CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrotechnical Engineering

Department of Control Engineering

Benchmark of Real-Time Java implementations

Diploma Thesis

Bc. Michal Janoušek

Vedoucí práce: Ing. Michal Sojka, Ph.D.

January 2012

Abstract

Diploma thesis is concerned with programming real-time systems. It is focusing on use of Java programming language in real-time systems. The base part is reprogramming algorithm for Monte-Carlo localization (MCL) of robots from C language to Java. The main goal is to create methodology for comparison of different implementations of RT Java. The part of thesis is development of benchmark application, which is based on MCL algorithm and its outputs are used for the comparison.

In the next part is described analysis of application requirements and reasons for use of individual technologies. Description of realization of benchmark development, its single parts and design of web interface for comparison of results and publication as open source project is described too.

Thesis contains final benchmark application and mentions the possibilities of further development.

Abstrakt

Diplomová práce se zabývá programováním systémů realného času. Zaměřuje se na využití jazyka Java v systémech reálného času. Základem je přeprogramování algoritmu pro Monte-Carlo lokalizaci (MCL) robotů z jazyku C do Javy. Cílem je vytvoření metodologie pro porovnání jednotlivých implementací RT Javy. Součástí práce je vývoj benchmarkové aplikace jejíž základem je MCL algoritmus.

V další části je popsána na analýzu požadavků na aplikaci a důvody pro použití jednotlivých technologií. Součástí je také popis realizace vývoje aplikace, jednotlivých částí, popis webového rozhraní pro porovnávání výsledků a zveřejnění jako open source projektu.

Práce obsahuje výslednou benchmarkovou aplikaci a zmiňuje se o možnostech budoucího vývoje.

Acknowledgment

I would like to thank my advisor Mr. Ing. Michal Sojka, Ph.D., who had enough patience and made possible to create this work. I would to thank Tomáš Kalibera, Ph.D., who has experiences with RTSJ and provided me help. I can not leave out my parents, i would like to thank them for their support.

