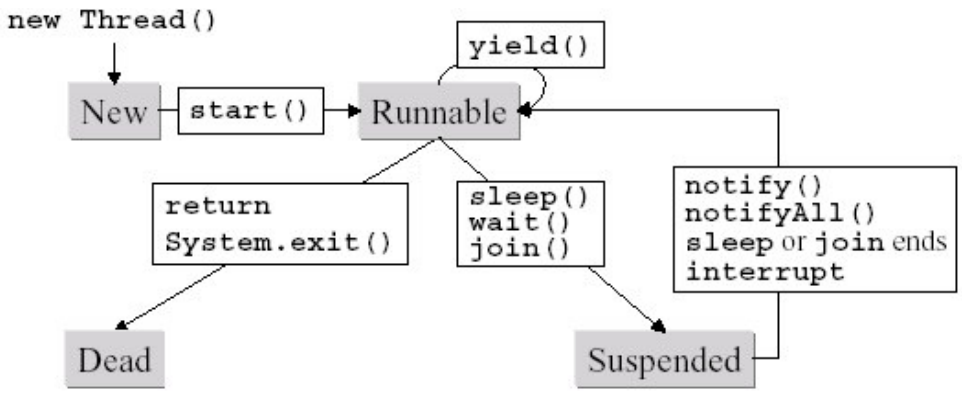


Figure 6-3 Java Thread life-cycle and state-transfer diagram

One more thing to mention is that the Java Language Specification does not explicitly define the mechanism of picking one thread from several threads that are ready, which means that in a specific Java virtual machine thread priorities may not determine the eligibility of the threads at runtime. However, in a real-time environment a high priority thread that is ready to run should always be given the eligibility to run first among the runnable threads because this is an important prerequisite for the deployment of the priority-based scheduling theory. We can say that without such a guarantee the priority-based Java thread mechanism will be meaningless and infeasible for the real-time application development. Therefore, in this application, such a guarantee is taken for granted for the real-time Java virtual machines to be tested. In addition, RTSJ also explicitly defines that thread's priority should be considered as execution eligibility in its implementations.

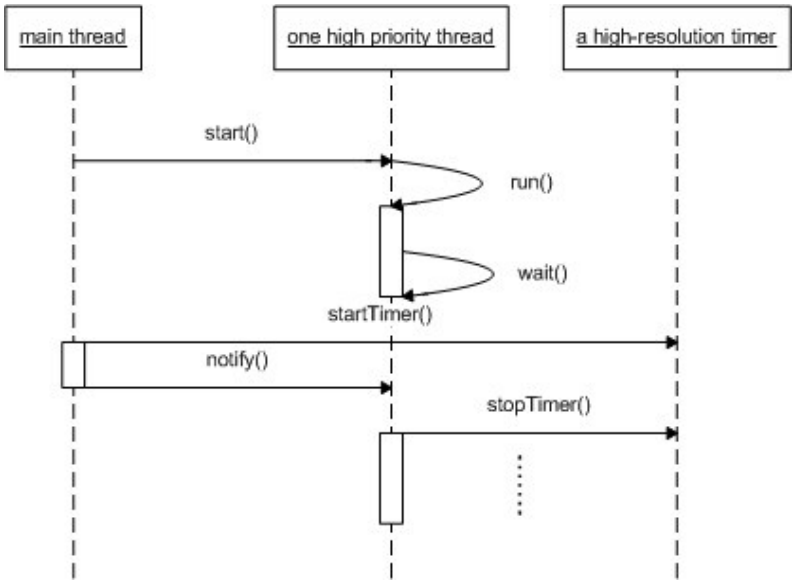


Figure 6-4 Sequence diagram: thread notification context-switch time test

The thread yielding context-switch test then makes the use of *yield()* method in the *Thread* class to achieve the context-switch between threads. When doing the design of this part of test, one thing

should be cared about is that, the *yield()* method doesn't suspend a thread, instead, it just takes the current thread off the CPU and puts it back into the queue of ready threads. Therefore, to make sure that another thread could take the context after that, only two runnable threads are kept in the system and both of them have the same priority so that they have the same eligibility when being chosen to run. Thus, one thread's yielding will consequently lead the context switch to the other thread. In Figure 6-5, the sequence diagram shows the main workflow in this part of test. Notice that, to cause the context switch, after starting two low priority threads, the main thread will suspend itself for a sufficient time in order to leave the two threads with the same priority in the system.

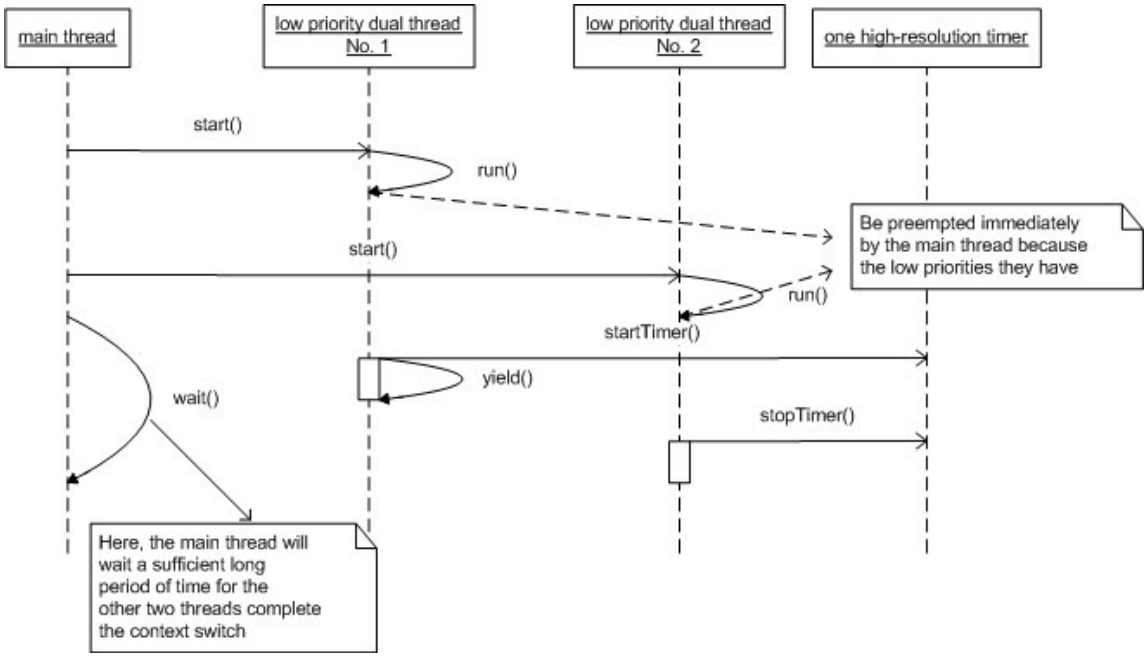


Figure 6-5 Sequence diagram: thread yielding context-switch time test

Some Comments on the context-switch test:

Though the scenarios described by the two context-switch tests cannot include all the control transferring circumstances of the system. Still, they can partially show the stability and efficiency of the thread context-switch work in a specific Java runtime environment.

6.1.3.3 Priority Inversion Test

To test whether the priority inversion problem exists in a specific virtual machine to be tested. A priority inversion test is design in the benchmark application in this thesis. To recur the priority inversion scenario in this test, four kinds of threads with four different priorities are introduced: the main thread acting as the thread dispatcher which has the highest priority; a high priority thread which has the second highest priority in this test; a low priority thread and a specified number of medial priority threads. The runtime interactions of these threads can be shown in Figure 6-6. As this sequence diagram shows, the main thread will start the low priority thread first and let it enter a

predefined monitor, and then the main thread is notified by the low priority thread, which is in its critical section, and invoke the high priority thread and all medial priority threads respectively. After that, the main thread suspends itself and waits until all the threads terminate. The completion time of each thread (except for the main one) will be recorded to see if the priority inversion has occurred. If the priority inversion problem is successfully avoided by the tested real-time Java virtual machine, the completion times of all these threads can also illuminate the efficiency and extra cost of implementing this.

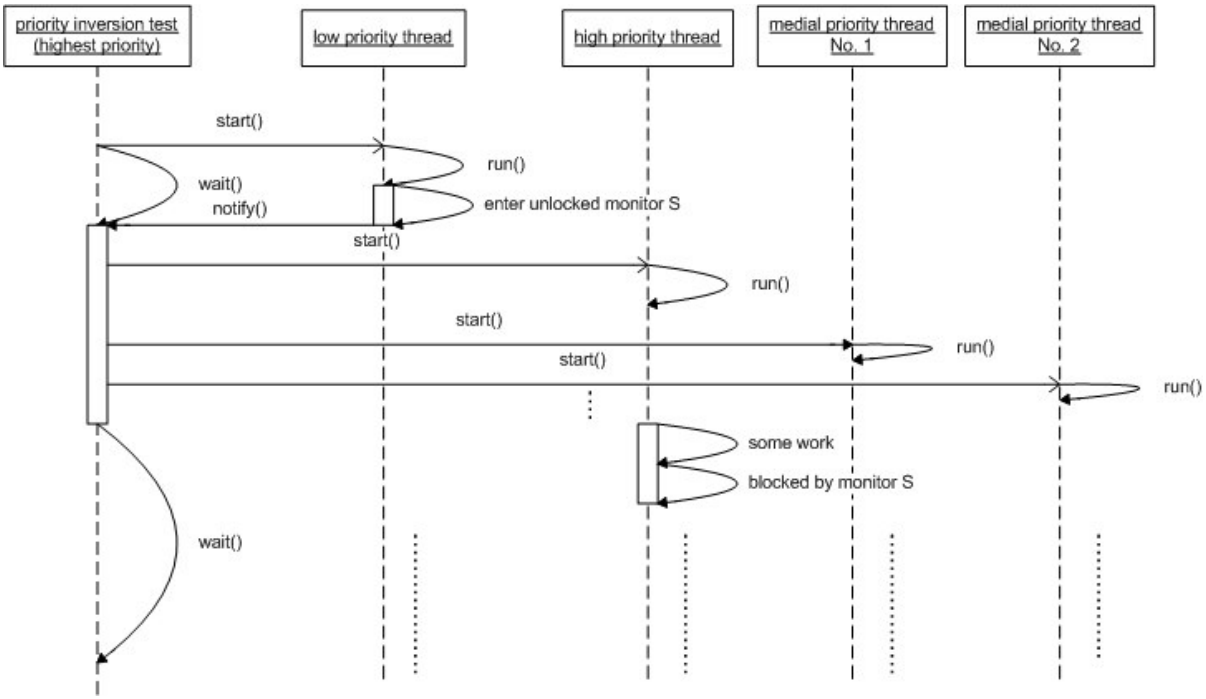


Figure 6-6 Sequence Diagram: thread priority inversion test

6.1.3.4 Entering Unlocked Monitor Time Test

The logic for this 'Entering unlocked monitor time test' is fairly simple, a thread in the application is defined to enter and leave an exclusive monitor for specific times, then the maximum, average and minimum time cost will be calculated and recorded.

6.1.4 Timer Test

The timer test application in the benchmark is comprised of two tests: a one-shot timer test and a periodic timer test. Since there is no RTSJ implementation in the chosen test virtual machines, the timer classes used in these two tests are different from the *OneShotTimer* and *PeriodicTimer* defined in RTSJ. In these two tests, the common timer class: *java.util.Timer* and the corresponding interface *java.util.TimerTask* will be used and examined.

6.1.4.1 One-shot Timer Test

The one-shot timer test, as its name indicates, inspects the time precision of the timer that only releases once. This test make use of the method *schedule(TimerTask task, long delay)* in the class *java.util.Timer* to define the task and its one-shot release time. The time period from this method invoked to the *run()* method of the scheduled task is called will be measured by the high-resolution timer, in such a way that the time difference between the expected release time and actual value can be determined and recorded. The main workflow of this test can be shown by Figure 6-7.

Such one-time tests can be repeated for a configured number of times for the convenience of test deployment and results collection. The average high-resolution timer overhead is removed from the raw results.

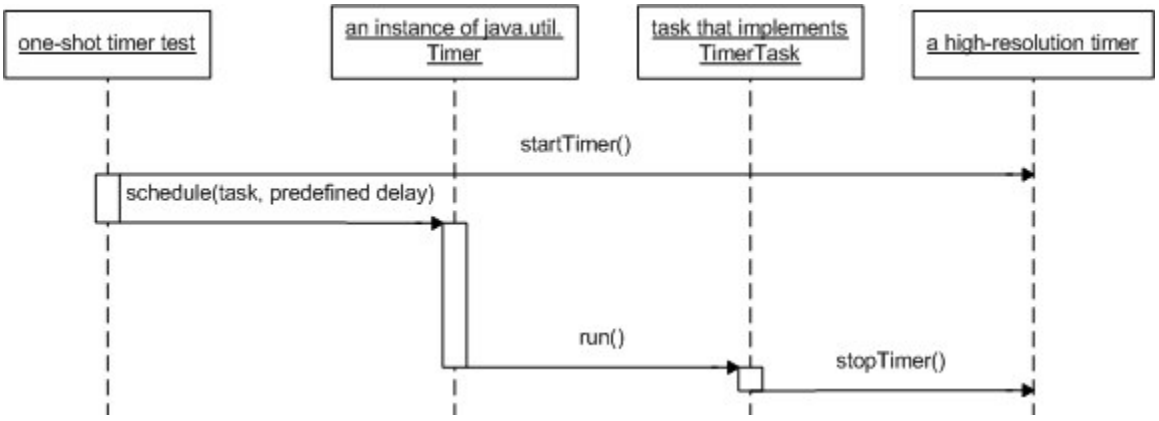


Figure 6-7 Sequence Diagram: One-shot timer test

6.1.4.2 Periodic Timer Test

The period timer test is conceptually very similar with the one-shot timer test. It also aims to test the precision of the *java.util.Timer.* Instead of examining the release of the task only once, this test inspects the precision of a series of periodic releases of one task scheduled by the timer. Like the one-shot timer test, the high-resolution timer is used to measure the actual release time and its overhead will also be considered.

6.1.5 JNI Access Time Test

Java Native Interface (JNI) is a very effective technology to connect java application with the platform dependent native codes or libraries, such as hardware driver or necessary service APIs provided by the operating system. As for the embedded system development domain, JNI is even more essential for the Java implementation. Therefore, the stability and predictability of JNI access is a very important criterion to evaluate a certain real-time Java solution.

The time cost of a JNI access can be divided into two parts:

- The time for the Java program to enter and leave the native interface.

- The time for the native code to interact with the local library

Apparently, the second part of the time cost could only be determined according to the specific library. In case of one real-time application's implementation, the predictability of the native library should be examined and proved by other means rather than within the Java environment. Therefore, in the JNI test of this thesis, only the first part of the time cost will be considered and examined.

According to the ways of interacting with native interface, the JNI test in this thesis is classified into the following seven types:

- Get a byte value from native interface
- Get an int value from native interface
- Get a double value from native interface
- Get a byte array with predefined length from the native interface
- Inform the native interface to make a call-back to the Java program
- Inform the native interface to modify a member variable of Java
- Inform the native interface to modify a parameter which is passed to it

Each of these seven ways of accessing JNI is implemented both at the Java side and at the native side. The time costs of them are recorded with the help of high-resolution timer. The time overhead of high-resolution timer is necessarily taken into consideration because this overhead is relatively too large to be neglected. Furthermore, all the above tests can be iterated for a pre-configured number of times for the convenience of large-scale tests.

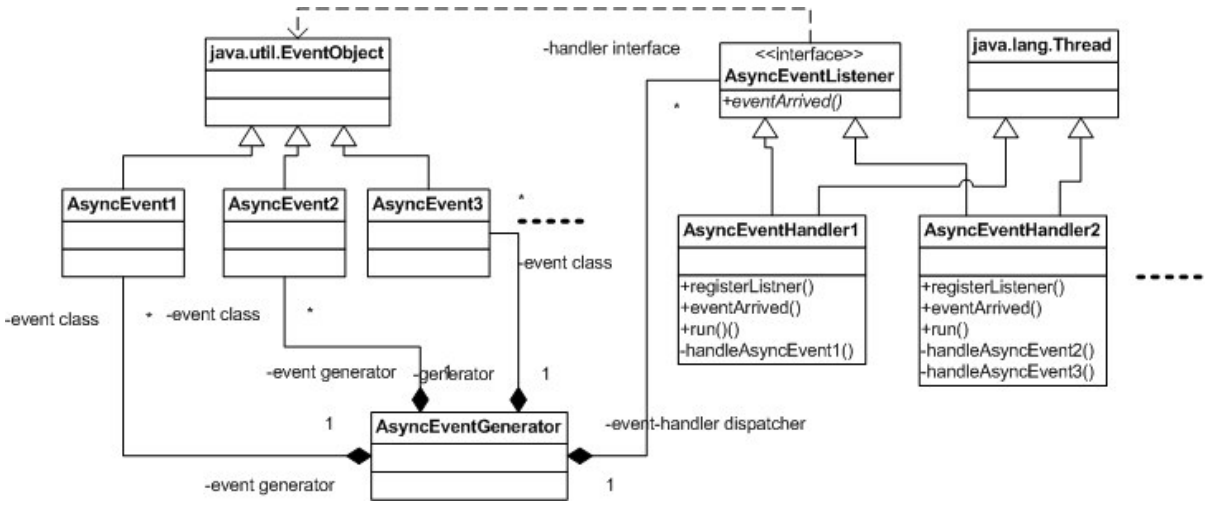


Figure 6-8 Class Diagram: asynchronous event handling test

6.1.6 Asynchronous Event Handling Test

As mentioned in 4.2.5, in RTSJ, the mechanism to effectively handle large amount of asynchronous events is specified. The main idea behind this mechanism is to use less threads while at the same time to handle the asynchronous event in a thread-like way (The *run()* method of an instance of *AsyncEventHandler* is invoked just like a thread). However, for a non-RTSJ supportive real-time Java virtual machine, there is no such mechanism and the developer has to manage it on the application level. Since none of the Java virtual machines to be tested in this thesis follows RTSJ, in order to evaluate their capabilities of handling asynchronous events, an effective and generalized method is designed and introduced in this thesis.

As shown in Figure 6-8, there are several asynchronous event classes defined in this test, they all inherit from *java.util.EventObject*, the class *AsyncEventGenerator* take the responsibility of generating these events and dispatching them to the corresponding registered handlers. The interfaces between the *AsyncEventGenerator* and the handlers are defined by an *AsyncEventListener* Java interface. Besides implementing this interface, all asynchronous event handler classes also extend from Java *Thread* class.

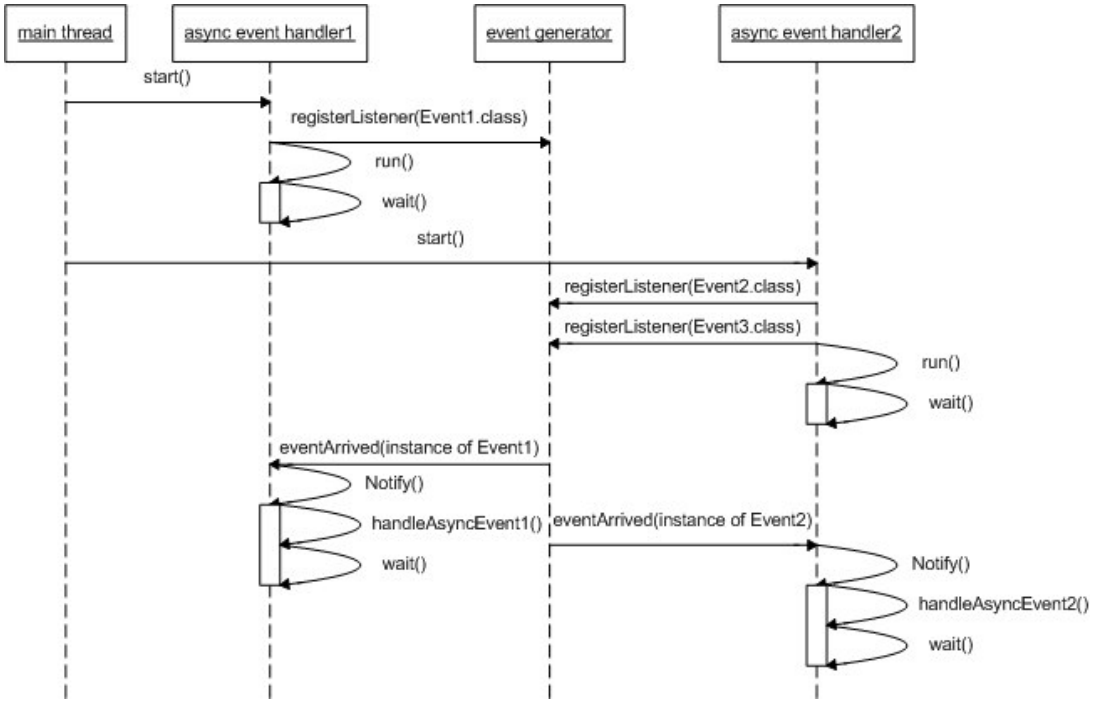


Figure 6-9 Sequence Diagram: asynchronous event handling test

By now, the logic still seems similar with the usual event handling mechanism. The difference lies in the design of the *eventArrived()* in the asynchronous event handler class. In this method, instead of creating a new thread for every new arriving event, it only uses one thread throughout the time. Every time one event arrives, this method will notify the thread it owns, handle the event and then suspend it again. Such a handler can also register more than one event so that two or more events can share the

same thread in this handler. The only drawback of this approach is that, if two events, which should be handled by the same handler, arrive one after another in too short a period of time, that handler may delay the handling of the latter one. Such weaknesses can only be fixed if an additional mechanism as defined in RTSJ is provided by the underlying Java runtime environment. The detailed workflow of this asynchronous event-handling test is described by a sequence diagram shown in Figure 6-9.