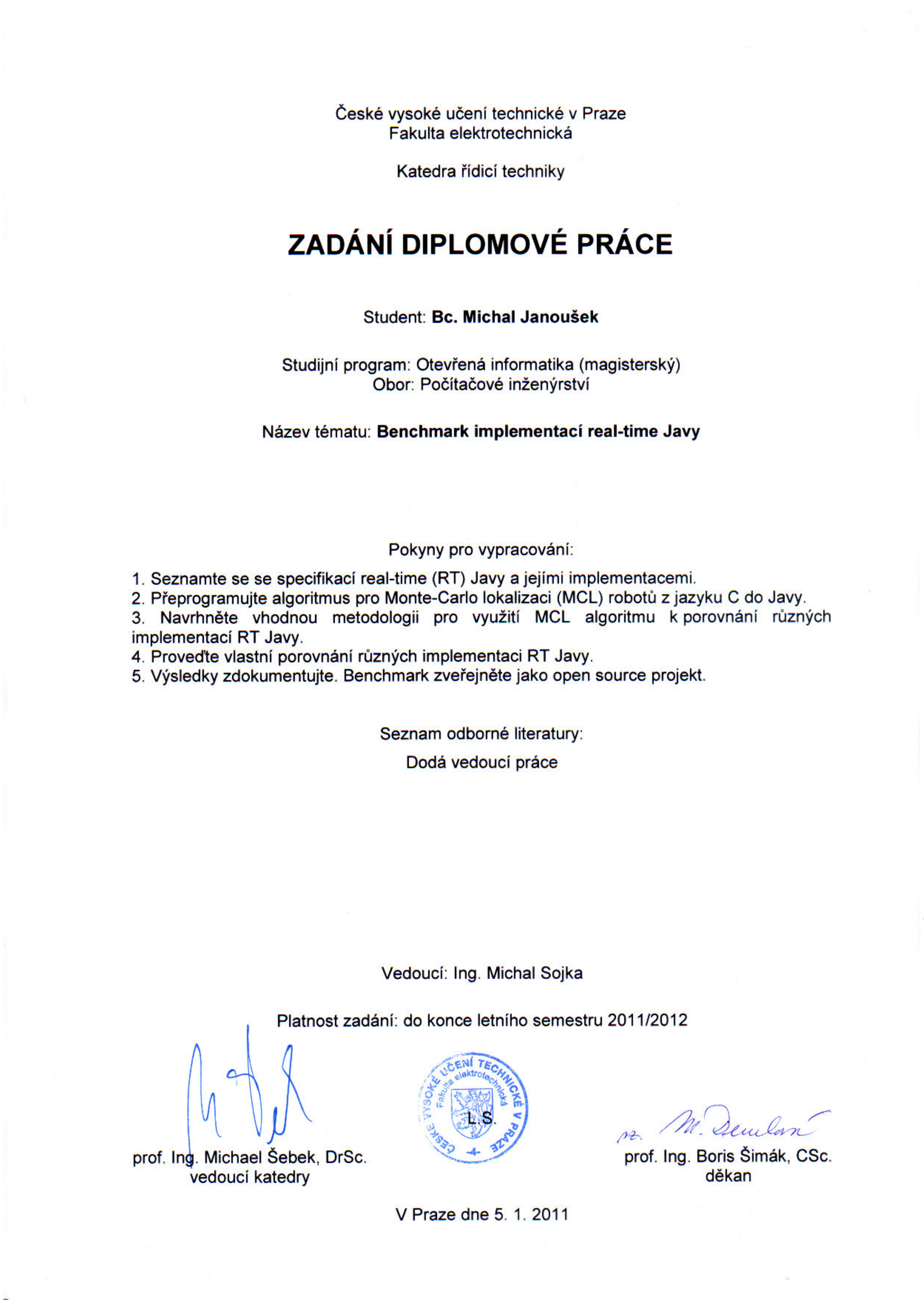
Declaration

I hereby declare that following thesis is my own work and I used only sources (literature, projects, SW etc.) quoted in enclosed reference list.

In Prague, January 1, 2012 Bc. Michal Janoušek

.

Zadání



České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Michal Janoušek**

Studijní program: Otevřená informatika (magisterský)
Obor: Počítačové inženýrství

Název tématu: **Benchmark implementací real-time Javy**

Pokyny pro vypracování:

1. Seznamte se se specifikací real-time (RT) Javy a jejími implementacemi.
2. Přeprogramujte algoritmus pro Monte-Carlo lokalizaci (MCL) robotů z jazyku C do Javy.
3. Navrhněte vhodnou metodologii pro využití MCL algoritmu k porovnání různých implementací RT Javy.
4. Proveďte vlastní porovnání různých implementací RT Javy.
5. Výsledky zdokumentujte. Benchmark zveřejněte jako open source projekt.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Michal Sojka

Platnost zadání: do konce letního semestru 2011/2012

[illegible]

České vysoké učení technické v Praze Fakulta elektrotechnická -4- L.S.

[illegible]

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Boris Šimák, CSc.
děkan

V Praze dne 5. 1. 2011

Table of Contents

List of Figures

List of Examples

Chapter 1

Introduction

In this time control engineering and real-time systems are very interesting parts of computer science. There is big space for research and development in this area. The number of devices with embedded microprocessor is increasing rapidly. This devices are managing different tasks in our live and have to be programmed properly. They are more or less real-time oriented, which does not necessarily mean, that they have to be programmed like real-time systems. But for some applications the usage of real-time programing is essential, so the programmers should have to do their job with this in mind. Programming applications for real time systems has its own specifics and all of programming languages are not usable for this purpose.

For real-time applications are usually used C based languages such as Real-Time POSIX programming language and there are some specialized languages such as ADA 95. Some program developers are trying to use Java, because Java is very popular and widely used language. Java programmers believe, that is possible to use their beloved language to program everything.

Java has a lot of advantages for programmers. Applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers use the "write once, run anywhere"(WORA) principle. Java is currently one of the most popular programming languages in use, and is widely used from application software to web applications. [1] JVMs are designated for high throughput, because Java is widely used for web applications development and throughput is very important requirement in usage on the web. But there are another JVM implementations focused on different parameters. For example there is Dalvik JVM from Google used in Google's Android operating system (OS) for embedded devices (smartphones, tablets, etc.). So programmers want to use Java, because it is well-known language and they want to use what they like and know, nevertheless its not fully suitable for all purposes.

There is conflict between the speed of Java programs and timing requirements in real-time systems. Standard edition of Java (Java SE) is not fully capable to satisfy requirements of soft or hard real time systems. So there is a specification of Java called Real-Time Specification for Java (RTSJ), which extends standard Java with package *javax.realtime*. There is not only one implementation of RTSJ, so it is possible to compare different implementations of big companies like IBM or Oracle (Sun Microsystems).