In this test, through controlling the frequency of firing different kinds of asynchronous events, the changes of event handling time will be recorded.

6.2 Sample Application Design and Implementation

The functions of the sample application have been specified in the previous chapter. In this section, the design and some implementation issues will be clarified.

6.2.1 CAN Bus Access Library and its Predictability Concern

As mentioned in earlier chapter, the sample application of this thesis will be built on the CAN bus. Therefore, the way to interact with CAN bus is a very important issue. This thesis adopts an existing Java library developed in BMW-CarIT named J2CAN, which is made to provide a set of Java API for the applications running on top of CAN. This library manipulates the raw frames received from the native CAN driver, composes them into messages and notifies the registered message handlers. In addition, J2CAN also takes the responsibility of sending CAN messages onto the bus.

The existing J2CAN library is not originally developed for the real-time applications. Therefore, the temporal behavior of it is very hard to predict. In addition, the role of this library in the sample application is the CAN message sender, receiver and dispatcher, which is the most time-critical part of work and cannot be given a low priority and ignored. Apparently, for a serious real-time application development, this kind of library should provide highly predictable behavior such as the worst-execution time of sending and receiving a CAN message and so on.

Considering the goal of developing the sample application of this thesis is to demonstrate a way to develop real-time automotive applications in Java and reveal the applicability of the chosen real-time Java solutions. This thesis chooses to build upon J2CAN library and deploy an experiment-based study on the temporal behavior of the functions in this library that are used in the sample application. The main idea of this experiment-based study is that, by deploying a large amount of black-box tests to the specific functions in the library, the worst-case execution time of these functions are retrieved by the statistical methods. These data will consequently be used to calculate the worst-case execution time in the sample application so as to apply the scheduling analysis.

A potential problem of such an approach is, when analyzing some bad or even failed cases in the final results, it could be hard to distinguish whether the fault exists in the target Java platform or it is caused by the exceptional temporal behavior of the non-real-time library. An easy way to avoid such a confusing situation is to examine the temporal behavior of the non-real-time library throughout the time, then, when the failure occurs, the causes of the failure can be located by examining the log. If

the temporal behavior of the non-real-time library doesn't exceed the worst-case execution times that are used in the scheduling analysis, the fault should then belong to the target real-time Java platform.

6.2.2 Data Structures of the Sample Application

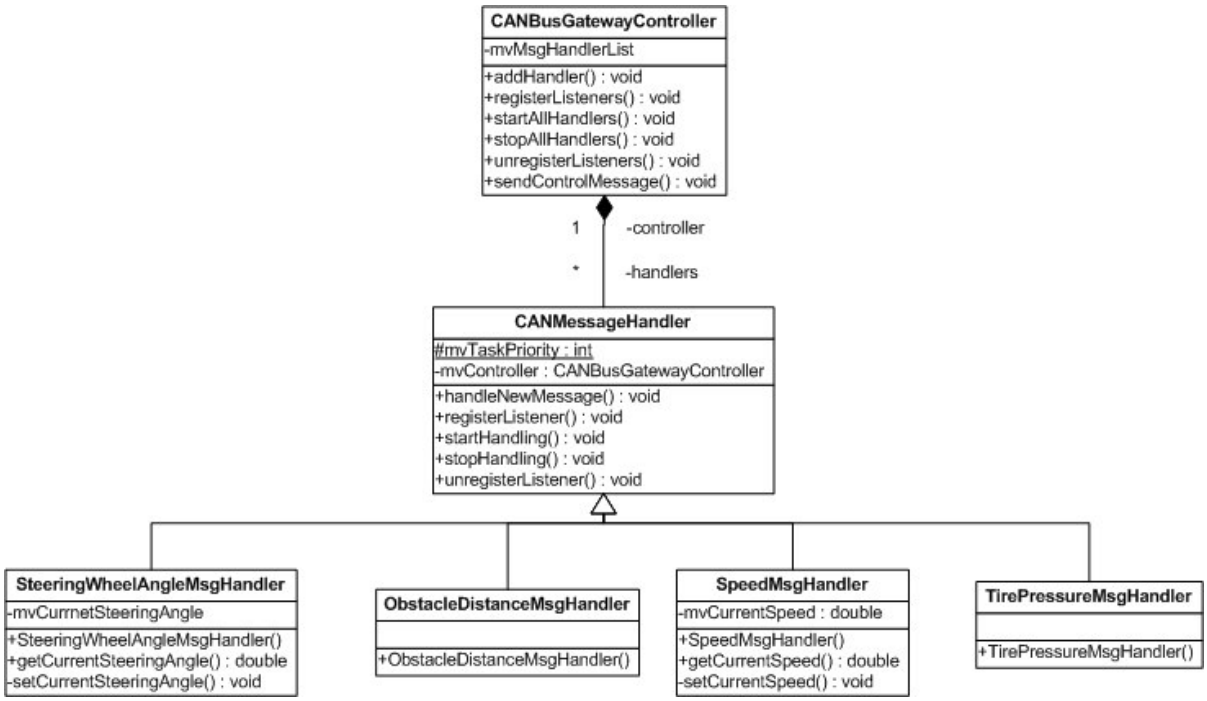


Figure 6-10 Class Diagram of the sample application

The data structure of the sample application can be shown by the class diagram in Figure 6-10. First, a *CANBusGatewayController* is defined to group and manage all the handlers, it takes the responsibility to register, remove message listeners and to start, stop all handlers. In addition, it also provides some necessary services for the handlers to use, such as sending messages on the CAN bus and so on. In order to handle the four kinds of messages as defined in 5.2, four handler classes were defined. They are, *SpeedMsgHandler*, *SteeringWheelAngleMsgHandler*, *TirePressureMsgHandler* and *ObstacleDistanceMsgHandler*. To make the design of this sample application more extensible, a *CANMessageHandler* class is abstracted from the four concrete handler classes so that the *CANBusGatewayController* class can treat all the handlers in the same way. Thus, to extend the handler set, one can simply define a handler class that inherits *CANMessageHandler* and add it to the controller class, then the controller will group it with the existing handlers and manage them together. Of course, to make sure the schedulability issue of the new task set, the necessary scheduling analysis work should be taken in advance.

6.2.3 Workflow of the Sample Application

The implementation of the sample application uses the event handling mechanism in the asynchronous event-handling test for a reference. This is because the periodic arrival of the CAN messages can be taken as the asynchronous events, and also they all satisfy that their release frequency won't exceed the capability of their handlers since none of them has a deadline larger than its period.

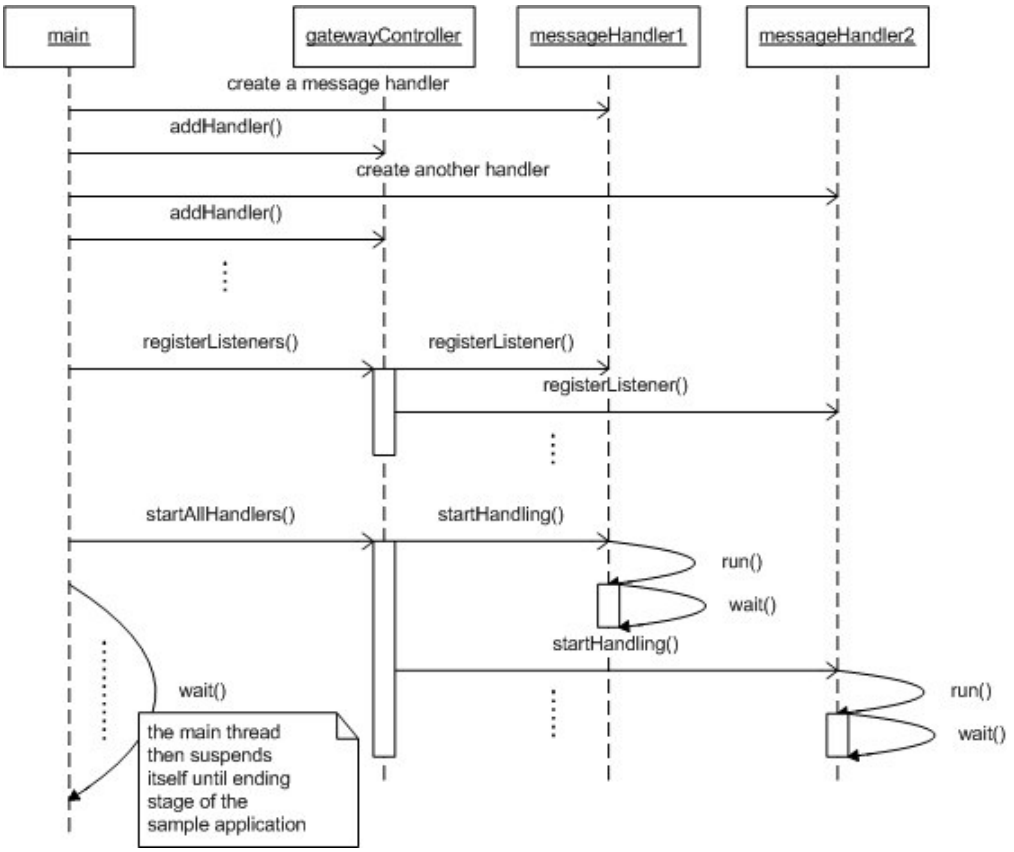


Figure 6-11 Sequence Diagram for Sample Application: start all handlers

The startup time workflow of the sample application can be shown by the sequence diagram in Figure 6-11. First, the main thread creates all the necessary handlers and add them to the instance of *CANBusGatewayController* class; in the second step, the gateway controller invokes all the *registerListener()* methods of the handlers; and then, it starts all the handlers by invoking their *startHandling()* methods. This method, for each handler, will get prepared to handle the specific CAN message by starting and suspending its thread.

After the handlers are started, they can begin to handle the CAN messages periodically. The sequence diagram in Figure 6-12 describes a typical scenario of the message handling workflow inside a handler. When the *handleNewMessage()* method in a handler is invoked by the event dispatcher (in this case, the function provided by the existing CAN access library), the handler will notify the thread

it holds and put it into the runnable state. Then the thread will contest for the CPU time according to its priority. Once the thread gets the chance to execute, it will possibly collect the necessary data for message checking, check the message content and send out the control message if necessary. After all these tasks completed, it will suspend itself again and be ready for the next period of release.

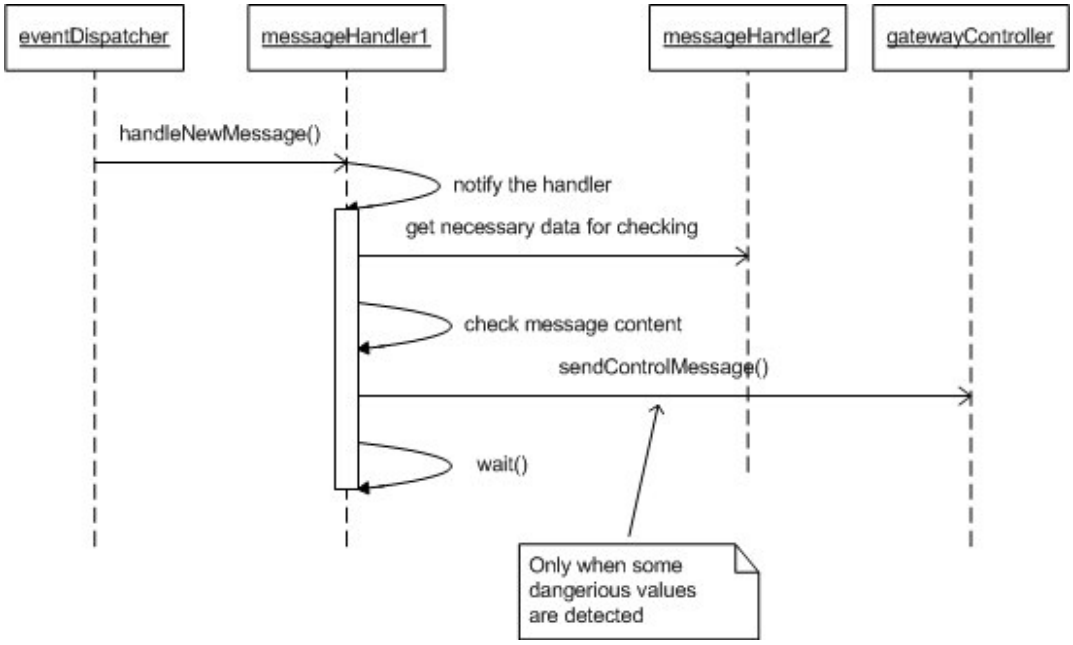


Figure 6-12 Sequence Diagram for Sample Application: handle new message

The end stage of the sample application is very similar with the startup time. The main thread awakens itself and consequently calls the *stopAllHandlers()* and *unregisterListeners()* methods in the gateway controller object, then the controller will stop the handlers one by one and unregister them from their corresponding messages.

6.3 Remote Graphic Controller for the Benchmark Application

In this thesis, to ease the test configuration and deployment work as well as the result collection and analysis work, a remote graphic controller application is designed and implemented. The main idea behind this remote graphic controller application is that, setting up a connection between the target devices (the gateway ECU) and the tester's PC and using it for transferring the test parameters to the target device at the startup time of the test and gathering the test results after the test completes. The remote graphic controller application makes use of the existing LAN connection between the target device and the tester's PC in the physical and data-link layer and sets up a TCP connection in the transport layer. In addition, the graphic package *javax.swing* are used to build the graphic user interface on the PC side and Java socket technology is used to make up the communication between two ends. One screen-shot of the remote graphic controller application on the PC side can be shown in Figure 6-13

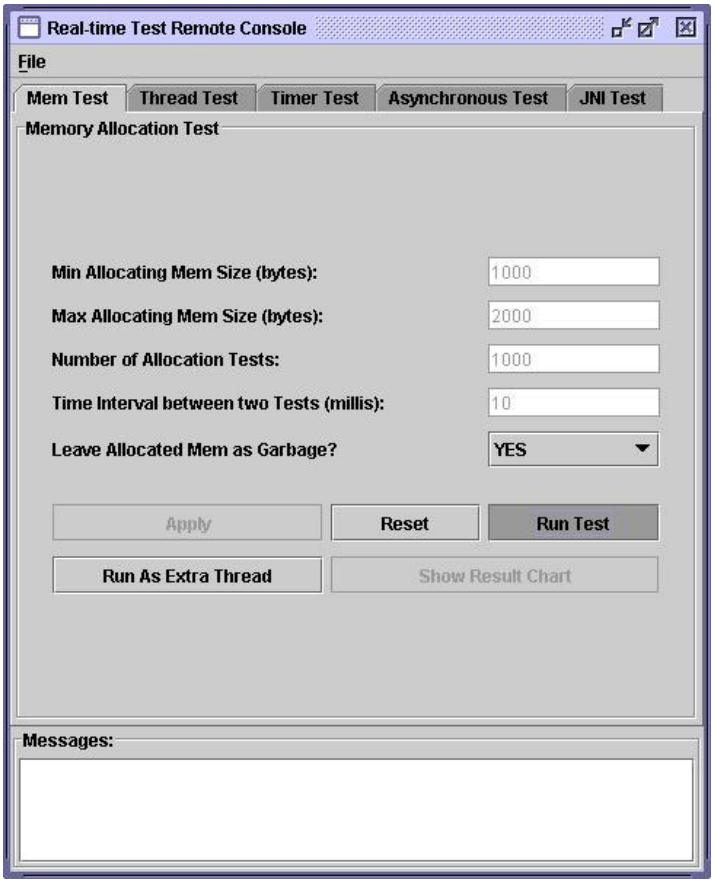


Figure 6-13 Screen-shot of the Remote Graphic Controller for the benchmark application

6.3.1 Runtime Overhead and Temporal Influence Issues

Apparently, the remote control architecture requires addition of more logic on the target device side and will consequently increase the workload of the benchmark application running on the target device. Therefore, how much these overheads influence the runtime performance and even predictability must be considered.

Let us first examine the workflow of the remote controlling test as shown in Figure 6-14. All the communication actions on the side of target device only happen either before the initialization of the test or after the results gathered. Therefore, except for the memory footprint occupied by the necessary communication objects, there is no other overhead to the runtime performance or temporal behavior of the benchmark test, and the memory footprint of the Java socket can be taken as the memory space occupied by the non-real-time tasks coexisting in the Java environment or a prerequisite utility object to load at the initial stage of the benchmark application. Thus the overhead brought by the remote control function into the benchmark application can be accepted.

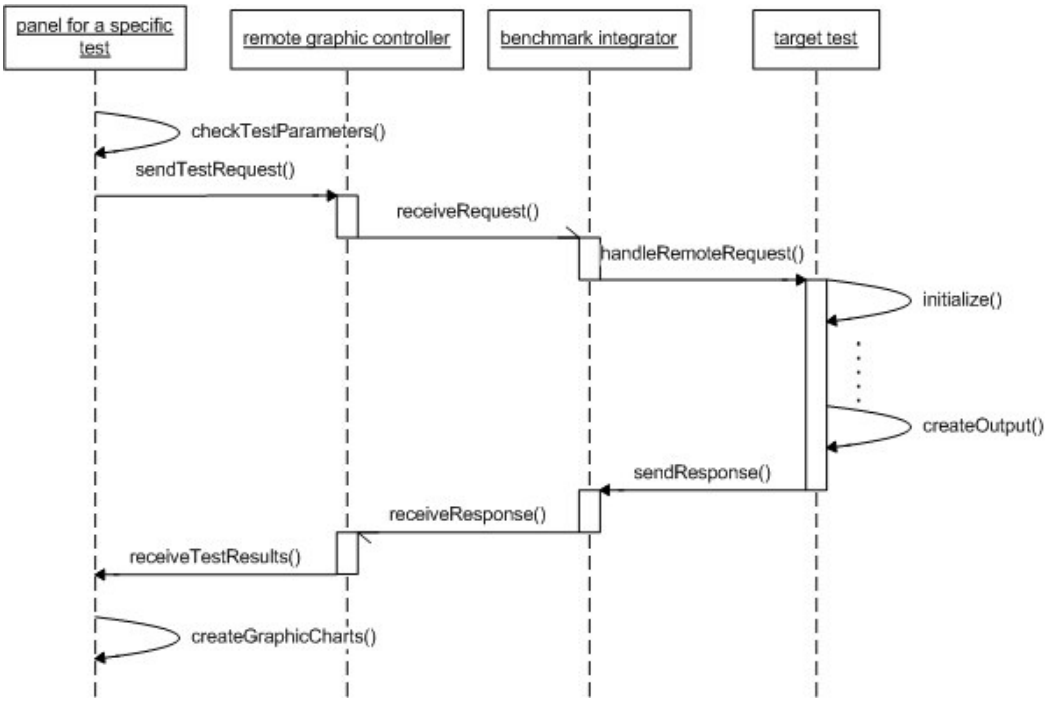


Figure 6-14 Sequence diagram: deploy benchmark test through remote control

6.3.2 Automatic Graphic Chart Generating Function

To better analyze the results of the benchmark tests, the remote graphic controller application in this thesis provides an automatic graphic chart generating function, which can generate the graphic chart immediately after the test results returned from the target device. One sample output is shown in Figure 6-15.

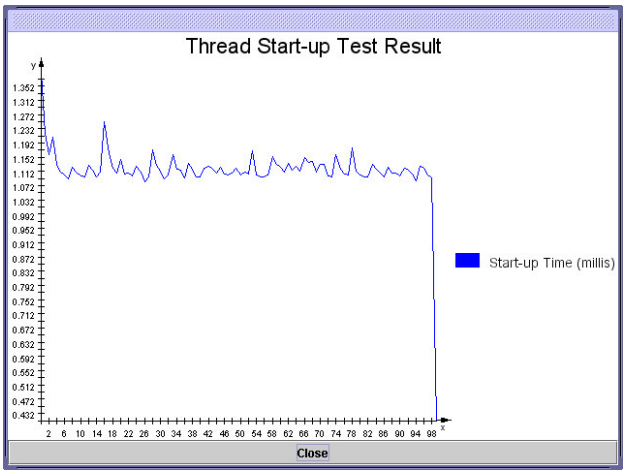


Figure 6-15 A sample chart generated by the remote graphic controller application

This function can greatly reduce the time used for the test results visualization. Furthermore, the impression directly gotten from the graphic chart can also help the tester to adjust the test parameters quickly so that a more effective and convincing test can be deployed in a short period of time.

6.4 Estimation of Implementation Workload in this Thesis

After the all the modules and applications designed and implemented in this thesis, altogether, there are more than fifty Java classes, four native C files generated. By a rough estimation, the program workload of this thesis is about eight thousand code lines.

The package structure of the Java classes implemented in this thesis can be shown in Figure 6-16.

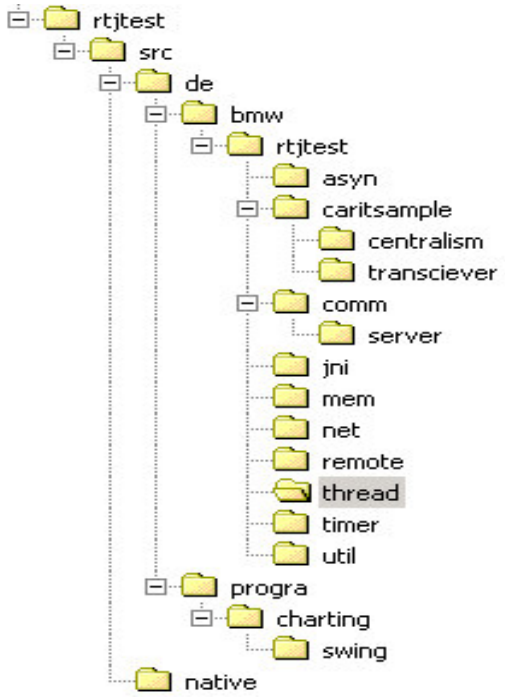


Figure 6-16 Package structure of the implementation work in this thesis

Chapter 7 Test Deployment and Result Analysis

In this chapter, the benchmark and sample application implemented in this thesis will be deployed on different real-time and non-real-time Java virtual machines on the target to compare and evaluate the real-time Java solutions. First, the test-bed environment will be described; and then the test deployment strategy follows; then in the third section of this chapter, the test cases and result analysis are brought out for both benchmark application test and sample real-time application test.

7.1 Test-bed Environment

In order to get more realistic data from the tests and make the evaluation more convincing, in this thesis, a test bed environment, which is set up with several in-car real components, will be put into use for the test. The real ECUs (Electrical Control Units), which are already adopted in the real cars, will generate the real traffic in the bus and make the most realistic environment for the tests.

In this thesis, the CAN (Controller Access Network) bus system will be chosen as the underlying communication bus for the evaluation work. The reason is because the CAN bus has the many eligible features to serve for the real-time communication as mentioned in Chapter 3. The topology of the ECUs in this environment will be set up into a centralized-gateway style as shown in Figure 7-1.

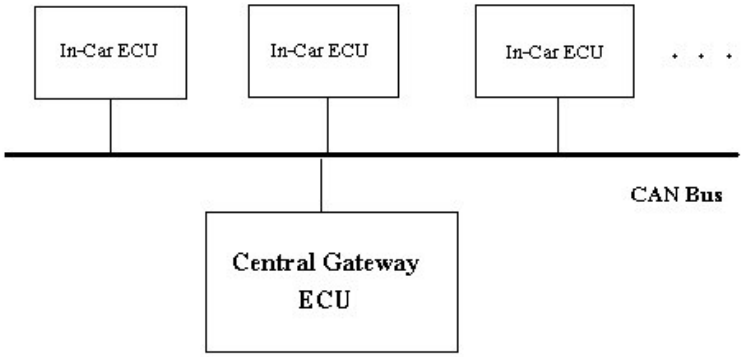


Figure 7-1 Test bed hardware environment

The central gateway ECU will be the target device used in this thesis. All the candidate Java virtual machines will be tried running on it with the benchmark applications and the automotive sample application. The hardware and software environment in the target device are listed below:

- **Target Hardware Device**

Processor: Intel Transmeta 200MHz

RAM: 128 MB

Flash ROM: 512MB

Network Interfaces:

Fast speed CAN Bus Adapter, Low speed CAN Bus Adapter, MOST Bus Adapter, Ethernet Adapter

- **Target OS**

QNX Neutrino RTOS v 6.2.1

- **Java Environments to be installed and tested on**

IBM J9 virtual machine version 2.0

PERC Virtual Machine version 4.1

- **Evaluation Applications**

Real-time Java benchmark applications

One sample automotive real-time application

- **Third-party Java Library**

A Non-real-time CAN Access Module (used in the sample application)

Test-bed Hardware Performance Consideration

The hardware platform used in this thesis has a reasonably higher performance than the typical ones adopted in the real cars in the market. The reason can be explained by the following two points: First, a better hardware platform can allow more real-time java solutions to take part into the evaluation work and consequently provide more convincing evaluation, since some solutions ask for more memory space and more powerful CPU than the existing real devices; Secondly, Moore's law indicates that the hardware performance doubles every couple of years. The exponential growth will result in increased performance and decreased cost in every domain of the computer world including the embedded systems, therefore with a higher performance hardware platform, the evaluation work will reveal the future of real-time java more prospectively, and at the same time constructively point out a way of doing further prototypes later on.

7.2 Test Deployment Strategy

There are many variations when deploying one Java application. These variations include choosing different execution modes, tuning compiler behaviors and adjusting execution options etc. This section describes the tradeoffs made on these options and the final decisions made for the tests in this thesis.

7.2.1 Alternatives of Execution Mode

Basically, there are three ways for the Java application running on the embedded devices.

- Pure Interpreting

The Java byte-code instructions of the class files are interpreted by the interpreter one by one without any performance optimization. Such way of running will cost the smallest memory footprint but cause a very low performance.

- Interpreting with Just-In-Time (JIT) compiling

The Java byte-code will be compiled into platform dependant machine code at runtime by the JIT compiler. In such a way, the execution performance can be well improved. But the compiling work at runtime also poses the impacts on the predictability of the Java application and makes its temporal behaviors difficult to predict.

- Ahead-Of-Time compiling and executing natively

All the Java byte-codes are compiled by the AOT compiler before the runtime. The output of the compilation will be a binary image file that can directly run on the target platform. In such a way, the running performance of the application can be greatly improved, and the temporal behaviors of the Java application are also more deterministic. The only drawback of this method is that, the AOT compiled binary file costs larger spaces both on the hard disk and in the memory than the previous two methods.

In practice, these three ways of execution are not exclusive to one another and can be combined together to run a single Java application. For example, you can AOT compile only some frequently used classes in the application and leave other classes to be dynamically loaded and interpreted (with or without JIT compiling) at runtime.

For the two chosen Java virtual machines in this thesis, their support for the above three execution options can be summarized in the following table:

JVM	Execution Option		
	Interpreting	**JIT compilation**	**AOT compilation**
IBM J9	Yes	Configurable	Yes
PERC VM	Yes	Configurable	Yes

Table 7-1 Execution Options of the chosen JVMs

Considering the runtime performance, predictability and evaluation fairness issues, in this thesis, we will use the AOT compilation in all the tests, including the benchmark test and the sample application test.

7.2.2 AOT Compilation Settings and Execution Options

There are many options to do the AOT compilation to a target Java package. The most accurate AOT compilation allows the user to decide whether every single method in the java classes needs to be AOT compiled or not. Both J9 and PERC virtual machines provide such options. However, to make the decision on which class or methods to be AOT compiled is an elaborate work and tends to be very time-consuming. Relatively, an easier approach will be pre-compiling all the classes needed by the

target application. This approach also excludes the possibilities that loading un-compiled classes may induce unpredictable temporal behavior to the application at runtime.

Another important issue when doing the compilation is that, how to treat the native JNI libraries used by the target application. There are two ways to do this. One is to statically link the native libraries into the image file; the other one is to dynamically load the libraries at runtime. PERC virtual machine supports both of the two options, but J9 only supports the second option, which is the dynamic loading approach.

Considering the predictability of the application at runtime, the first approach seems to excel the second one. But, on the other hand, these JNI libraries normally only take the responsibility of linking Java application to the other native libraries such as device driver or system libraries. Apparently, those libraries cannot be statically compiled into the final image anyway. Therefore, the overhead of dynamic loading native libraries cannot be avoided. Additionally, such loading overhead normally takes place when the first time virtual machine loads the class that declares to use JNI. To avoid the influence of such overhead, one application can just explicitly load these classes in the startup stage for a warming up, thus for the whole running period, the system doesn't need to concern it any more. So, in this thesis, in order to unify the settings for the two test virtual machines, the dynamic loading approach is adopted.

Since all the classes that may be used at runtime are AOT compiled, the execution options for both of the Java platforms are highly simplified. Neither *classpath* nor *jit* needs to be specified. In both cases, the tester only needs to provide the JNI library path and then load the single compiled image file.

7.3 Test Results and Analysis

In this section, both the prepared benchmark applications and sample real-time application will be deployed. All the test results will be recorded and analyzed, especially, the output of the benchmark application will be visualized by the remote graphic controller application developed in this thesis.

7.3.1 Benchmark Application Test

7.3.1.1 Memory Allocation Time Test:

There are altogether four different test cases picked into this thesis. The detailed test cases and results can be found in Appendix B 1, Appendix B 2, Appendix B 3 and Appendix B 4.

Result analysis:

The first test (Appendix B 1) shows that, when the benchmark application starts the memory allocation test for the first time, the memory allocation time will suffer a long delay up to several milliseconds. This may be caused by the following reason: since the AOT compiled image files are several megabytes in size, the operating system may refuse to load it all into the RAM at one time when it starts. Instead, the operating system may only load a part of the binary file and for the rest of the file, the system will postpone the loading until that part of the code is explicitly called.

Such unpredictability caused by memory and process manager of the operating system can be minimized by taking some warming-up operations at the initialization time of the application. In such a way, the compiled code that is needed in the application will be preloaded into the memory before the actual work starts.

The second test (Appendix B 2) shows the situation after the startup stage of the memory allocation test. The result got from PERC shows that the average allocation time is about 0.09 ms. During the whole period of test, there is one peak value of allocation time going up to one millisecond. From the memory status graphic, we can see that the garbage created during the test is successfully collected so that the free memories in the heap are steadily kept at a high level. Whereas the result got from J9 shows that the average allocation time is about 0.03 ms, which is more satisfactory than PERC, but the memory status indicates that the garbage is not collected in time and the free memories in the heap decline rapidly. Such circumstance will consequently oblige the virtual machine to invoke one garbage collection operation, where in J9 it could be very slow. In the third test, we will show the impact in such a situation.

In the third test (Appendix B 3), a more intensive memory allocation test is deployed. This time, the result got from PERC shows that, the memory allocation time isn't affected by the heavy allocation workload; the worst execution time is still lower than one millisecond. As for J9, the weakness of the unpredictable garbage collector is exposed, though the average allocation time is still far below one millisecond, but being influenced by the garbage collection work, there are several peak values appearing which are close to or even above ten milliseconds. The difference of garbage collection between PERC virtual machine and J9 can also be shown by the memory status charts of them. The compact saw-toothed shape in the PERC chart explicitly shows the contribution of its incremental garbage collector, whereas the severe free memory changes in J9 chart are clearly caused the unpredictable garbage collector of it.

In the last memory allocation test (Appendix B 4), we tried to simulate the memory allocation situation in the sample real-time application. Though the simulation is fairly rough, it can still help us find out what the memory allocation situation in the sample application would look like. Through the results got from PERC and J9, we can once again see the difference between the two kinds of garbage collection mechanisms, the non-real-time one of J9 and the real-time one of PERC. In this test, the worst-case allocation time in PERC is about 0.2 ms and this value in J9 is more than one ms.

Summary:

To summarize the results got from this memory allocation time test, we can say that, though IBM J9 wins the average allocation time in most of the tests, the incremental real-time garbage collector in PERC virtual machine helps it achieve a much better worst-case allocation time than J9, which is more important to the real-time application development.

7.3.1.2 Thread and Synchronization Test

- Thread Startup Time Test

The test parameters, test description and test results of this test can be found at Appendix B 5.

Result Analysis:

First of all, the unsteady startup stage problem happening in the memory allocation time test is recurring in this test. This once again proves that such kind of problems are caused by executable file loading strategy of the operating system because by the time this test was deployed the whole benchmark application had started for some time and several memory allocation time tests had been deployed, but the startup stage of this test still cause the instability.

In the second test after the startup stage both of the test results got from PERC and J9 tend to be more stable then. The PERC result shows that, the average thread startup time is about 1.1 ms and the worst-case value is about 1.6 ms, which is possibly caused by the incremental garbage collection work. As for J9, the result shows that, the average thread startup time is about 1.2 ms, and there is no evident peak value appeared in this test. But when we prolong the test period and deploy it again, the long period garbage collection work of J9 begins to show up (shown in the last chart of Appendix B 5). It causes the worst-case execution time run up to more than seventy milliseconds. Therefore, we can say that, in this test, the impact of garbage collection still dominates the worst-case startup time.

- Thread Yielding Context-switch Time Test

The test parameters, test description and test results of this test can be found at Appendix B 6.

Result Analysis:

From the results of this test, we can see that, such yielding context-switch in PERC virtual machine is very instable and may be interfered by the garbage collector and induce a long delay even up to twenty milliseconds. As for J9, the result is quite satisfactory and stable.

- Thread Notification Context-switch Time Test

The test parameters, test description and test results of this test can be found at Appendix B 7.

Result Analysis:

In this test, the result of PERC shows that, such context-switch caused by one thread notifying another is efficiently supported. The worst-case result in 100, 000 times of switching is kept below 2.3 ms. While, the result from J9 looks a little confusing because it shows that, very few context-switch happened during the test, which makes the final result data meaningless. After further survey on the thread scheduling strategy of J9, we find that in the J9 virtual machine, the thread dispatcher does not strictly take the thread priority as the meaning of execution eligibility, instead it may choose any thread from the queue of runnable threads to optimize the overall runtime performance no matter what that thread's priority is. With such mechanism, we can say that J9 does not provide the strict priority-based environment which the real-time system demands. This fact will be reminded and proved again in the later priority-inversion test.

Thread Entering Unlocked Monitor Time Test

The test parameters, test description and test results of this test can be found at Appendix B 8.

Result Analysis:

This test shows that J9 beats PERC on both average time and worst-case execution time. But reminding that, the garbage collection operation of PERC runs more frequently than J9, the default GC period is about 250 ms as for J9 this period is highly dependent on the left free memories in the heap. Apparently, the result we got from J9 doesn't experience the garbage collection work for the low memory cost and short period of runtime. Therefore, the worst-case result of PERC is more close to the real value.

- Thread Priority Inversion Test

The test parameters, test description and test results of this test can be found at Appendix B 9.

Result Analysis:

In this test, we set the medial priority thread number to 5 so that the execution time difference between the low priority thread and high priority thread can be distinguished better. One thing to mention is that, all the medium priority threads have the same priority. Therefore, their sequence of completion can be neglected.

The test result from PERC virtual machine shows that the priority inversion problem is effectively solved. The high priority thread finishes first and the low priority thread is kept in the end. Such tests have been repeated many times in practice and all the results show the same effect as this one.

The test results from J9 tend to be various. Three kinds of situations are attached in the appendix, the first one shows that the low priority thread ends first and the high priority thread is delayed to finish after four of the medial priority threads, which means priority inversion occurs; the second one shows, the low priority thread still finishes first and then followed the high priority thread, all the medial threads finish after them; the last result chart shows that priority inversion is avoided this time that high priority thread finishes first and low priority thread is left in the end. The variations of the results again prove that the thread scheduling mechanism in the J9 virtual machine does not strictly follow the priority-based scheduling. That is to say, a thread with a highest priority does not always get the first execution eligibility. This violates the policy of priority-based real-time system so that none of the priority-based scheduling theory can be applied here.

- Summary

The thread and synchronization related tests in this thesis show that the difference of garbage collection mechanisms between PERC and J9 also makes the conclusive effects to some of the tests here, such as thread startup time test, thread yielding context switch time test and thread entering unlocked monitor test. The situation in PERC shows that, the frequent incremental garbage collection work makes the average results slower, but helps to achieve a better worst-case execution time. As for J9, although the average results are quite satisfactory, the worst-case results of each of these tests are all made much higher than PERC by the unpredictable garbage collector.

In the thread notification context-switch time test and priority inversion test, we find that PERC strictly follows the priority-based scheduling policy and provides priority inheritance mechanism, which successfully avoids the priority inversion problem. As for J9, it does not completely obey the priority-based scheduling and makes other approaches to pursue a better average-case performance.

This is why the notification context-switch test failed in J9 and in the priority inversion test J9 gets different results from time to time.

7.3.1.3 Timer Test

The test parameters, test description and test results of this test can be found at Appendix B 10 and Appendix B 11, respectively presenting the one-shot timer test and the periodic timer test.

Result Analysis:

As we can see from the results in the appendix, for both the one-shot timer test and periodic timer test, the PERC virtual machine shows an instable performance caused by its garbage collector. The peak timer release offset can be four to six milliseconds and happens frequently corresponding to the GC period value. On the contrary, J9 shows much better results in these two tests. The release offsets of the timer are controlled within one millisecond (without interference of the garbage collector).

One thing to mention is that the timer tests in the benchmark make use of the timer class in Java standard library, which is *java.util.Timer*. The special timer provided by PERC in its extra library is not applied because its specialties contradict with the generality pursued by this benchmark application. A particular application, which is fixed to run in the PERC virtual machine, can consider adopting such an approach.

7.3.1.4 Asynchronous Event Handling Test

There are two asynchronous event-handling test cases adopted and attached in this thesis. Their parameters, test descriptions and test results can be found at Appendix B 12 and Appendix B 13.

Result Analysis:

Two asynchronous event-handling tests respectively represent the situation of one handler and the situation of multiple handlers. The memory allocation task and calculation task of one handler is kept the same in two tests. In addition, one low priority thread is inserted into the second test to add more workload, and in each test, a symbolic deadline is set as a reference line in order to show the different results got from two virtual machines more clearly.

The results of PERC in two tests show that, the response times of the asynchronous event are quite steady and kept much lower than the predefined deadline. As for the results got from J9, the response times seem to be surprisingly long and in both of the two tests, there exists missed deadlines, which never happens in PERC.

As mentioned in the previous chapter, the asynchronous event-handling test intends to show an overall performance of how the runtime environment handles the asynchronous events. The great disparity shown in the results can be explained by the following two reasons:

- The thread notification context-switch technique used in the asynchronous event-handling test is not well supported by the J9 virtual machine for the reason we analyzed before.
- The intensive memory allocating operations make the garbage collection work in J9 frequently invoked and therefore cause many peak values that exceed the predefined deadline.

7.3.1.5 JNI Access Time Test

There are two JNI access time test cases attached in this thesis. Their parameters, test descriptions and test results of this test can be found at Appendix B 14 and Appendix B 15.

Result Analysis:

The first test mainly intends to show the performances of retrieving primitive data from JNI for both virtual machines. From the results chart, we can see that the average access times of getting Java byte, Java int and Java double are all faster than the values in PERC. In addition, the last two charts attached in this test show that, the JNI access time of PERC can be delayed by the garbage collection work and get a worst-case value up to more than twenty milliseconds. Although, theoretically, J9 may have such problem too, after a large amount of tests with different kinds of parameters, no such situation appears eventually.

The second test aims to show the performances of two virtual machines on retrieving data arrays from JNI by several means. The test results shows that, J9 has slight advantages over PERC on the average access time in all cases, including the ways of directly getting array as return value, JNI call back and sending member array to JNI and bringing the values back. As for the worst-case values, PERC still has unstable peak values for each way of retrieving the arrays. The worst-case JNI access time in these cases is about 23 ms.