Applications include embedded systems (thermostats, controllers), robotics, industry and science. Embedded systems are my study specialization, so it is one of the reasons why choose this topic. When I will have to program such system it will be good to know something about real-time programming. So I wanted to know what real-time Java means and how to use it. This led me to the writing of this thesis. When I was finding the subject of this work, I knew, that it should be something connected with Java. Of course I could not forget about embedded system, and that is why i chose this topic.

Hard real-time applications had to be run on systems, which are optimized for real-time. This type of OS is called real-time operating system (RTOS).

In this work was my goal to develop the benchmark application as free open source software, which can be used by everyone, who is interested in this area. It should provide the comparison between different JVM and show us some information about performance of RTSJ implementations. I am trying to introduce the developing of this benchmark application and explain why I do something or not. But not only this benchmark will be product of this work. Very important is the interpretation of benchmark outputs. Interpretation is a part of this thesis and the outputs can be found on the web pages of this project. On this pages are the graphs and measured data for simpler comparison. I tried to use only free and open source applications and software in this work. Some Real-Time JVMs are not free of charge, because real time applications are lucrative business for companies. There are implementations with limited time of usage or you can get some with academic license, after proving that you are using it for research. So I do not have to pay for tested implementations.

Benchmark application developed in this thesis is based on Monte Carlo Localization (MCL) algorithm. The real-time models part in this application is in the receiving data from sensors. When a sensor receive data specific event occurs, which is processed by the application. Data are used to predict and update the position of a robot, which moves on a defined playground. For good results of applications in real-time systems is important the utilization of sensor data on time (before the next event occurs). MCL algorithm is not so dependant on precise timings. It works well with some deadline misses, but we can use this model for comparison in benchmark.

This thesis brings a view on different implementations of real-time Java virtual machines and trying to compare their performance. The results could be used for finding out how good is real-time Javas performance in general opposite to others real-time languages.

The remainder of this thesis is organized as follows: in the next section is introduced real-time systems programming. In Section 3 is described the benchmark and used metrics. Then, in Section 4, is described the implementation of benchmark. In Section 5 are discussed the results of benchmark outputs and evaluation of JVM implementations. Finally the conclusion is given in Chapter 6.

Chapter 2

Real-Time Systems Programming

Programing for real-time systems is connected with real-time computing (RTC) and is often misunderstood by many people. They think, that Real-time programing is maximization of application performance or minimization of response time and latency. Real-time application is equal to fast application for them. There is connection between performance and Real-time programing, but it is not the major part. Real-time applications are usually fast, because they have to satisfy Real-time constraints. So there is the reason for this opinion, but slower application can still satisfy the constraints too. Speed of applications is one of the prerequisites for successful use in real-time system, but an application in real-time systems does not need to be resource consuming. The most important requirement is predictability.

The predictability means predictable behavior of application. Simple example is executing database query multiple times. It will take a different amount of time to execute. The execution time depends on various factors. These factors are current system load, size of available memory, number of connected users to the database, speed of I/O operations, load of Internet connection if it is remote database etc. The response time is not deterministic, so it is not a real-time operation.

Above it is mentioned the latency, it means time between two events in real-time, which triggers action. When such event occurs, the response begins after first event and have to complete before next event occurs.

Requirements of real-time systems are logical and temporal:

- Logical correctness
  We get correct results of computing tasks. For example: 1+1 = 2

- Temporal correctness
  We get results in time.

The use of real-time applications is not only in embedded systems, but also in telecommunications and financial sector. Almost every control system have to satisfy soft or hard and relative or absolute deadlines.

2.1. Classification of Real-Time Systems

The classification of real-time systems is simplified description of different systems types. It is very important part for fully understanding the different behavior of systems. And it is crucial for learning how to write programs for it. Typical characteristics of almost all real-time systems [6]:

- event driven, reactive
- multi thread programing
- big problems after critical failure (dangerous or very expensive)
- fault-tolerance
- long lasting operation time, without external interventions
- predictable behavior

The logical correctness is obvious, so the classification depends on time correctness. Systems are classified according their capability to satisfy time constraints. In real-time terminology is more time constraints. The most important one is called deadline. *Deadline* is time, when job has to end.

- Relative Deadline
  amount of time, in which system has to respond to request
- Absolute Deadline
  precise time when the task must be completed

Additional important terms in real-time terminology are task, job, release time and response time. *Task* is a set of jobs, which serves for single function of system. *Job* is a instance of a task. *Release time* is opposite to deadline, a time when job is ready for execution, but it does not mean, that the job has to be executed exactly in this time. Job can be executed later, after release time. *Response time* is response time of job execution. It is the difference between the time when the job is completed and release time.