Summary:

In the JNI access time tests, we find that J9 has a satisfactory and stable performance in this field. For most of the tests, J9 wins on both average access time and worst-case access time. As for PERC, it is found to have trouble with preventing garbage collection interfering JNI access. Therefore, the JNI access in PERC is more unpredictable than J9 for such reasons. Furthermore, PERC real-time Java solution also provides an individual native interface solution for Java called PNI. It is not included in the benchmark application for the compatibility issue.

7.3.2 Sample Application Test

For the sample application test, we will first deploy a scheduling analysis work to examine the possible runtime behavior of all the tasks and verify their schedulability. After that, we will refer to some of the results we got after running the sample application to analyze the contribution and limitation of the scheduling work.

Before applying the scheduling analysis to the sample application, we should first check the suitability of our target Java platforms: J9 and PERC virtual machine.

For IBM J9, a fatal problem, which makes the real-time scheduling work impossible to deploy, is that, it does not strictly follow the priority-based scheduling policy, which is one of the prerequisites for the fixed priority preemptive scheduling theory. Apparently, such violation will make all the theorems and formulas we introduced in the previous chapter become inapplicable. Therefore, we cannot apply these theories to the sample application when it is running on the J9 virtual machine.

As for PERC virtual machine, both the theoretical study and the result of the benchmark tests have proved that, it is a strictly priority-based preemptive system and it supports the priority inheritance protocol to avoid unbound priority inversion problem. Therefore, the PERC virtual machine has the basic attributes to apply the fixed priority preemptive scheduling theory.

Now we will try to do the scheduling analysis work step by step based on the benchmark results of PERC virtual machine.

7.3.2.1 Scheduling Analysis on Sample Real-time Application

Scheduling theory:

Fixed-priority preemptive scheduling

Scheduling algorithm:

Deadline Monotonic Scheduling

Applicable formula:

The response time of each task $R_i = J_i + W_i^n$, where W_i^n is the converging value of a series of number conducted by the following formula:

$$W_i^{n+1} = B_i + C_i + C_{sw} + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n + J_i}{T_j} \right\rceil (C_j + C_{sw})$$

and J_i is the release jitter time for that task.

(For more information, one can refer to section 5.3 in this thesis)

Necessary parameters and their values:

The necessary parameters for applying the above formula can be shown in Table 7-2:

Parameters	**All Tasks Running in the System**				
	CAN Message Dispatcher ($Task_0$)	**Obstacle distance msg handler** ($Task_1$)	**Speed msg handler** ($Task_2$)	**Steering angle msg handler** ($Task_3$)	**Tire pressure msg handler** ($Task_4$)
T	100 ms	100 ms	100 ms	200 ms	500 ms
D	No	70 ms	100 ms	150 ms	200 ms
J	No	4.0 ms	4.0 ms	4.0 ms	4.0 ms
B	0	6.0 ms	0 ms	0 ms	0 ms
C_{sw}	0.5 ms	0.5 ms	0.5 ms	0.5 ms	0.5 ms

C	16.0 ms	37.0 ms	2.0 ms	36.0 ms	36.0 ms
P	$P_{\max}$	$P_{\max}-1$	$P_{\max}-2$	$P_{\max}-3$	$P_{\max}-4$

Table 7-2 Necessary parameters and their values for sample application scheduling analysis

Explanation for some of the values in Table 7-2:

1) CAN message dispatcher with its period *T* , execution time *C* and priority *P*:

   The CAN message dispatcher thread resides in the CAN access library used by the sample application. Its responsibility is to receive the CAN messages from the bus and notify their handlers. The reason why it becomes a task is because the time from one message arriving to its handler being notified cannot be neglected. Through a large amount of tests, we experimentally get the worst-case execution time of this period of time, which is 4.0 ms in value. Therefore, this message dispatcher should be taken as another real-time task and get involved into the scheduling analysis. For the special and important role it plays, its priority is set to the highest value among all the tasks. Its period can be determined by the shortest period of all the handlers, which is 100 ms. As for the worst-case execution time, it is easy to prove that, in any specific 100 ms period at runtime, there are at most four messages arriving and being dispatched by the dispatcher. Therefore, the execution of this message dispatcher could not exceed four time of worst-case dispatching time: 4.0 ms, 16.0 ms is then got.

2) Release jitter time of the message handlers:

   Since the message dispatcher has 4.0 ms of worst-case dispatching time and from the experiments it also shows that the best-case dispatching time is only 0.1 ms long. Therefore, in a handler's view, the biggest release jitter is less than 4.0 ms.

3) Maximum blocking time:

   From the priority inversion test on PERC, we have seen that the priority inversion problem is avoided by the priority inheritance protocol. Therefore, according to Theorem 2-6 introduced in chapter two, the obstacle distance message handler can at most be blocked twice by the lower priority handlers. Furthermore, from the results of priority inversion test, we roughly take the maximum execution time of the high priority thread in all the tests as the worst-case blocking time in the sample application. In such a way, the maximum blocking time of obstacle distance message handler is calculated out, which is 6.0ms in value.

4) Context-switch factor C_{sw} :

   Though the thread yielding context-switch time test shows that, PERC has a very large worst-case switching time for this type of context-switch, considering that such context-switch is never used in the sample application, we just take the notification context-switch time as a reference. Through picking the maximum notification context-switch time from all the tests, we set C_{sw} 0.5ms as its value.

5) Worst-case execution time *C* of all the handlers:

Since in the sample application, the workflow in each handler is kept rather simple, the only time-consuming part of the handling task is sending the control message to the bus. In each handler, this part of work is accomplished by calling an API in the CAN access library. Therefore, a set of black box tests is deployed to measure the worst-case execution time of sending message to CAN. After many tests are deployed, the maximum value is recorded, which is 35.0 ms (the peak value caused by the system loading delay is excluded here).

CPU utilization check:

$$U = \sum_{i=1}^{5} U_i = \sum_{i=1}^{5} \frac{C_i}{T_i} = \frac{16}{100} + \frac{37}{100} + \frac{2}{100} + \frac{36}{200} + \frac{36}{500} = 0.16 + 0.37 + 0.02 + 0.18 + 0.072 = 0.802 < 1$$

Since 0.802 > 0.69, we must do an exact completion time analysis on this task set.

Exact completion time analysis:

CAN Message Dispatcher:

For the CAN message dispatcher: $Task_0$, it is a virtual task that we make to reflect the necessary CPU cost by the CAN message dispatching work, and there is no explicit deadline for it. But apparently, its response time should not exceed its period, which is 100ms:

$$W_0^0 = 0;$$

$$W_0^1 = B_0 + C_0 + C_{sw} = 16.5;$$

$$W_0^2 = B_0 + C_0 + C_{sw} = 16.5 = W_0^1;$$

$$R_0 = J_0 + W_0^1 = 16.5 < D_0 = 100$$

So, the message dispatcher task is schedulable;

Obstacle Distance Message Handler ($Task_1$):

$$W_1^0 = 0;$$

$$W_1^1 = B_1 + C_1 + C_{sw} + \sum_{j \in hp(1)} \left\lceil \frac{W_1^0 + J_1}{T_j} \right\rceil (C_j + C_{sw}) = 6 + 37 + 0.5 + 1 \times (16 + 0.5) = 60;$$

$$W_1^2 = B_1 + C_1 + C_{sw} + \sum_{j \in hp(1)} \left\lceil \frac{W_1^1 + J_1}{T_j} \right\rceil (C_j + C_{sw}) = 6 + 37 + 0.5 + 1 \times (16 + 0.5) = 60 = W_1^1;$$

$$R_1 = J_1 + W_1^2 = 64 < D_1 = 70$$

Therefore, the obstacle distance message handler is also schedulable.

Speed Message Handler ($Task_2$):

$$W_2^0 = 0;$$

$$W_2^1 = B_2 + C_2 + C_{sw} + \sum_{j \in hp(2)} \left\lceil \frac{W_2^0 + J_2}{T_j} \right\rceil (C_j + C_{sw}) = 0 + 2 + 0.5 + 1 \times (16 + 0.5) + 1 \times (37 + 0.5)$$

$$= 56.5;$$

$$W_2^2 = B_2 + C_2 + C_{sw} + \sum_{j \in hp(2)} \left\lceil \frac{W_2^1 + J_2}{T_j} \right\rceil (C_j + C_{sw}) = 0 + 2 + 0.5 + 1 \times (16 + 0.5) + 1 \times (37 + 0.5)$$

$$= 56.5 = W_2^1;$$

$$R_2 = J_2 + W_2^2 = 4 + 56.5 = 60.5 < D_2 = 100$$

Therefore, the speed message handler is also schedulable.

Steering Angle Message Handler ($Task_3$):

$$W_3^0 = 0;$$

$$W_3^1 = B_3 + C_3 + C_{sw} + \sum_{j \in hp(3)} \left\lceil \frac{W_3^0 + J_3}{T_j} \right\rceil (C_j + C_{sw})$$

$$= 0 + 36 + 0.5 + 1 \times (16 + 0.5) + 1 \times (37 + 0.5) + 1 \times (2 + 0.5) = 93;$$

$$W_3^2 = B_3 + C_3 + C_{sw} + \sum_{j \in hp(3)} \left\lceil \frac{W_3^1 + J_3}{T_j} \right\rceil (C_j + C_{sw}) = 0 + 36 + 0.5 + \sum_{j \in hp(3)} \left\lceil \frac{93 + 4}{T_j} \right\rceil (C_j + C_{sw})$$

$$= 0 + 36 + 0.5 + 1 \times (16 + 0.5) + 1 \times (37 + 0.5) + 1 \times (2 + 0.5) = 93 = W_3^1;$$

$$R_3 = J_3 + W_3^2 = 4 + 93 = 97 < D_3 = 150$$

Therefore, the steering angle message handler is also schedulable.

Tire Pressure Message Handler($Task_4$):

$$W_4^0 = 0;$$

$$W_4^1 = B_4 + C_4 + C_{sw} + \sum_{j \in hp(4)} \left\lceil \frac{W_4^0 + J_4}{T_j} \right\rceil (C_j + C_{sw})$$

$$= 0 + 36 + 0.5 + 1 \times (16 + 0.5) + 1 \times (37 + 0.5) + 1 \times (2 + 0.5) + 1 \times (36 + 0.5) = 129.5;$$

$$W_4^2 = B_4 + C_4 + C_{sw} + \sum_{j \in hp(4)} \left\lceil \frac{W_4^1 + J_4}{T_j} \right\rceil (C_j + C_{sw}) = 0 + 36 + 0.5 + \sum_{j \in hp(4)} \left\lceil \frac{129.5 + 4}{T_j} \right\rceil (C_j + C_{sw})$$

$$= 0 + 36 + 0.5 + 2 \times (16 + 0.5) + 2 \times (37 + 0.5) + 2 \times (2 + 0.5) + 1 \times (36 + 0.5) = 186;$$

$$W_4^3 = B_4 + C_4 + C_{sw} + \sum_{j \in hp(4)} \left\lceil \frac{W_4^2 + J_4}{T_j} \right\rceil (C_j + C_{sw}) = 0 + 36 + 0.5 + \sum_{j \in hp(4)} \left\lceil \frac{186 + 4}{T_j} \right\rceil (C_j + C_{sw})$$

$$= 0 + 36 + 0.5 + 2 \times (16 + 0.5) + 2 \times (37 + 0.5) + 2 \times (2 + 0.5) + 1 \times (36 + 0.5) = 186 = W_4^2;$$

$$R_4 = J_4 + W_4^3 = 4 + 186 = 190 < D_4 = 200$$

Therefore, the tire pressure message handler is also schedulable.

Since all the tasks in the task set of sample application are schedulable, the whole application is proved schedulable too. The scheduling analysis here also shows that the most critical task, whose worst-case response time is closest to its deadline, is the obstacle distance message handler.

7.3.2.2 Test Results and their Analysis

Here are some results got from the sample application tests on the target (the system startup loading overhead has been avoided by running some warm-up operations first):

After running the task for hours, the results can be shown in Figure 7-2:

- Result from PERC:

Obstacle Distance Message Handler:
Max value is: 33.20
Min value is: 0.60
Average value is: 3.35
Speed Message Handler:
Max value is: 9.50
Min value is: 0.50
Average value is: 1.13
Steering Angle Message Handler:
Max value is: 5.70
Min value is: 0.62
Average value is: 1.32
Tire Pressure Message Handler:
Max value is: 32.20
Min value is: 0.57
Average value is: 3.58

- Result from J9:

Obstacle Distance Message Handler:
Max value is: 48.77
Min value is: 0.31
Average value is: 1.84
Speed Message Handler:
Max value is: 5.75
Min value is: 0.24
Average value is: 0.59
Steering Angle Message Handler:
Max value is: 46.70
Min value is: 0.29
Average value is: 2.73
Tire Pressure Message Handler:
Max value is: 30.17
Min value is: 0.53
Average value is: 2.18

Figure 7-2 Test results for the sample application

Both of the result sets show that, the sample application keeps running without missing any task's deadline during the test period. However, the result we got from J9 cannot be verified by the scheduling analysis. All we can do to verify this result is to deploy large amount of tests and examine the results using statistical methods, so that from a probability point of view the real-time performance of such approach can be reflected.

From the result we got from PERC, we can see that, the worst-case response times of all tasks are much shorter than the value we got from the scheduling analysis, this may be caused by the following reason:

For all handlers except speed message handler, the worst-case execution time of a task only happens when they meet some critical situations. Though, through the program, we set the probabilities of such situations relatively much higher than they are in reality. Still, such frequent variation of the task execution time makes the probabilities of those worst cases very small.

The CAN access library adopted by the sample application shows great unpredictability. The best-case sending and receiving message time is much shorter than the worst case. Therefore, using the worst-case sending and receiving time in the scheduling analysis makes the final results too pessimistic comparing the real data.

7.3.2.3 Contribution and Limitation of the Scheduling Analysis Work

The most important contribution of the scheduling analysis work is that it points out a way of examining and verifying the schedulability of a real-time application. In addition, the feedback of the scheduling analysis can also help the developer realize the most critical part of the application and conduct some constructive improvements to the program.

The limitation of the scheduling analysis work on the sample application can be summarized as follows:

- The values of the necessary parameters are given by the results got from the benchmark application tests and other black box tests on the CAN access library. Due to the lack of theoretical support of such approach and the limited testing period in this thesis, the final data used in the scheduling analysis may be not exact enough, which can make the result of scheduling analysis deviates in some degree.
- The incremental garbage collection work in PERC virtual machine is not included in the scheduling analysis. Though the PERC virtual machine allows the developer to change the priority of the garbage collector thread into the lowest value in the system, the garbage collection work may still interfere the schedulability of the whole task set for its uninterrupted memory copying operation issue. This is also one of the reasons why the PERC virtual machine is not suitable for building the hard real-time system.

Chapter 8 Conclusion and Future Prospects

In this final chapter, we will summarize the theoretical and experimental results we got in this thesis and draw a conclusion. Then, some limitations of this thesis and the future prospects in this area are discussed.

8.1 Summary of Results

From the survey on the theories in the real-time system design domain, we know that, to build a predictable and steady real-time application with theoretical guarantee, a scheduling analysis work must be accomplished to guide the development work. The most appropriate scheduling technique to be applied in the Java language is the fixed priority preemptive scheduling theories. When doing the fixed-priority preemptive scheduling analysis, some necessary attributes of the underlying real-time platform, such as thread context switch overhead, priority inversion avoidance mechanism etc, should be known in advance. Normally, such knowledge can be obtained in the following three ways: product specification, specific vender-provided analysis tools, and individual test applications. Among the three ways, results got from the first two are more efficient and more reliable; as for the results got by means of individual test applications, they mainly rely on the large amount of experiments and probability study and may not be as accurate and exact as the results got by the first two methods. However, from the theoretical analysis on the available real-time Java solutions, we can see that, none of them provides the first two kinds of supports. Their approaches are mainly applicable to build the soft real-time applications based on empirical and experimental knowledge.

Although it is difficult to apply the scheduling analysis work on the available real-time Java products, this thesis still makes a lot of effort to try this method out in order to illustrate a correct way of doing hard real-time automotive applications in Java.

In the implementation part of this thesis a set of benchmark applications are designed to test and get the necessary parameters. These parameters are summarized by the preliminary scheduling analysis on the functions of a sample real-time automotive application. The functions of this sample application are abstracted from the common behaviors of the real-time automotive applications running on the gateway ECUs. The design of the sample application also uses an extensible pattern so that it can be reused by the further evaluation work. Some additional tests are also added to the benchmark applications to reveal the overall real-time capability of a specific Java platform.

The evaluation work of this thesis puts two embedded Java environment into test. One is IBM J9, representing the most mature and steady non-real-time Java virtual machine in the embedded system domain; the other one is the PERC virtual machine, which is one of the most influential real-time Java solutions today. The test results got from the benchmark application show that, IBM J9 wins in the average performance of memory allocation and some of the overall real-time capability test, including timer precision test and JNI access time test. While on the other hand, the PERC virtual machine shows the advantages in worst-case memory allocation time (especially under an intensive

memory allocation scenario), asynchronous event handling test, thread context switch test and the priority inversion avoidance test.

Particularly, the two-space copying real-time garbage collection mechanism provided by PERC virtual machine seems still to cause some unpredictability during the tests. This may be since that for a particular frequency of one periodic task, the PERC virtual machine allows developers to adjust the virtual machine parameters, such as the GC period, to avoid the conflicts between garbage collection work and memory allocation operations in the task. However, for the benchmark test of this thesis, this approach is difficult to apply because the frequency of each test is frequently adjusted to try more situations and it will bring much complexity to adjust the virtual machine parameters for every single test. Therefore, considering the above issue, for a specific real-time application, the real-time performance of the PERC virtual machine may be better than it shows in the benchmark test in some degree.

For the sample real-time application, we deployed a scheduling analysis for its schedulability running on PERC virtual machine based on the benchmark test results. The result of this scheduling analysis work proves that, the sample application is schedulable on the PERC virtual machine. Since some of J9's attributes violate the priority-based scheduling policy, we cannot do the same work on J9 to verify its results. Finally, the sample application test results shows that, all the chosen CAN messages are handled successfully before the predefined deadlines in both of the Java platforms. The contribution and limitation of such approaches are then analyzed respectively.

In summary, this thesis successfully accomplishes building a sample real-time application with the guidance of deadline monotonic scheduling theory (only for PERC solution). The parameters used in the scheduling analysis come from the experimental results which are got from the benchmark application tests. Such an approach can guarantee that the deadlines of the real-time tasks are met in a soft way.

During the test and evaluation, some existing difficulties for the real-time Java solutions today to be applied in automotive systems are found and listed as follows:

- Some solutions have too limited platforms support constraints to suit the automotive system's requirements, such as RTSJ implementations.
- Some solutions are still immature to build large and complex real-time Java applications required in the automotive systems.
- For some solutions, there are lack of ways and tools to cater for the existing mature real-time scheduling theories to build hard real-time systems

Nevertheless, there are also some remarkable progresses shown in the real-time Java domain, which makes the future of real-time Java applied in automotive systems still bright and expectable:

- Real Time Specification of Java has addressed most of the problems facing real-time Java and raised a set of considerable methods to solve them.
- Some approaches for developing soft real-time Java applications has emerged, these approaches can be used to build non-safety-critical soft real-time applications based on empirical and experimental knowledge.

- More and more companies and entities have devoted into the real-time Java domain so that more mature research and products will emerge quickly.

8.2 Limitations and Future Prospects

The limitations of this thesis lie in the following two aspects:

- Fixed target device and operating system limit the use of available real-time Java solutions. Especially, none of the RTSJ implementations can fit in this environment, which makes the implementation and testing part of this thesis lack variation.
- To build the sample automotive application, the necessary CAN access library is not originally designed for the real-time application. This causes the overall temporal behavior of the sample application more unpredictable and also causes some problems and difficulties when applying the scheduling analysis work for the sample application.

For the future prospects of this area, first, there is some expectancy for the real-time Java solution providers:

- There are still very few solutions for the hard real-time application development and hence we expect to see more of them in the future. Such real-time Java solution should lean upon one specific scheduling theory and provide the necessary parameters for the convenience of deploying this scheduling analysis by either explicitly listing them in the product specification or giving platform dependant low level tools to make it easy to obtain.
- A specific real-time Java product is expected to provide the support of a wide range of embedded platforms, so that the real-time application developers don't need to switch to new platforms to cater for changing a new product. This is also one of the benefits conventional Java provides.
- As the theoretical analysis of this thesis shows, the additional semantics specified by RTSJ are appropriate and convenient for helping the real-time application development in Java. After its reference implementation releases, its applicability will be verified and improved. As more real-time applications are developed following RTSJ, their experiences and practical knowledge can be referred and introduced to the automotive system development.