Jobs provide temporal results, which values is depending on deadline satisfaction. There are 2 main classes of real-time systems: soft real-time systems and hard real-time systems. Simple comparison of this 2 classes provides following table [6]:

2.1.1. Non Real-Time System

There is question what is the main difference between normal system and real-time system. We can say, that there are no important deadlines in a non real-time system. The deadlines of this system can be missed. But on the other hand, there are deadlines such days, when something has to be done. For example system have to send information email for users each month. Deadline is month, but there is not a problem to satisfy this constraint. So non real-time systems solves problems, which timings are obvious, and there is not any problem to provide desired functions. Real-time systems have extreme timing requirements and its satisfying is very difficult.

In the figure below is the dependency chart of value of computation on time. The value of computation is nearly constant in time, its not dependant on the deadline. In non real-time systems computations do not loose their significance unlike in real-time systems. But the more current computation is of course better, depending on concrete computation.

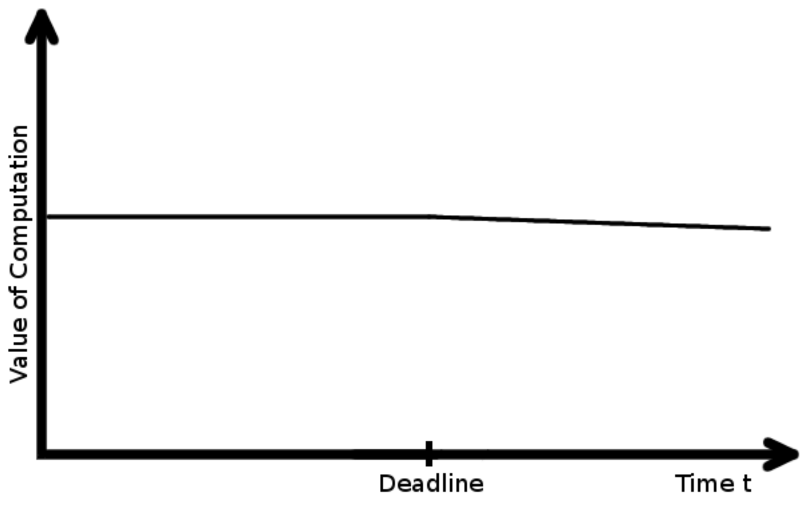


Figure 2.1. Non Real-time system

Examples for application of these systems are batch processing, web services (there can be timeouts) etc.

2.1.2. Soft Real-Time System

A soft real-time system is defined by soft deadline. Soft deadline is requested time of completion. The word soft means, that sometimes the deadline does not need to be satisfied. There is more approaches how to decide, what deadline misses are acceptable. First is percentual. For example 90% of jobs have to satisfy the deadline constraint. The second is the dependency of job value on time.

On dependency chart we can see, that before the deadline is the value of computation constant. The value is decreasing fast after deadline, but its still acceptable.

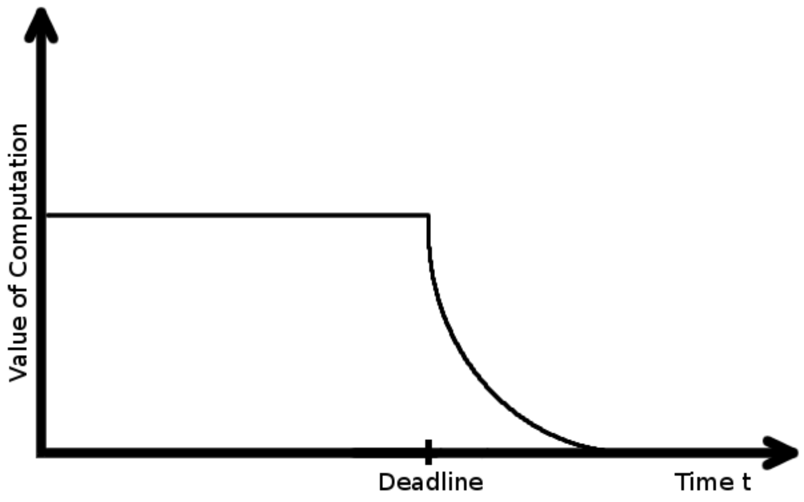


Figure 2.2. Soft Real-time system

Examples for application of these real-time systems are multimedia applications,routing in networks, automated trading systems etc.

2.1.3. Hard Real-Time System

Hard real-time system is a system, where all deadlines are strict. Deadline of hard real-time system is called hard deadline and its time of completion have to be necessarily complied. If any of the deadlines are not met, the behavior of system is wrong. This behavior can often have bad consequences.

Part of hard real-time systems is classified as safety-critical systems. These systems are directly connected to the safety of people and things. Failures of these system can cause big money loses or life threatening.

The hard deadline term can be use as firm deadline. This is described on the chart below. The function of usability or the value of computation go to the zero, when the firm deadline is reached. Firm deadlines are often used in soft real-time systems.

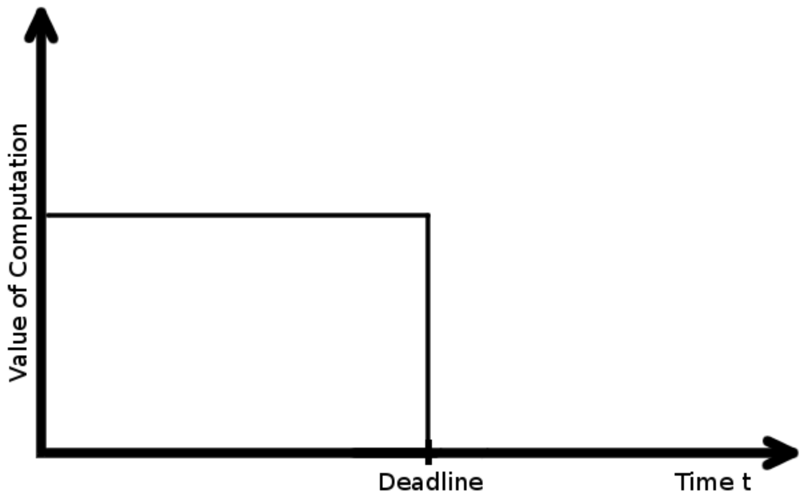


Figure 2.3. Hard Real-time system

Examples for application of these real-time system are control systems in planes, power plants, motion control, anti-lock brake system etc.

2.1.4. Classification of Scheduling Problems

Real-time applications are very sensitive to the exact time scheduling, which is actually the most important part of JVM implementations. Because of this, the development of real-time applications is often used terminology for classifying scheduling problems.

The main term in the classification of scheduling problems is job, which represents a process that must be done to meet a specific goal. The job consists of a number of smaller tasks. Task is the smallest unit of the work which uses different resources. Jobs are often repeatedly executed and because of that the timings of particular tasks are very important. Tasks are divided according to how often they performed. Existing task types follow:

- Periodic: Tasks that run on a fixed schedule, such as reading a sensor everyone millisecond

- Sporadic: Tasks that do not run on a fixed schedule but that have a maximum frequency

- Aperiodic: tasks whose frequency and timing cannot be predicted

2.2. Real-Time Specification for Java (RTSJ)

As it was discussed in chapter 1 Java SE or enterprise edition(EE) can be used for wide spectrum of applications, which performance depends on concrete optimization of JVM for concrete application. This Javas versatility and the WORA principle causes, that Java is not capable of fulfilling the basic requirements for usage in real-time environment. Running code within virtual machine(VM), using garbage collection and other things makes running applications unpredictable. Predictability is crucial for satisfying deadlines and precise timings, which are critical parts in real-time application. So the main problem of Java is unpredictability. Sources of unpredictability in application written in Java SE [3] :

- Operating-system scheduling
  The JVM creates threads which are scheduled by the OS. It causes time delay, when OS must react to an action of JVM. So the JVM is dependent on capabilities of OS. OS must fulfill scheduling and latency guarantees demands, through providing a robust priority-based scheduler, program defined low-level interrupts and high resolution timer.

- Priority inversion
  Priority inversion is a well known problem in applications with different priorities of threads. There are three threads with high, medium and low priority. Using lock on shared resource between two threads running on high and low priority can cause it. The low-priority thread is holding a lock at the moment when the higher-priority thread needs it. The medium-priority thread is started. The medium-priority thread did not use the shared resource and preempts the low-priority thread so it is scheduled first. Then the medium-priority runs till end. The high-priority threads priority is downgraded to the same priority of the low-priority thread and the medium-priority thread ran before the high-priority thread. This can cause deadline misses of high-priority threads, which are the most important for the application.

- Class loading, initialization, compilation
  The Java language has lazy initialization of required classes. The initialization of a required class occurs when an application first uses it. Instantiations of objects through constructors, which can contain some user code, causes jitter and a variance in latency. Classes can be lazily loaded, which means they can be stored on disk or somewhere on the network. This can cause delays when a class is referenced firs time. Methods are compiled to native code only when they are frequently executed and if it is faster in whole to compile them and use them compiled. But when class is compiled it still be the subject of re-optimization in future. In normal application is the use of just-in-time (JIT) Hotspot compiler one of Java benefits, but in real-time view brings problems.