Some future work for real-time Java approach in the automotive system domain:

- Existing Java code reuse issues:

  The existing Java applications that are not time critical can be reused directly by setting a low priority in the system. However, the necessary attributes of these applications such as CPU utilization, still need to be examined to prove that they have sufficient time to run and will not starve.

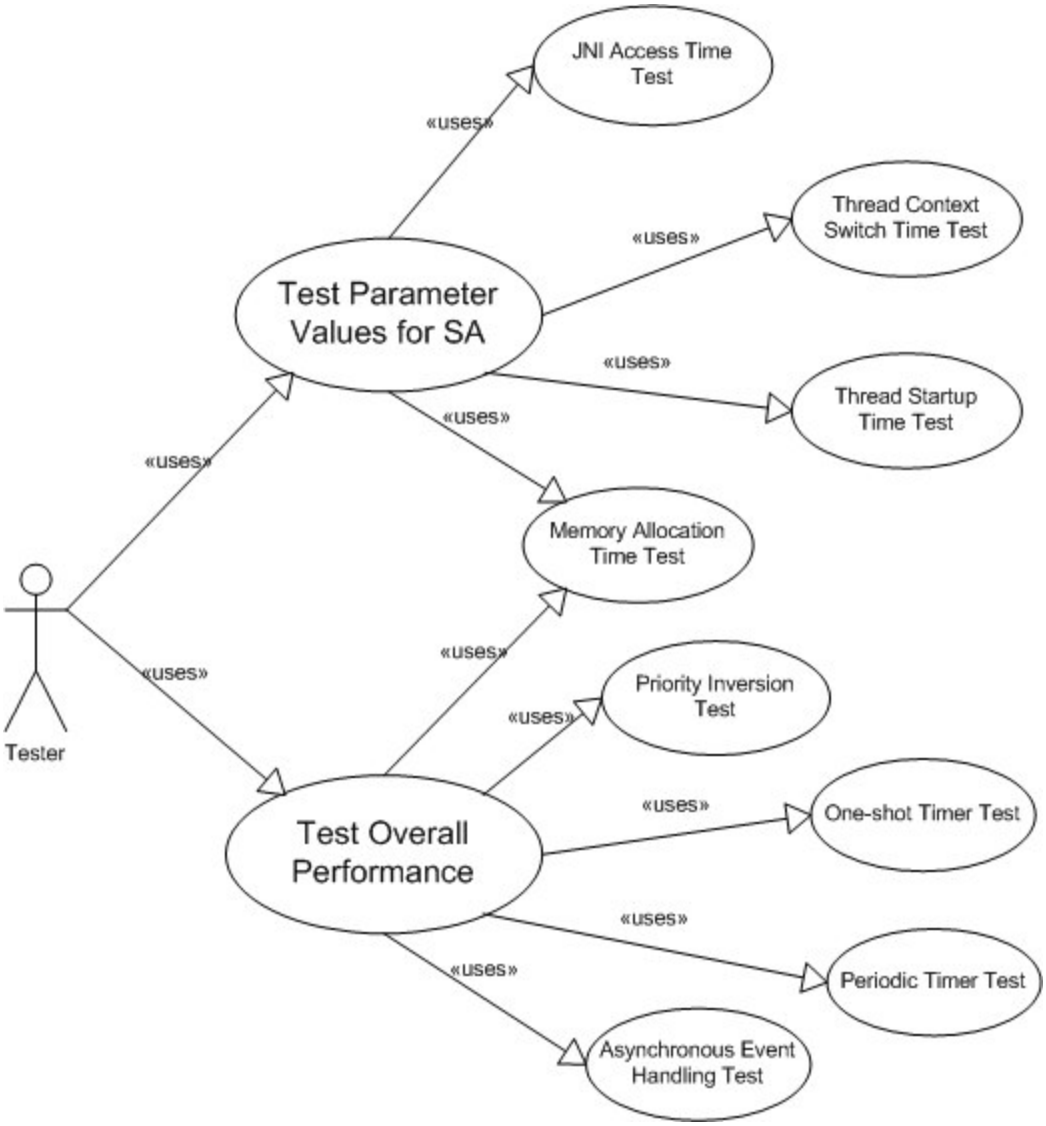
  The existing non-real-time applications, which are time critical or can directly determine the upper-layer application's predictability such as the CAN access library used in this thesis, cannot gain the real-time capability by simply migrating them to a real-time Java environment for the reasons we discussed in Chapter 4.Therefore, they should be redesigned following the real-time

system design criteria in order to be used by the real-time system development. Additionally, the scheduling analysis work is necessary to lead and verify the entire design work.

- Building in-vehicle real-time function prototype using real-time Java technologies

  Given the theoretical fundaments and practical experiences achieved in this thesis, future work in this domain can be building in-vehicle realistic real-time function prototypes in real-time Java so that the time for real-time Java being applied in automotive system won't be too far away.

Appendix A
Use-case Diagrams for the Benchmark Application



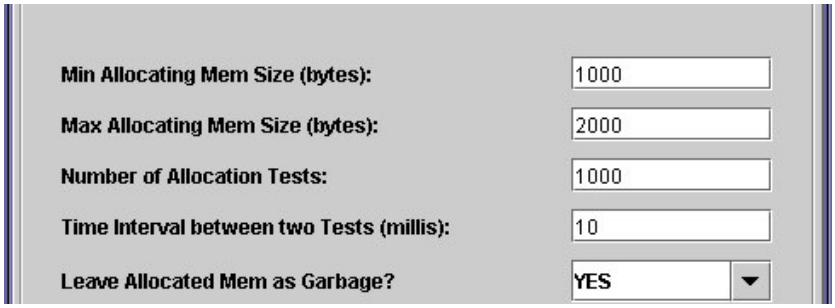
Appendix A Benchmark Application Use-case Diagram

Appendix B

Benchmark Test Cases and Results

Memory Allocation Test 1

Test parameters:

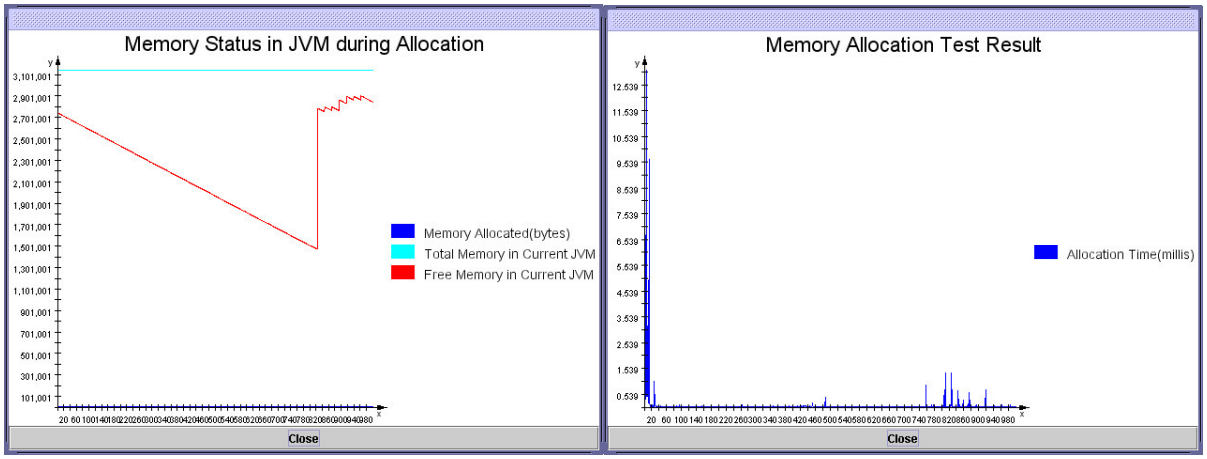


Test description:

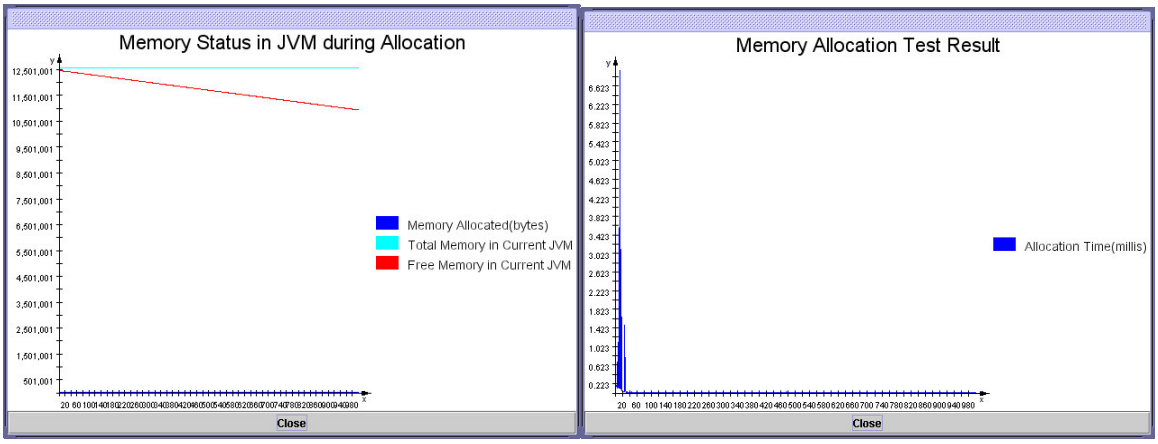
It is the first test after the benchmark application started on the target device.

Test results:

PERC virtual machine:



IBM J9:



Appendix B 1 Memory Allocation Test 1

Memory Allocation Test 2

Test parameters:

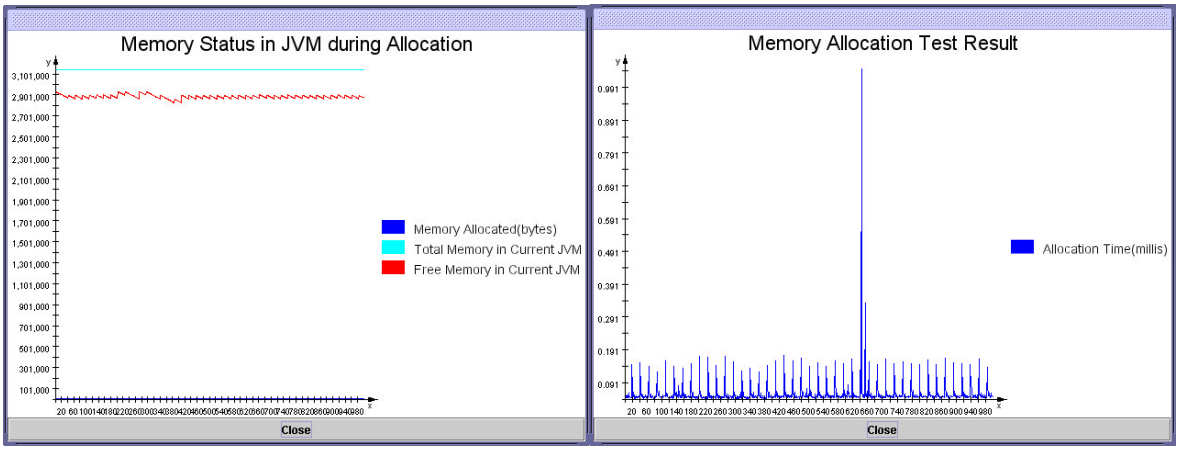
The same as used in test case 1

Test description:

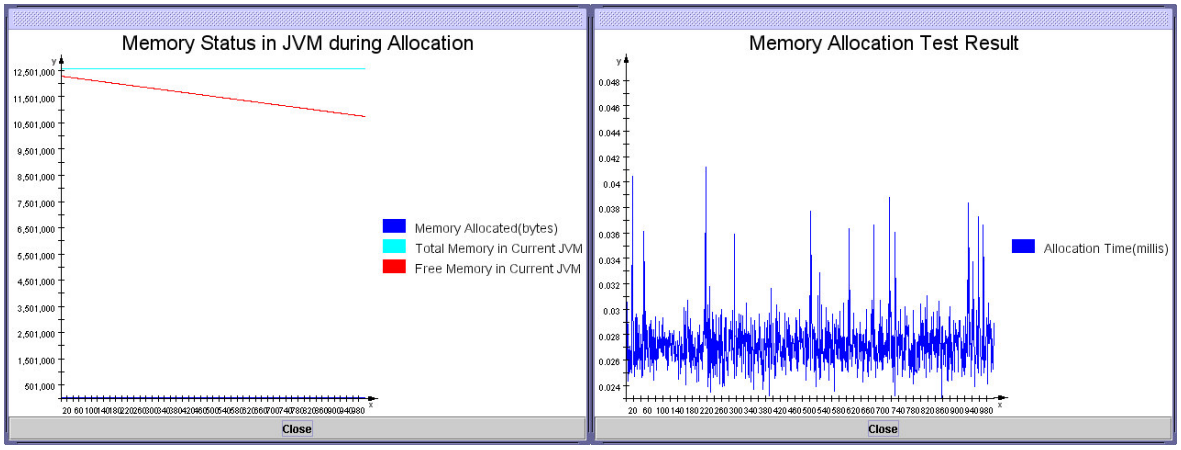
Following the first memory test.

Test results:

PERC virtual machine:



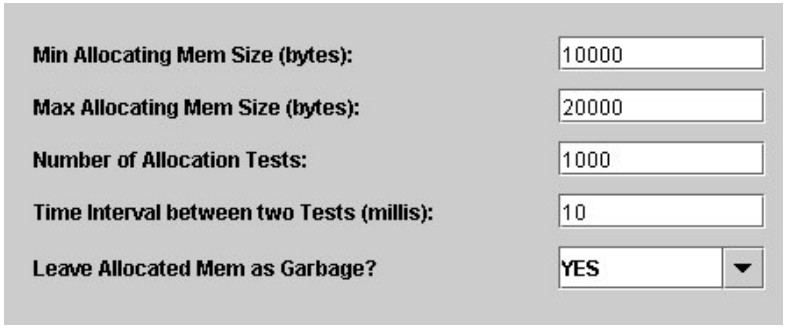
IBM J9:



Appendix B 2 Memory Allocation Test 2

Memory Allocation Test 3

Test parameters:

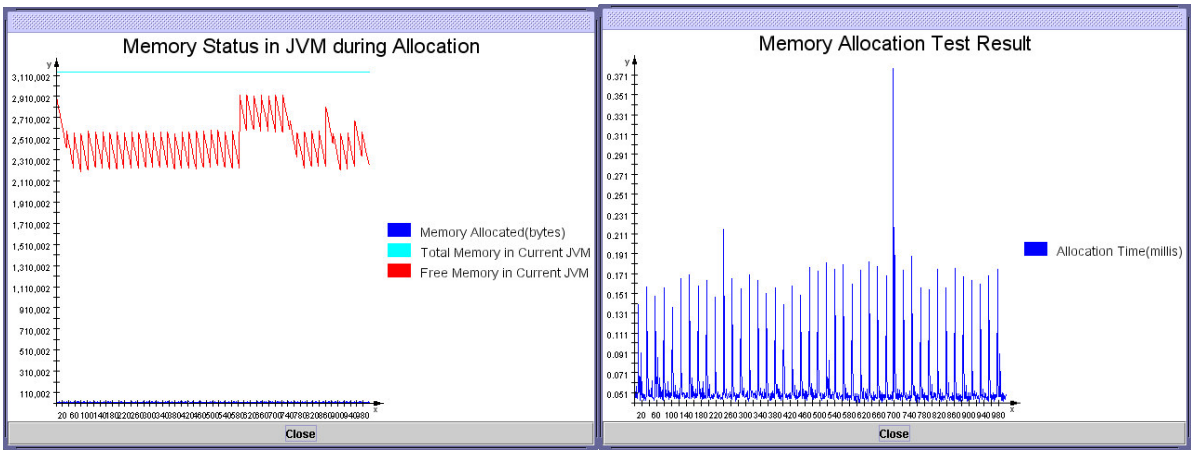


Test description:

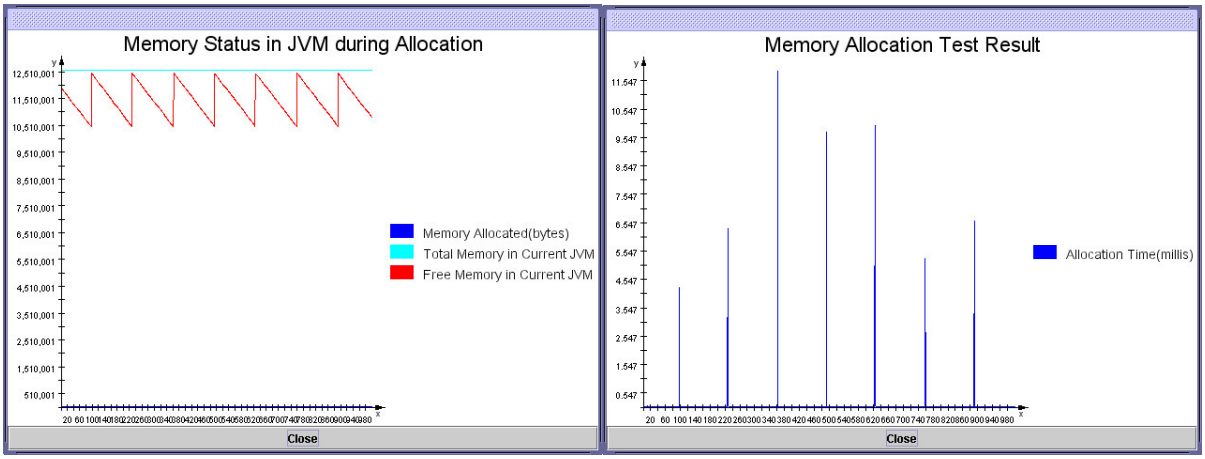
Allocating large size of memories in a high frequency after the startup stage of the benchmark application

Test results:

PERC virtual machine:



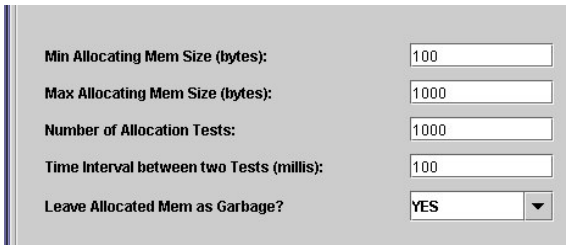
IBM J9:



Appendix B 3 Memory Allocation Test 3

Memory Allocation Test 4

Test parameters:

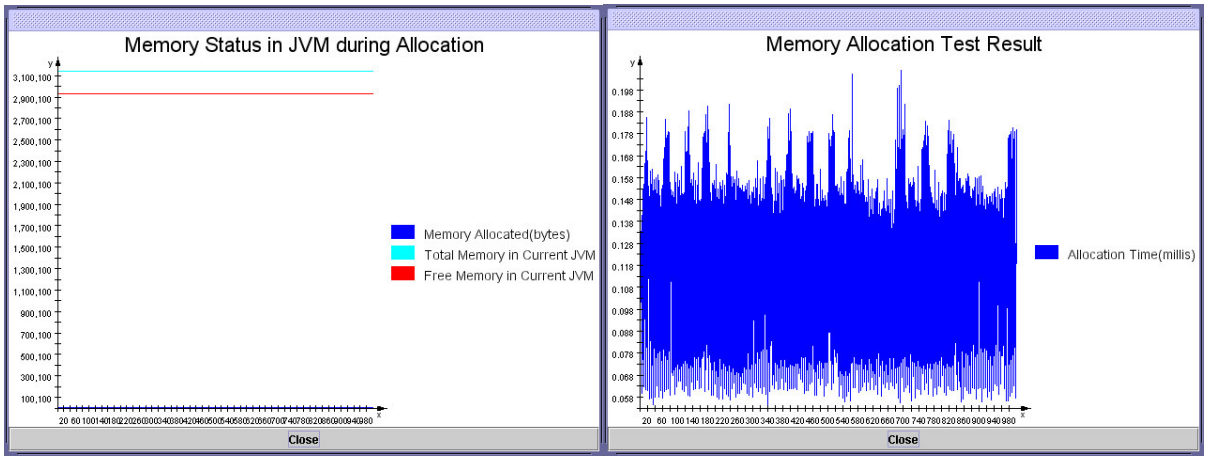


Test description:

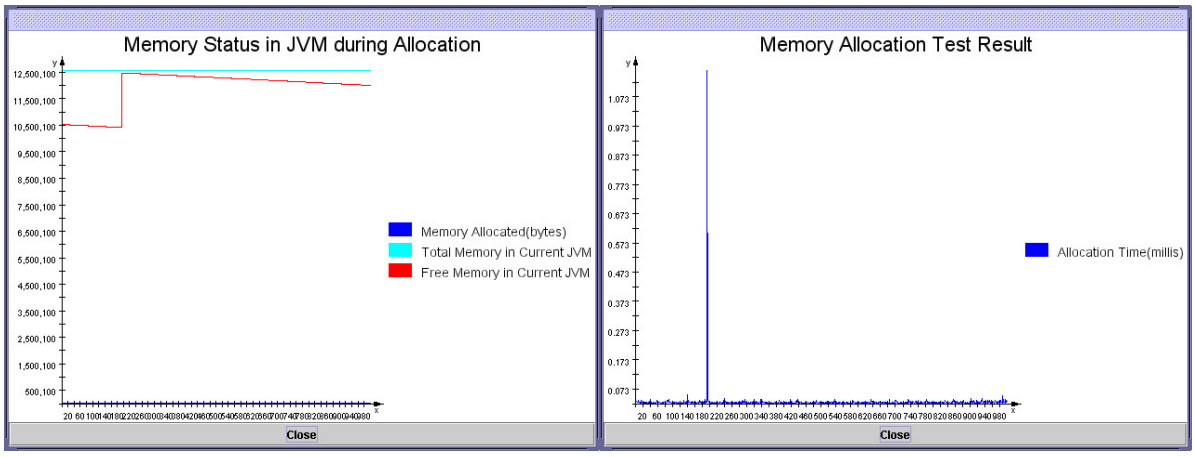
Sample application memory allocation circumstance simulation

Test results:

PERC virtual machine:



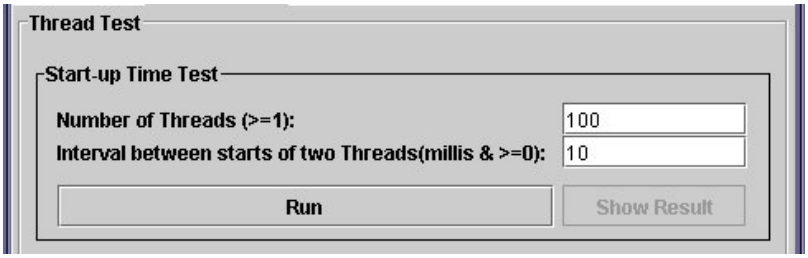
IBM J9:



Appendix B 4 Memory Allocation Test 4

Thread Startup Time Test 1

Test parameters:

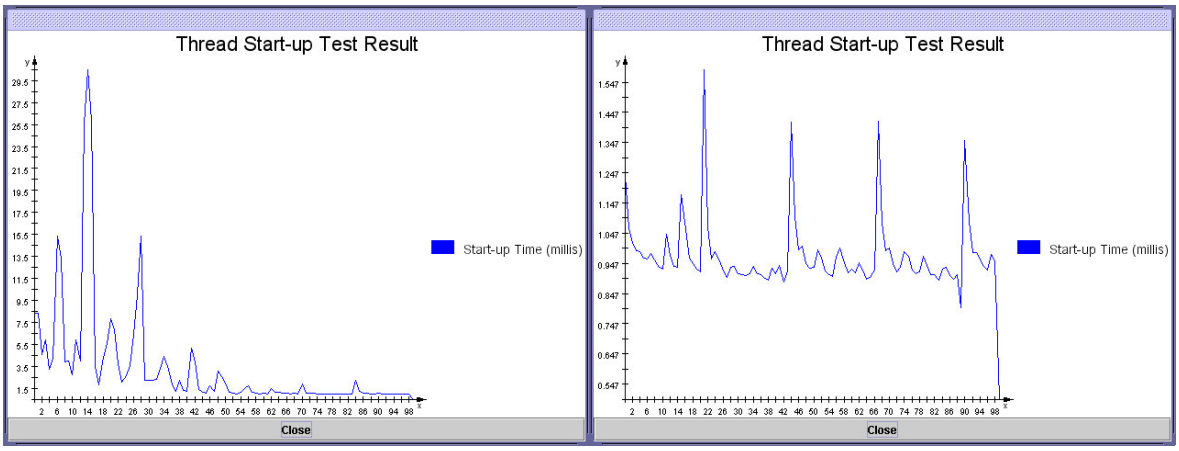


Test description:

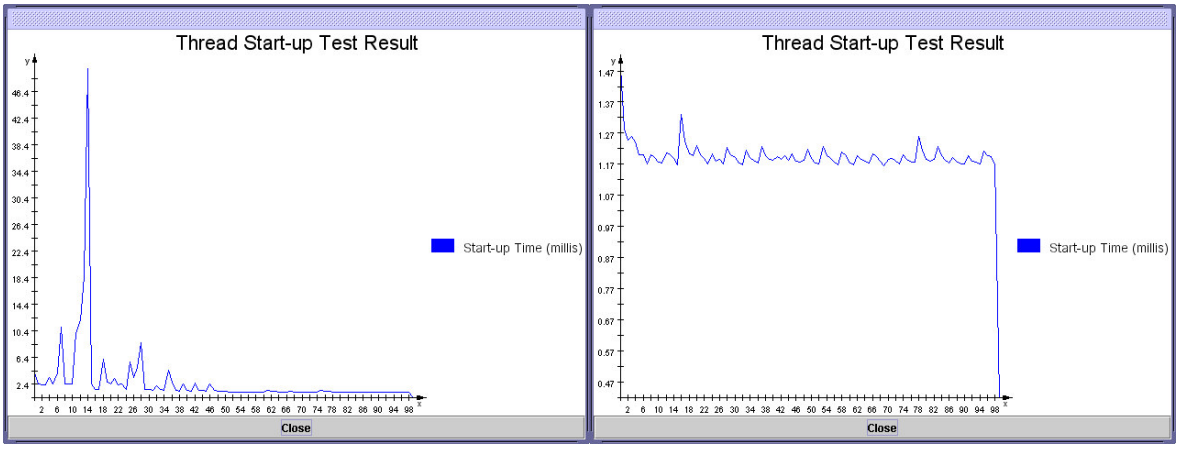
Deploy the same test twice, one is on the startup stage of the application, the other one is after that

Test results:

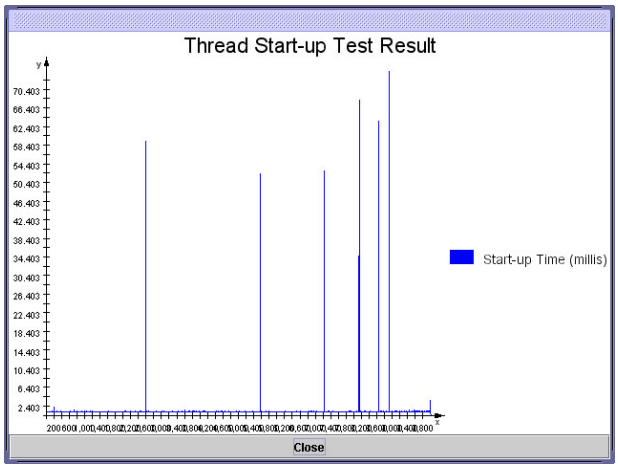
PERC virtual machine: (the one on the left is the startup stage test)



IBM J9: (the one on the left is the startup stage test)



When prolonging the test period, the result from J9 begins to suffer long time GC operation:



Appendix B 5 Thread Startup Time Test 1

Thread Yielding Context-switch Time Test

Test parameters:

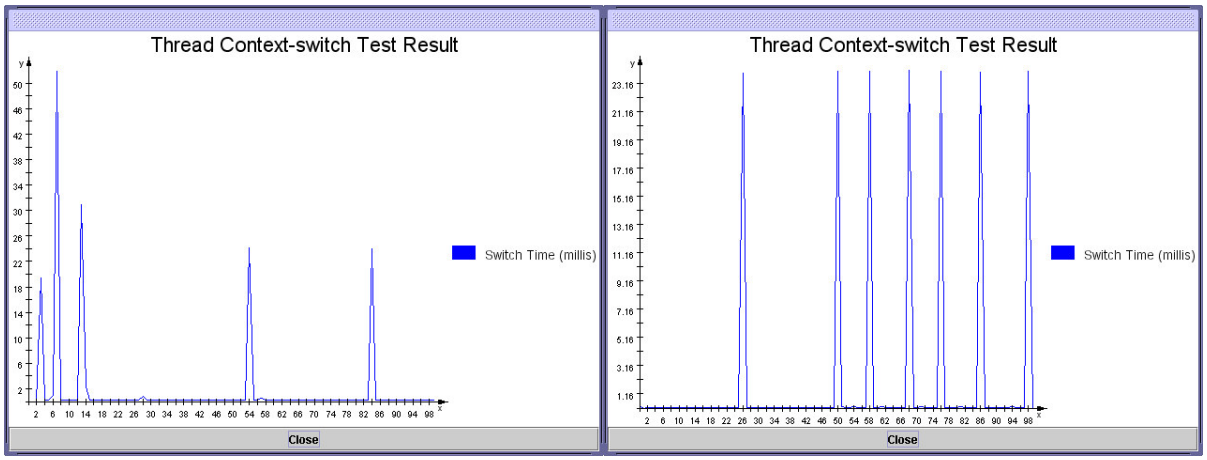
Test number: 100

Test description:

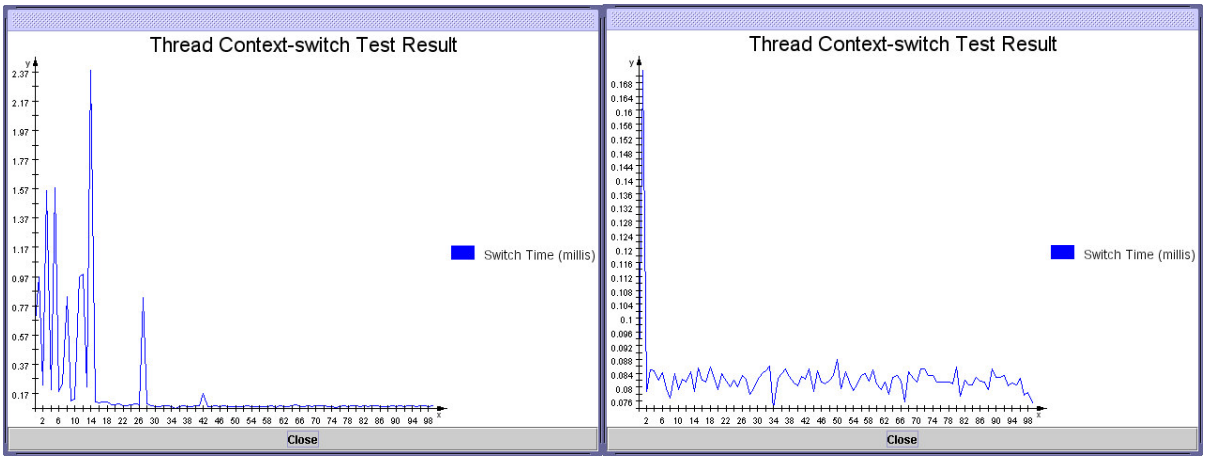
Deploy twice, one on the startup stage, the other after the startup stage

Test results:

PERC virtual machine (the one on the left is the startup stage test)



IBM J9 (the one on the left is the startup stage test):



Appendix B 6 Thread Yielding Context-switch Time Test

Thread Notification Context-switch Time Test

Test parameter:

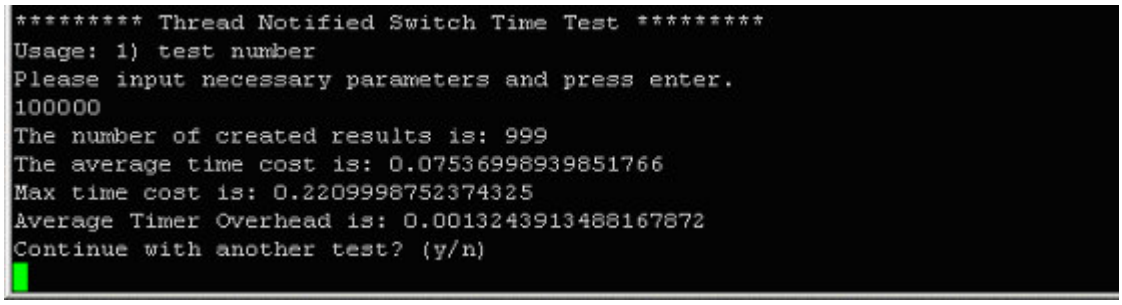
Test number: 100,000

Test description:

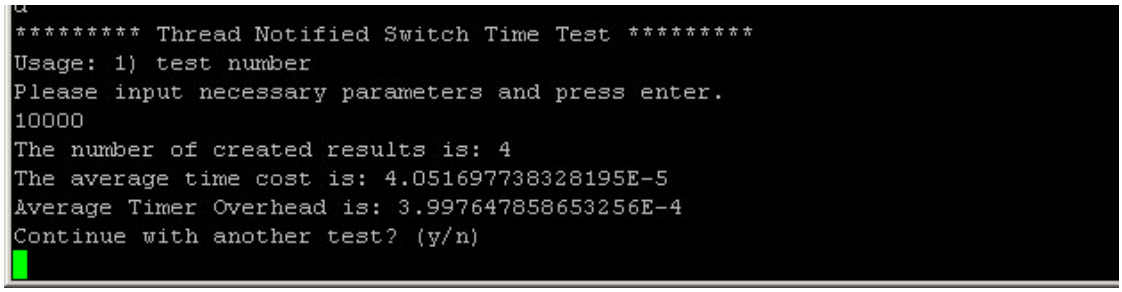
After the startup stage of the test application

Test results:

PERC virtual machine (without generating graphic charts due to the large test number)



IBM J9 (failed results)



Appendix B 7 Thread Notification Context-switch Time Test

Thread Entering Unlocked Monitor Time Test

Test parameter:

Test number: 100,000

Test description:

After the startup stage of the test application

Test results:

PERC virtual machine (without generating graphic charts due to the large test number)

```
********* Entering Unlocked Monitor Time Test *********
Usage: 1) test number
Please input necessary parameters and press enter.
100000
The number of created results is: 100000
Average Time cost: 0.0024503080367747803
Max Time cost: 0.9129917863531101
Average Timer Overhead: 0.001154678145934757
Continue with another test? (y/n)
```

IBM J9

```
********* Entering Unlocked Monitor Time Test *********
Usage: 1) test number
Please input necessary parameters and press enter.
100000
The number of created results is: 100000
Average Time cost: 0.0011347108644160545
Max Time cost: 0.1910168282686625
Average Timer Overhead: 4.6801316278446004E-4
Continue with another test? (y/n)
```

Appendix B 8 Thread Entering Unlocked Monitor Time Test

Thread Priority Inversion Test

Test parameter:

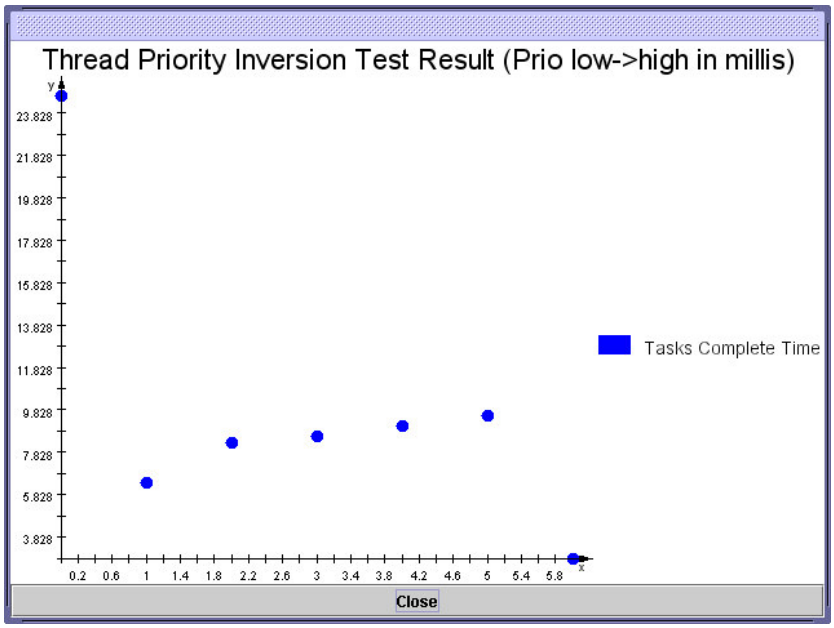
Medial priority thread number: 5 (the five threads having the same priority)

Test description:

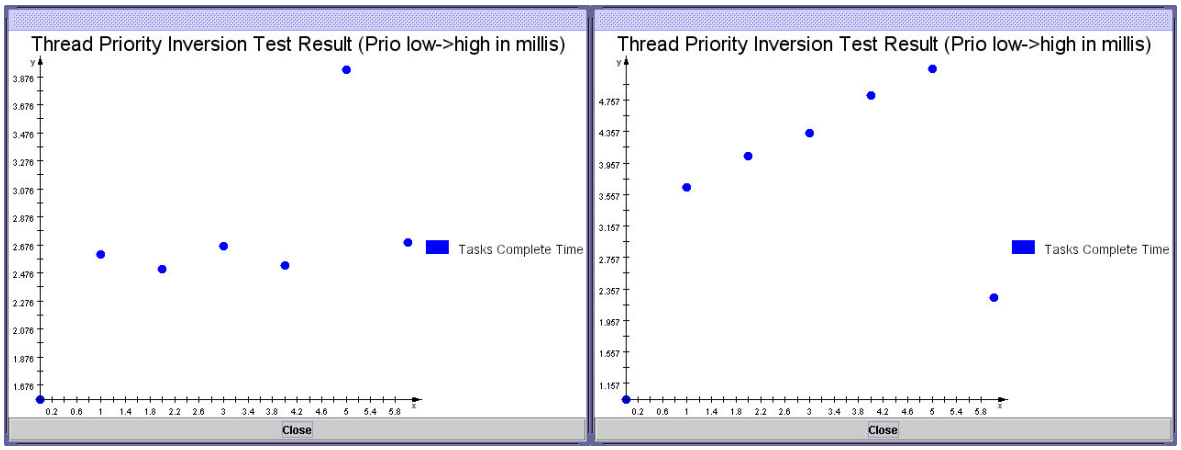
No further constraints

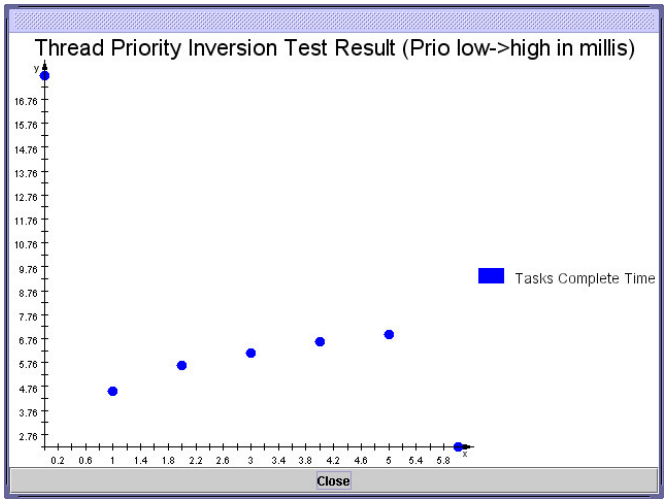
Test results:

PERC virtual machine (priority inversion avoided):



IBM J9 (three kinds of results, priority inversion often occurs)





Appendix B 9 Thread Priority Inversion Test

One-shot Timer Test

Test parameter:

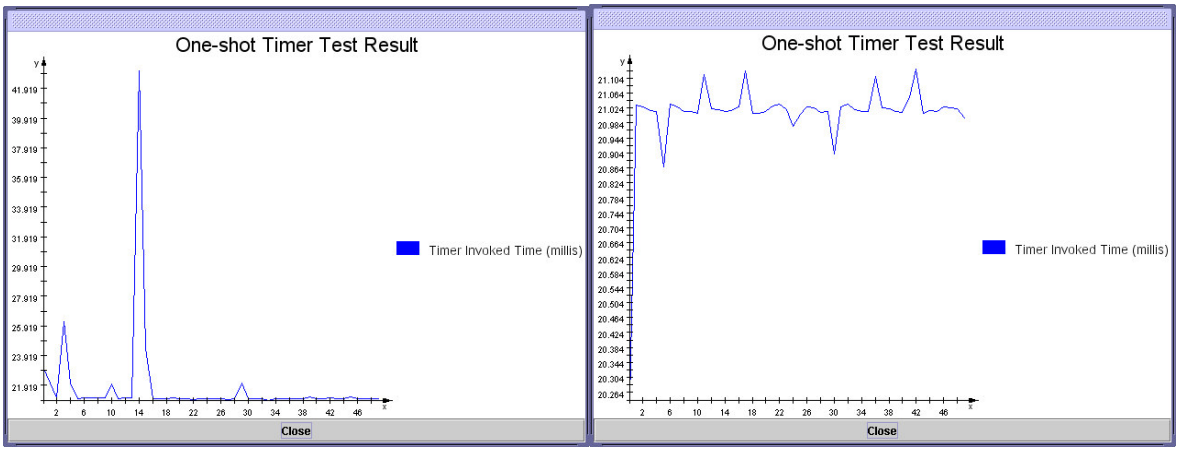
Timer release delay: 20 ms; test number: 50

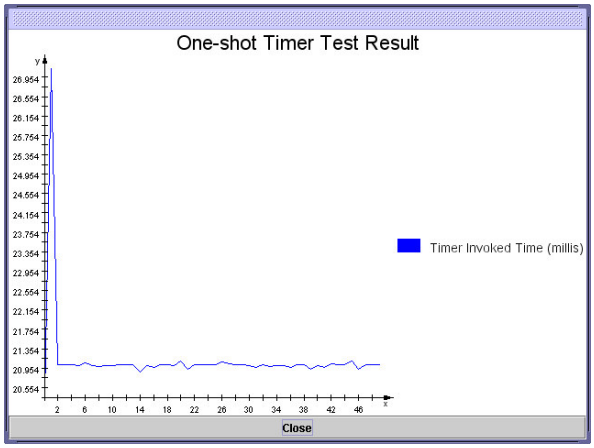
Test description:

First testing on the startup stage, and then testing on the later steady stage

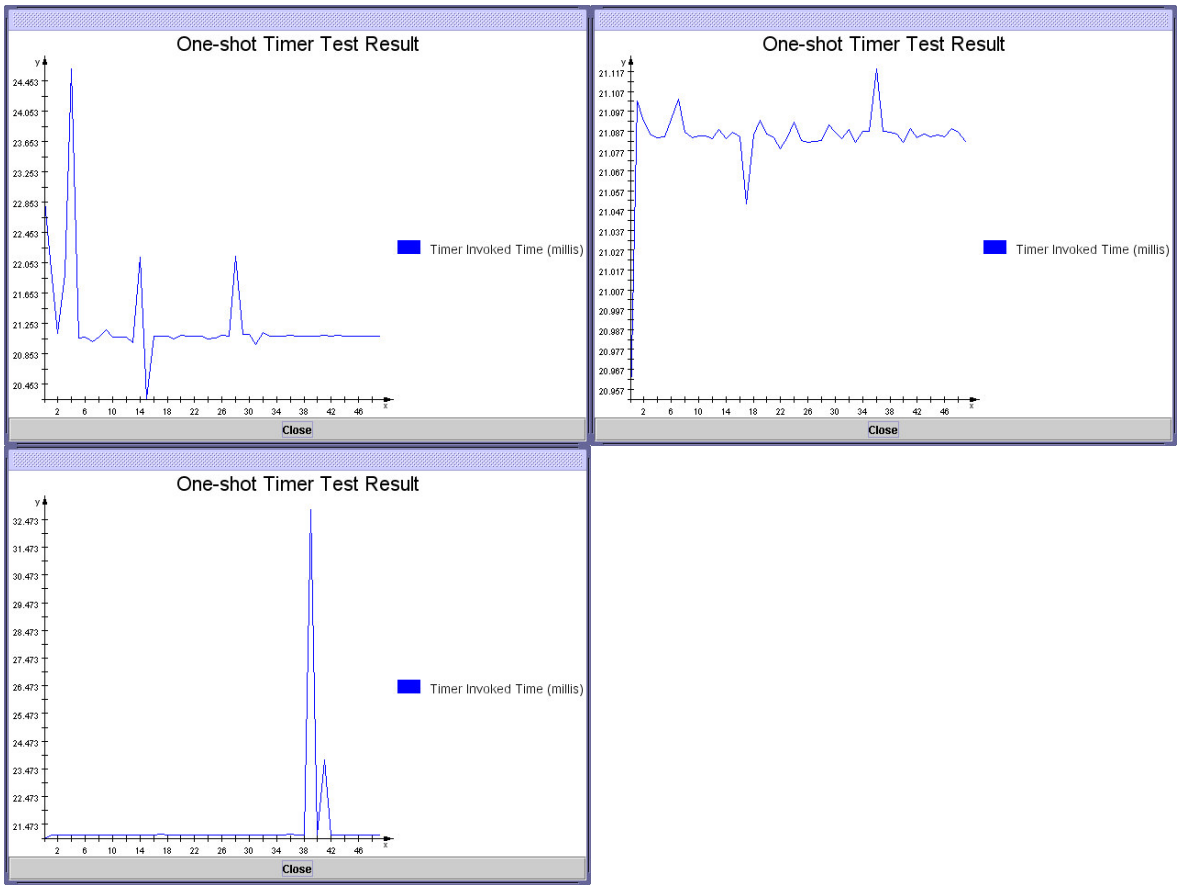
Test results:

PERC virtual machine (first one shows the startup scenario, second one shows the common behavior, the third one is a special case happening sometimes):





IBM J9 (first one shows the startup scenario, second one shows the common behavior, the third one is a special case happening sometimes):



Appendix B 10 One-shot Timer Test

Periodic Timer Test

Test parameter:

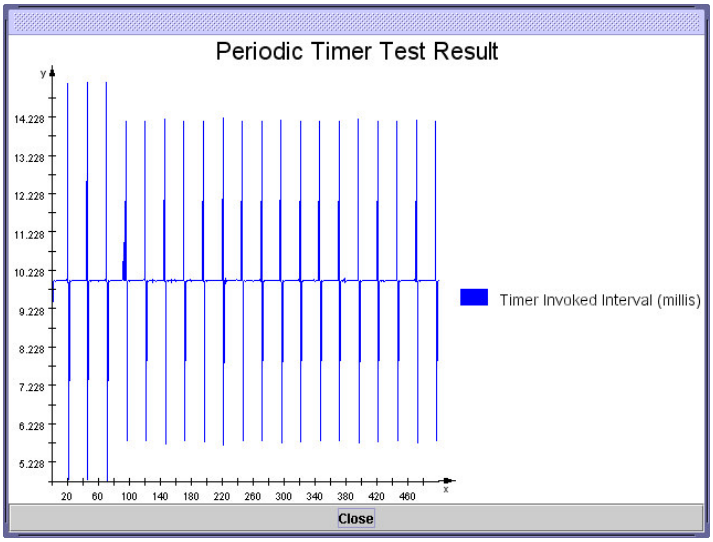
First release delay: 10 ms; period: 10 ms; test number: 500

Test description:

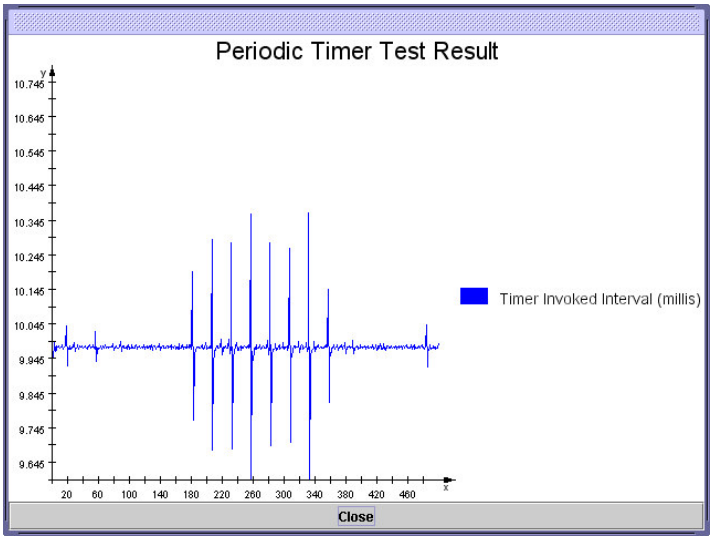
Testing on the steady stage after the application startup

Test results:

PERC virtual machine:



IBM J9:



Appendix B 11 Periodic Timer Test

Asynchronous Event Handling Test 1

Test parameter:

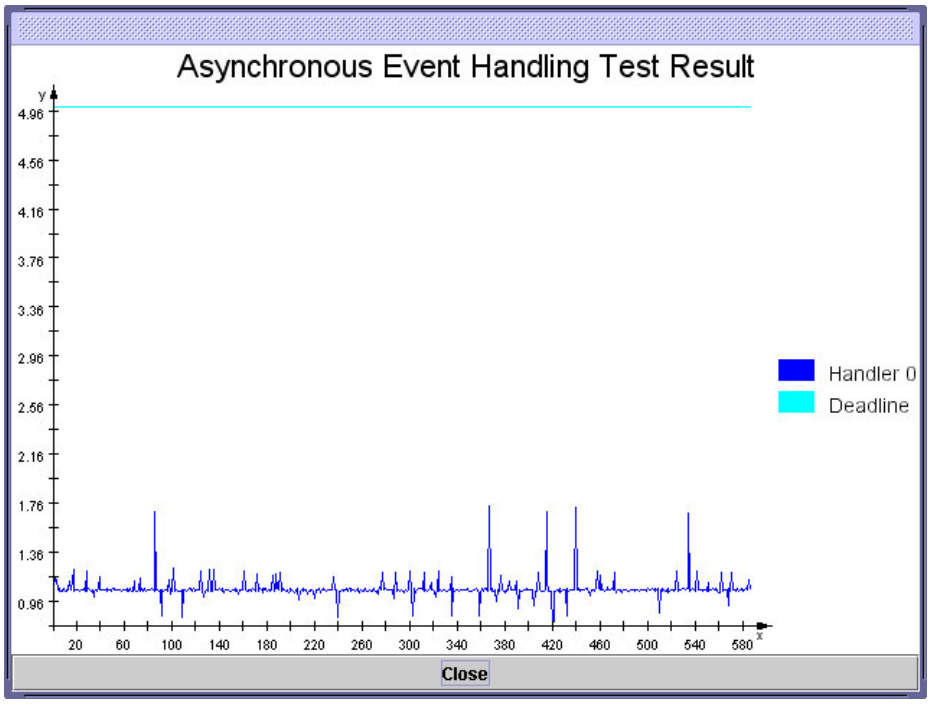
Min Event Interval (millis >0):	5
Max Event Interval (millis >=Min):	10
Number of Asynchronous Event Handling Tasks (>=1):	1
Size of Memory Block needed for each task (bytes):	200
Number of Memory Blocks needed for each task (>=0):	5
Number of Fundamental Calculation needed (>=0):	10000
Deadline of Tasks (millis >0):	5
Total testing time (millis >Deadline):	5000
Running casual low priority work background?	NO
Casual Work Interval (millis >=0):	

Test description:

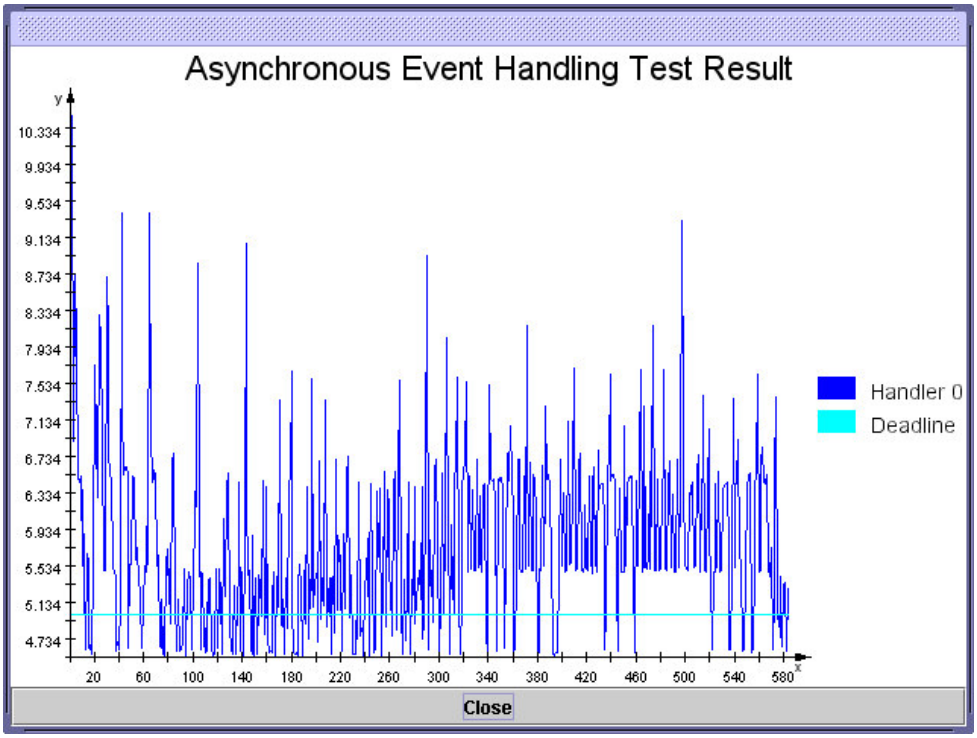
Testing on the steady stage after the application startup

Test results:

PERC virtual machine:



IBM J9:



Appendix B 12 Asynchronous Event Handling Test 1

Asynchronous Event Handling Test 2

Test parameter:

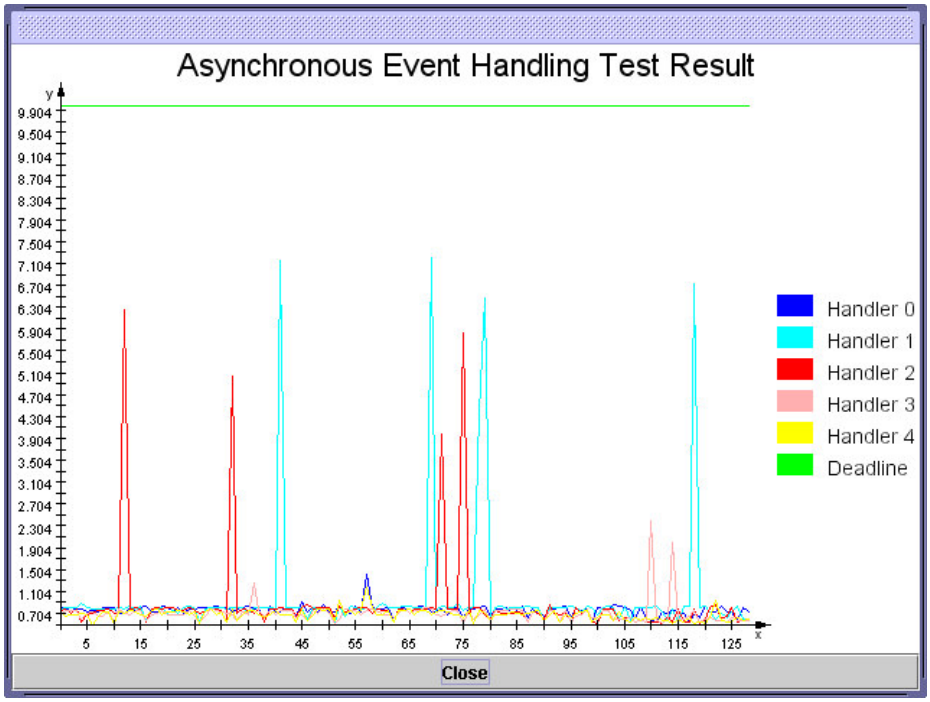
Min Event Interval (millis >0):	50
Max Event Interval (millis >=Min):	100
Number of Asynchronous Event Handling Tasks (>=1):	5
Size of Memory Block needed for each task (bytes):	200
Number of Memory Blocks needed for each task (>=0):	5
Number of Fundamental Calculation needed (>=0):	10000
Deadline of Tasks (millis >0):	10
Total testing time (millis >Deadline):	10000
Running casual low priority work background?	YES
Casual Work Interval (millis >=0):	500

Test description:

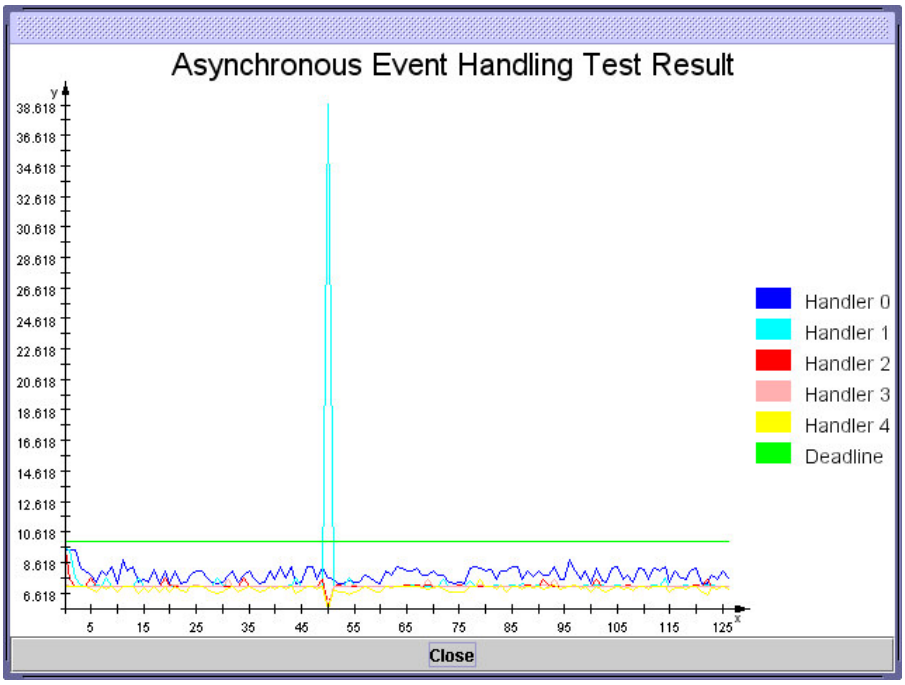
Testing on the steady stage after the application startup

Test results:

PERC virtual machine:



IBM J9:



Appendix B 13 Asynchronous Event Handling Test 2

JNI Access Time Test 1

Test parameter:

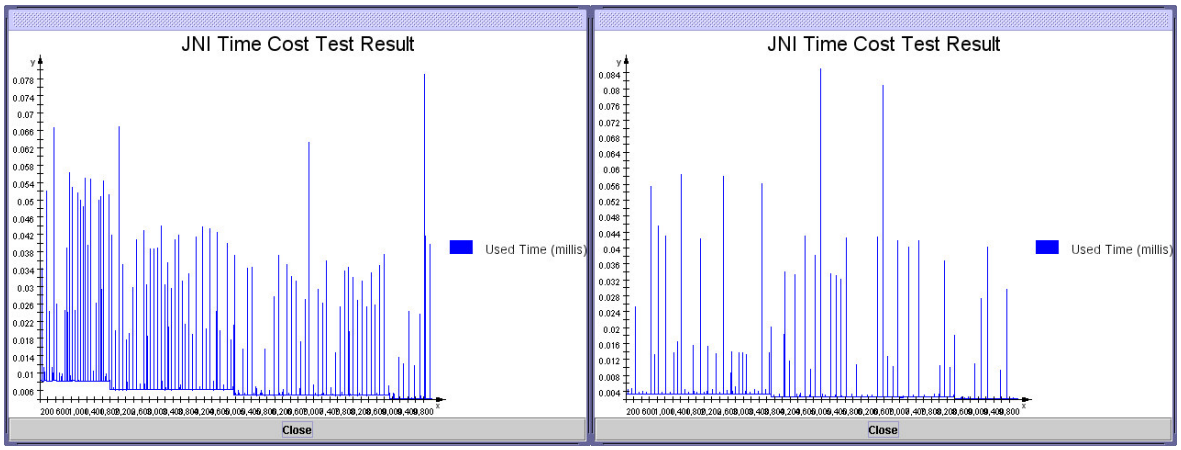
Test number: 10,000 (The last test use 100,000 to show the long period situation)

Test description:

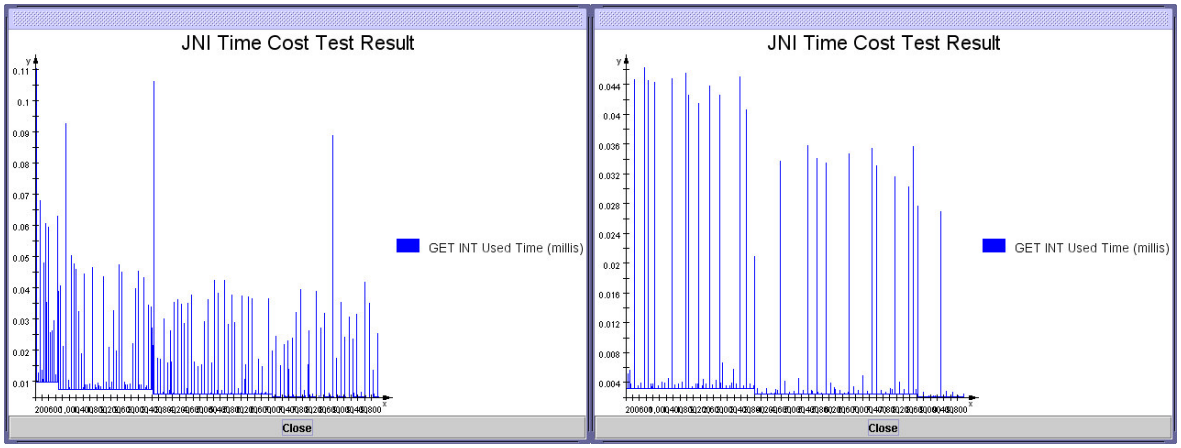
After the benchmark application first loaded the JNI native library used in this test

Test results:

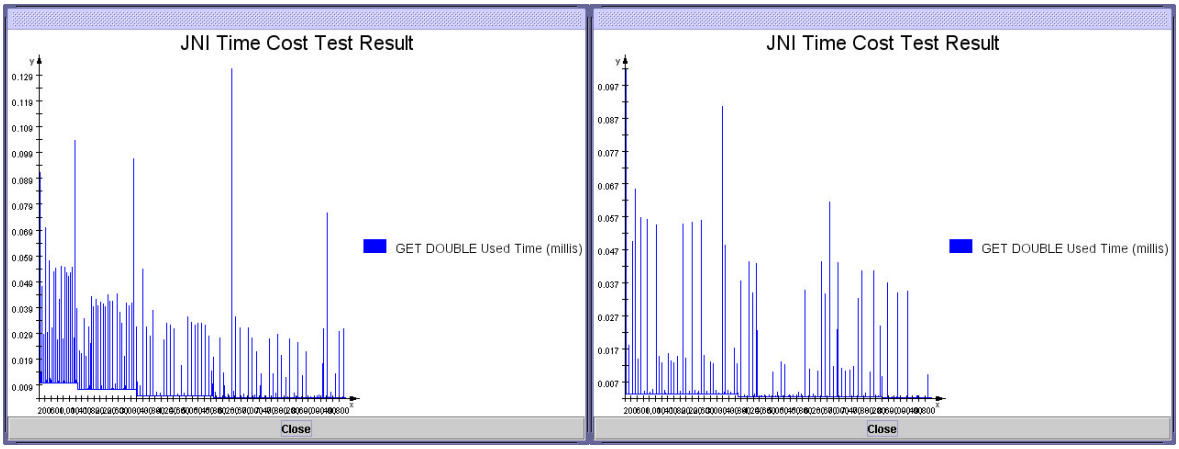
Get byte through JNI (PERC virtual machine VS IBM J9):



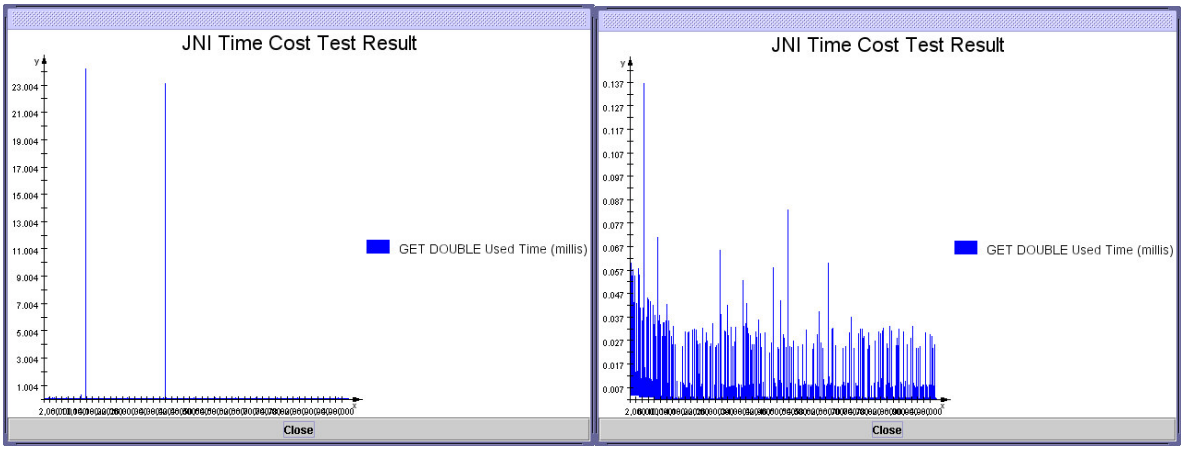
Get int through JNI (PERC virtual machine VS IBM J9):



Get double through JNI (PERC virtual machine VS IBM J9):



After running 100000 times PERC virtual machine starts to show the peak value while J9 still remains steady (left is PERC virtual machine)



Appendix B 14 JNI Access Time Test 1

JNI Access Time Test 2

Test parameter:

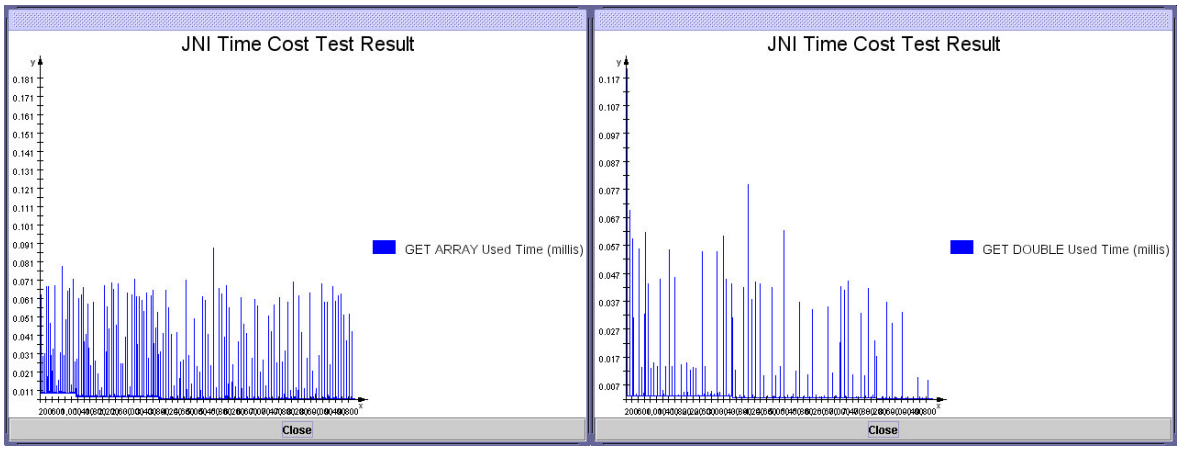
Byte-array length: 100; Test number: 10,000

Test description:

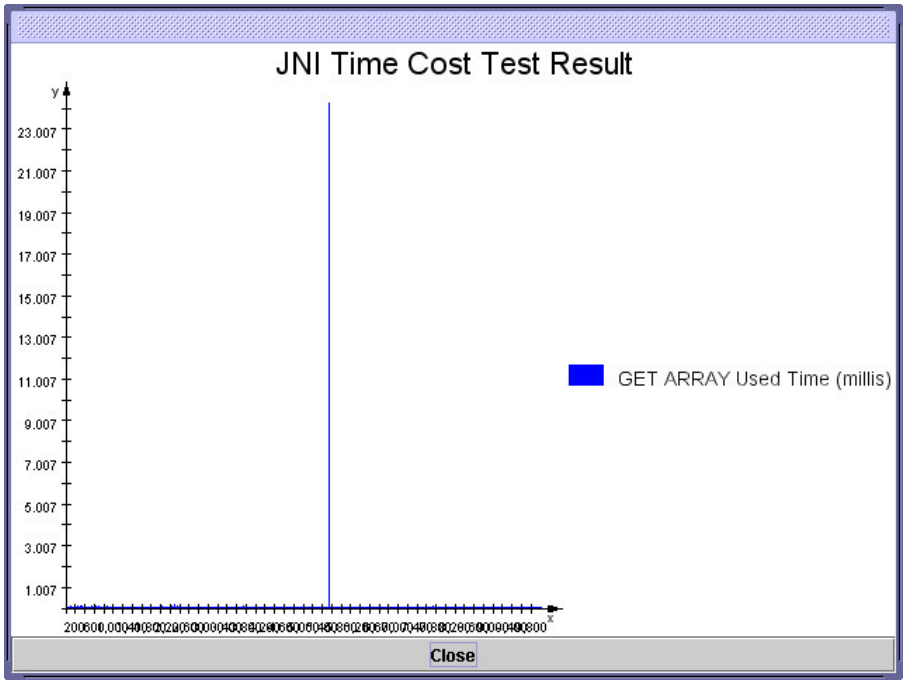
After the benchmark application first loaded the JNI native library used in this test

Test results:

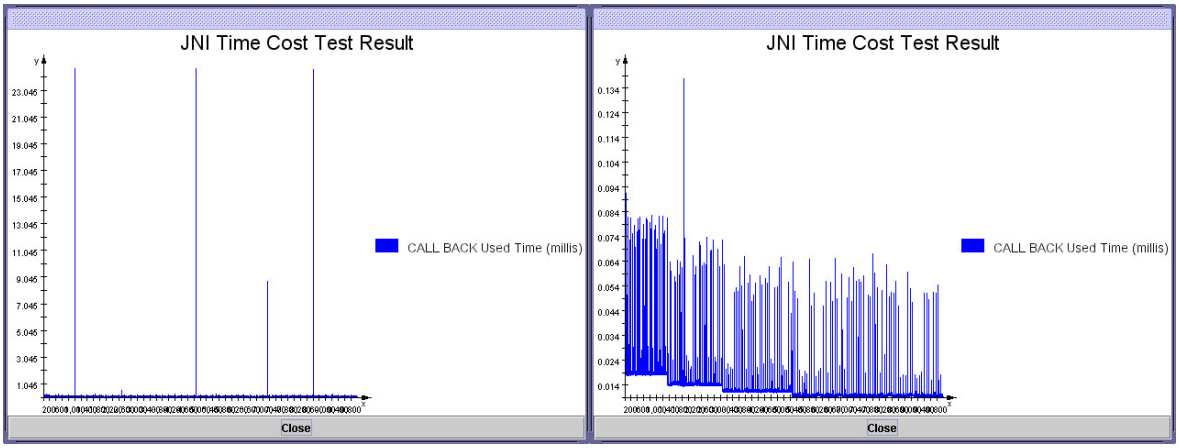
Get byte-array through JNI (PERC virtual machine VS IBM J9):



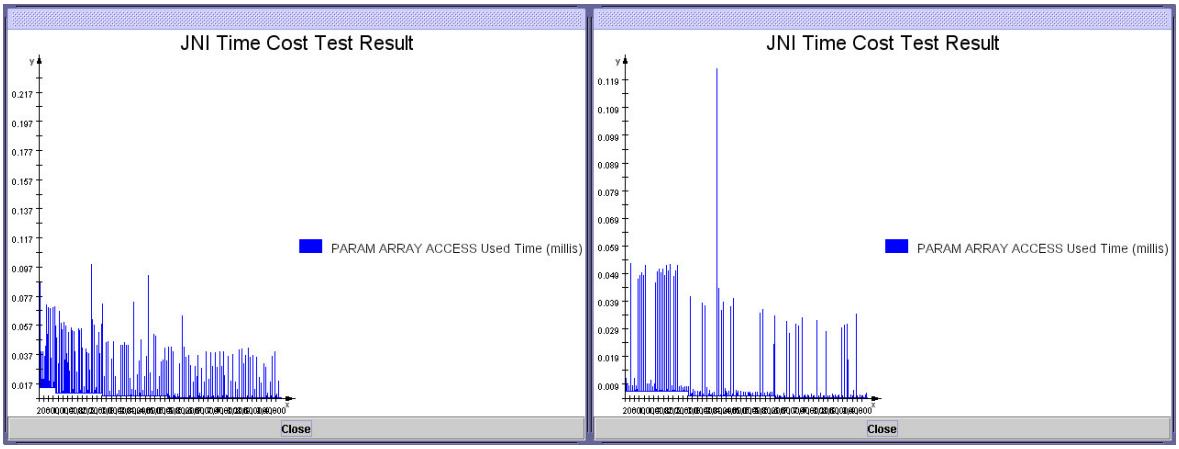
Some unstable peak value appears sometimes in PERC virtual machine, which never occurs to J9



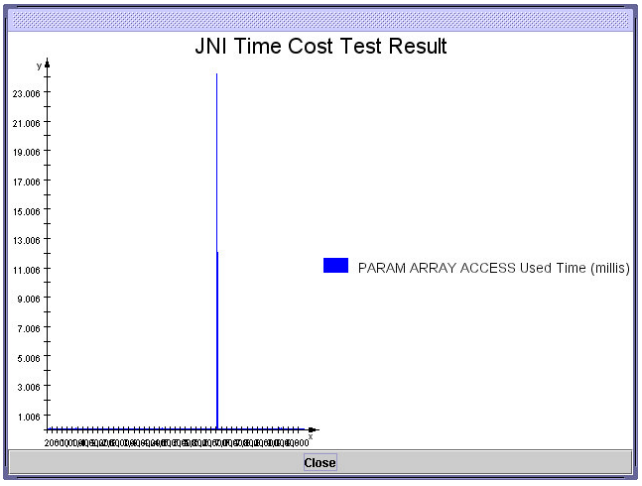
Get byte-array through JNI call back (PERC virtual machine VS IBM J9):



Get byte-array through passing member variable to JNI (PERC virtual machine VS IBM J9):



PERC virtual machine still shows instability



Appendix B 15 JNI Access Time Test 2

References

[1] Burns, A. & Wellings, A. *Real-Time Systems and Programming Languages*, 3rd edition, Addison-Wesley, 2001

[2] Tokuda, H.; Mercer, C.W.; Ishikawa, Y.; Marchok, T.E.; *Priority inversions in real-time communication*, Real Time Systems Symposium, 1989, Proceedings. , 5-7 Dec. 1989 Pages: 348 - 359

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

[4] *Real-Time Core Extensions*. J Consortium. September 2000 [http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf, June, 2004]

[5] *Jamaica Real-time Java virtual machine* [http://www.aicas.com, June, 2004]

[6] *PERC Real-time Java virtual machine* [http://www.newmonics.com, June, 2004]

[7] *aJile Real-Time Java solution* [http://www.ajile.com/ajevb100.htm, June, 2004]

[8] A. Burns, *Scheduling Hard Real-Time Systems: A Review*, Software Engineering Journal 6(3), pp. 116-128 (1991).

[9] Colnaric, M.; *State of the art review paper: advances in embedded hard real-time systems design*, Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on, Volume: 1, 12-16 July 1999, Pages:37 - 42 vol.1

[10] N. C. Audsley, A. Burns, M. Richardson, and A. Wellings. *Hard Real-Time Scheduling: The Deadline-Monotonic Approach*. In Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software, pages 133-137, May 1991

[11] Liu, C. L., J. W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM 20(1) (1973) pp.46-61.

[12] Joseph, M., P. Pandya, *Finding Response Times in a Real-Time System*, BCS Computer Journal (Vol. 29, No.5, Oct 86) pp.390-395

[13] Sha, L., Rajkumar, R. and Sathaye, S. S., *Generalized rate-monotonic scheduling theory: A framework for developing real-time systems*, Proceedings of the IEEE, Vol. 82, No. 1, January, 1994.

[14] Sha, L.; Rajkumar, R.; Lehoczky, J.P.; *Priority inheritance protocols: an approach to real-time synchronization Computers*, IEEE Transactions on Volume: 39, Issue: 9, Sept. 1990, Pages: 1175 – 1185

[15] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. The Journal of Real-Time Systems, 1:27–60, 1989.

[16] B. Sprunt. *Aperiodic task scheduling for real-time systems*, Ph.D. thesis, Carnegie Mellon University, 1990.

[17] Leung, J.Y.T., and J. Whitehead. *On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks*, Perf. Eval. (Netherlands), 2, pp. 237-250, 1982

[18] A. Burns. Preemptive *Priority-Based Scheduling: An Appropriate Engineering Approach*. In S. H. Son, editor, Advances in Real-Time Systems, pages 225-248. Prentice Hall International, Inc., Englewood Cliffs, NJ, 1994.

[19] Leen, G.; Heffernan, D.; *Expanding automotive electronic systems*, IEEE Computer, Volume: 35, Issue: 1 , Jan. 2002, Pages:88 - 93

[20] Deicke, A.: *The Electronics Concepts of the new 7 Series*. BMW intern, EE-1, 2003.

[21] *OSEK/VDX Operating System 2.2.1 Specification* [http://www.osek-vdx.org, June, 2004]

[22] *OSGi Service Platform Release 3*, [http://www.osgi.org/resources/spec_download.asp, June, 2004]

[23] *CAN Specification 2.0B*, [http://www.can.bosch.com/docu/can2spec.pdf, June, 2004]

[24] A. Albert, W. Gerth, *Evaluation and Comparison of Real-Time Performance of CAN and TTCAN*, Robert Bosch GmbH Proceedings 9th International CAN Conference; iCC 2003; Munich

[25] G. Leen and D. Heffernan; *TTCAN: A new time-triggered controller area network*. Microprocessors and Microsystems, 26(2): 77–94, 2002

[26] R. Belschner, J. Berwanger, C. Ebner, H. Eisele, S. Fluhrer, T. Forest, T. F¨uhrer, F. Hartwich, B. Hedenetz, R. Hugel, A. Knapp, J. Krammer, A. Millsap, B. M¨uller, M. Peller, and A. Schedl. *FlexRay – Requirements Specification,* FlexRay Consortium, Internet: http://www.flexray.com, Version 2.0.2, April 2002

[27] *QNX Neutrino RTOS (v6.2.1) Book set*, [http://www.qnx.com/developers/docs/qnx_6.1_docs/qnxrtp/index.html, June, 2004]

[28] *OSGi Expert Group Structure*, [http://www.osgi.org/about/eg_overview.asp, June, 2004]

[29] Fridtjof Siebert, *Bringing the full Power of Java Technology to Embedded Real-time Applications*, MSy'02 Embedded Systems in Mechatronics, 3-4. Oct 2002, Winterthur, Switzerland [http://www.aicas.com/papers/msy02.pdf, June, 2004]

[30] L. Carnahan and M. Ruark, etc; *Requirements for Real-Time Extensions for the Java Platform*, Sep, 1999 [http://www.nist.gov/rt-java, June, 2004]

[31] Bollella, G.; Gosling, J.; *The real-time specification for Java,* IEEE Computer, Volume: 33, Issue: 6, June 2000, Pages:47 - 54

[32] L. Sha, R. Rajkumar, and J. Lehoczky, *Real-Time Computing using Futurebus+*, IEEE Micro, June 1991, pp. 30-33; 95-99.

[33] PERC Whitepaper: *Differentiating Features of the PERC Virtual Machine*, [http://www.newmonics.com/perceval/perc_whitepaper.shtml, June, 2004]

[34] Kelvin D. Nilsen: *Doing Firm-Real-Time with J2SE APIs*. OTM Workshops 2003: 371-384

[35] Kelvin D. Nilsen: *Using Java for Reusable Embedded Real-Time*, Component Libraries, Aonix, 2004

[36] Fridtjof Siebert, *Hard Real-time Garbage Collection in Modern Object Oriented Programming Languages*, Invited talk for the Java User Group Switzerland WIP Session Zürich, 7 February 2000, [http://www.aicas.com/papers/jugs_7-Feb-2000_slides.pdf, June, 2004]

[37] Jakob Axelsson: *Holistic Object-Oriented Modelling of Distributed Automotive Real-Time Control Applications*. ISORC 1999: 85-92

[38] Corsaro, A.; Schmidt, D.C.; *Evaluating real-time Java features and performance for real-time embedded system*; Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE, 24-27 Sept. 2002