- Garbage collection/collector (GC)
  GC is major source of unpredictability in all programming languages, which use it. The main problem is the "pausing the world". Application threads are stopped and GC frees the memory. Pauses have bad use time constraints. Because of it system cannot comply with needed requirements. There are more implementations of GC and they are better and better, but in the matter of real-time application it is not good enough. It is true, that exist real-time GCs suitable for some applications but not for all. When GC preempts all the application threads – a deadline miss can occur.

- Application itself

The source of unpredictability could be wrong written application. Programmer have to be careful to use third party libraries, which may not meet the real-time requirements. There is a problem with thread priorities. Java has thread priorities, but the guarantees of JVM are too weak, so they are not used so much. Threads compete for resources, which can cause unpredictable behaviour of an application. In Java applications we almost know, how long will a operation last, but when it is longer than expected it can cause waiting of other tasks. This behaviour brings greater demands on programmers. They have to do additional testing and tunning to achieve good results.

- Other system activities

  In systems are other high-priority tasks than the real-time applications. There could be more real-time applications and hardware interrupts. This influences the predictability and timing precision in our application.

Java have to deal with these performance problems. There are more possibilities of avoiding these problems and RTSJ offers some solutions. RTSJ contains specification of API and VM, which is aware of above sources of unpredictability. Specified in the [7] :

The programming environment must provide abstractions necessary to allow developers to correctly reason about the temporal behavior of application logic. It is not necessarily fast, small, or exclusively for industrial control; it is all about the predictability of the execution of application logic with respect to time.

RTSJ is designed to support both hard and soft real-time applications. Among its major features are: scheduling properties suitable for real-time applications with provisions for periodic and sporadic tasks, support for deadlines and CPU time budgets, and tools to let tasks avoid garbage collection delays.

2.2.1. Scheduling

The basic requirement for thread scheduling in realt-time JVM is real-time scheduler implementation. The specification itself does not contain concrete algorithm, which will be used for scheduling. There is only two requirements on default scheduler – it has to be priority-based scheduler and it must support at least 28 unique priorities. The scheduler may not be part of the JVM implementation when the OS scheduler satisfy required needs.

RTSJ brings new model of application development with real-time requirements. In Java SE the smallest unit of scheduling is java.lang.Thread. RTSJ is going further with this abstraction. All instances of schedulable entities in RTSJ scheduling framework implements **javax.realtime.Schedulable** interface. This interface extends java.lang.Runnable interface. Thank to this abstraction is possible to ensure expected behavior and determined requirements of schedulable objects with respect to time. It is possible through set of characteristic, which can be set to schedulable objects such as release time, period and priority. Big advantage of this abstraction is, that the objects, which are allowed to be controlled by scheduler are not only threads.

2.2.1.1. Schedulable objects

All implementations of RTSJ must include four classes, which implements the Schdeulable interface:

- RealTimeThread (RTT)
- NoHeapRealtimeThre (NHRTT)
- AsyncEventHandler (AEH)
- BoundAsyncEventHandler (BAEH)

The relations between classes are described in following figure.

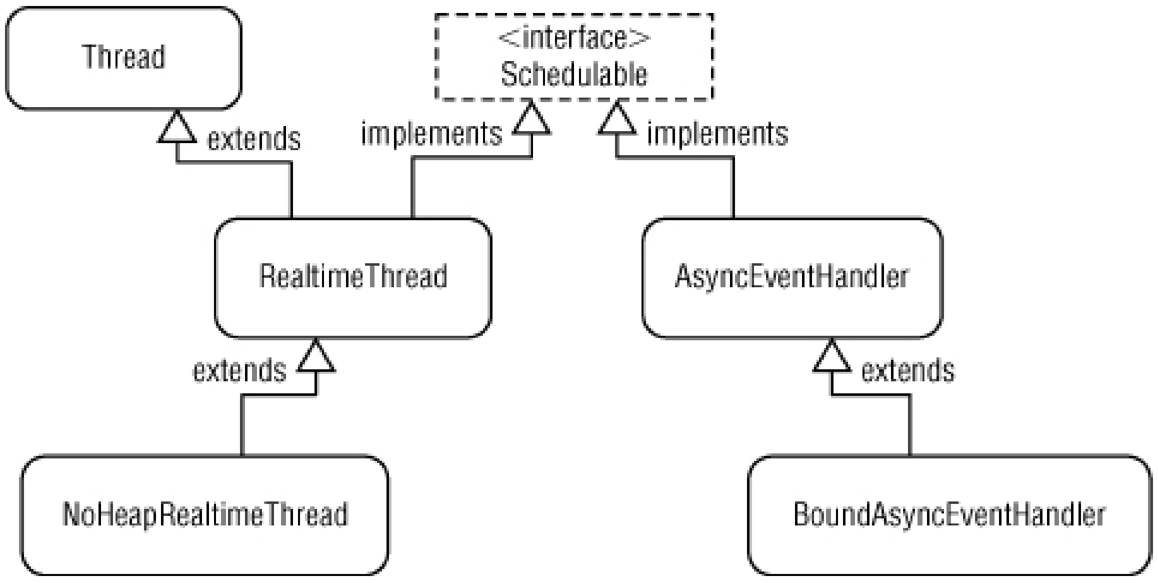


Figure 2.4. The class diagram of Schedulable objects from [5]

Schedulable objects have set of parameters serving for achieving desired behavior, which are used by scheduler:

- Priority parameters
  These parameters set the schedulable objects priority.
- Importance parameters
  When system is overloaded and there are two threads with same priority, importance parameters are used as second metric to distinguish which thread is more important.
- Release parameters
  These parameters are used for defining the type of schedulable object according to task types. They contain of start time, release cost and handlers for missed deadlines

and cost overruns. They use relative or absolute High-ResolutionTime object for setting the time parameters. Cost is used in feasibility analysis, which may not be implemented.

- Periodic parameters
- Aperiodic parameters
- Sporadic parameters

2.2.1.2. Real-Time Scheduler

In the scheduling framework is scheduler represented by abstract class **javax.realtime.Scheduler**. Subclasses of this class contain concrete implementation of scheduling algorithm. The required priority-based scheduler object for all RTSJ JVM implementations is PriorityScheduler. This object is singleton and reference can by obtained by calling static instance method. Besides of scheduling controll the scheduler provides informations about possible range of priority values, priority of a concrete thread. Scheduler has methods for calculating and recalculating the feasibility of actual schedule. This allows adding new Schedulable objects into running application system. Scheduler performs feasibility checks while adding new object to schedule and can trigger asynchronous event.

2.2.2. Memory Management and Models

Because GC is an important part of Java language but can not be used unchanged in real-time system. There are more approaches for GC and every type is suitable for concrete usage. This leads to dividing the memory into areas with different behavior. RTSJ specifies four classes which extends the MemoryArea abstract class. RTSJ defines two special memory regions.

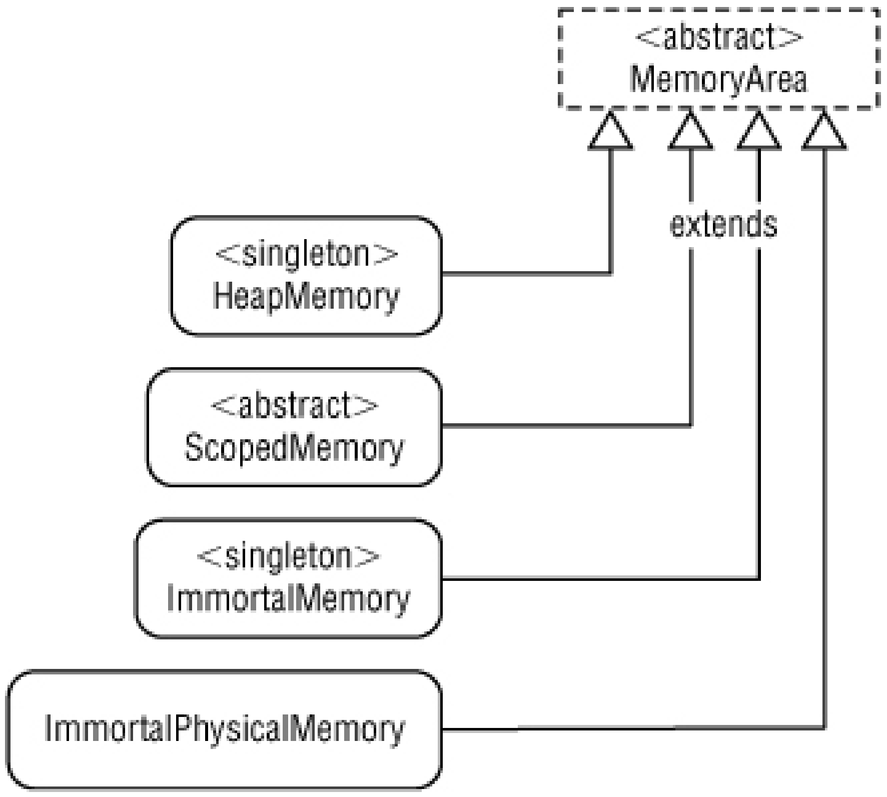


Figure 2.5. The class diagram of MemoryArea objects from [5]

2.2.2.1. Heap

Heap is referenced as a singleton and represents free memory, which is used for dynamic allocation and automatic reclamation. Real-time VM heap is just like standard VMs heap. It means that is garbage-collected and pauses can occur. Real-time JVM may not include implementation of GC acorrding to RTSJ. But there are requirements for the way of implementation of included GC. It is for better predictability of how GC impacts the application. Heap can be used by both java.lang.Thread (JLT) and RTT threads.

2.2.2.2. Scoped Memory

Scoped memory (SM) is memory created during development. Programmer can set its starting size and optionally maximal size. Setting the maximal size of SM is a protection against consuming all free memory by one thread. Schedulable objects which can run within SM are only RTT and NHRTT threads. SM object accepts Runnable class as constructor parameter, which is executed within this memory. The purpose of SM is for allocation of objects with know lifetime, especially for objects created and used only in one period of task processing. GC do not affect SM. The memory is reclaimed when no references to SM exists. This operation is quick and bounded in terms of time.

2.2.2.3. Immortal Memory

Immortal memory is according to its name created at the startup of JVM and objects in it exist for the life of whole JVM. Allocation and reclamation of memory is almost the same as in C based languages (using malloc() and free()). So it means that, IM uses static allocation and it is not influenced by GC. Objects allocated in IM can be referenced from anywhere of real-time application, and they can reference objects anywhere except the objects residing in SM. Schedulable objects can run within IM. The reason for usage of IM is to do allocation ahead of time avoiding dynamic allocation. Managing immortal memory requires greater care than managing memory allocated from the standard heap, because if immortal objects are leaked by the application, they could not be reclaimed.

2.2.2.4. Physical Memory

RTSJ has resources for low-level physical memory usage. Most real-time applications do not need this function. This type of memory is used for direct memory access (DMA) and for communication with external devices and IO operations. It makes sense to use it only in specific cases. Types of representation are LTMemory and VTMemory - memories with linear time of allocation and variable time of allocation, which extends SM. They have extension for physical types – LTPhysicalMemory and VTPhysicalMemory.

2.2.3. Real-Time Garbage Collection

Real-time garbage collection(RTGC) may not be the part of RT JVM. RTGC never blocks critical high-priority threads and works concurrently and incrementally. It does not guarantee deterministic behaviour of all threads. RTGC used in Sun Real-time system is implemented to provide hard-realtime behavior for critical high priority threads. These threads are never preempted by RTGC. For other threads with decreasing priorities is achieved soft real-time and non real-time behaviour. The garbage collection is optimizing itself at runtime. Basically is GC using auto tunning for better performance. Tunning parameters can be set manually before the start of JVM. Parameters specification depends on concrete RTGC implementation.

Example of parameters used by Oracle (Sun) RTS Java implementation. RTGCNormalPriority - set the value of GC threads priority. Threads with higher priority are not preempted by GC. NormalMinFreeBytes - set the value of minimal available free memory threshold, after which GC begins its work. RTGCNormalWorkers - sets the number of parallel threads, which do the garbage collection. This ensures enough resources for critical and higher priority threads of application. This is an advantage especially on multi-processor systems when a small subset of processors count is set as the number of RTGC threads. Similar parameters are defined for another modes of RTGC.

RTGC has function modes according to the size of allocated memory. This RTGC is used in Sun Real-time system.

- Normal Mode – startup memory threshold - RTGC runs on normal priority, only non real-time threads are blocked
- Boosted Mode – boosted memory threshold - RTGC runs on boosted priority, non real-time threads and normal priority real-time threads are blocked
- Determined Mode – GC occurs when memory reaches critical low threshold. All threads except high priority RTT and NHRT are blocked.

For further reading and information about GC and tuning is in materials about concrete JVM implementations or read [4][5]

There are more command line parameters for real-time JVM not only for RTGC, but they are defined by the real-time JVM itself. With this can be tuned the behaviour of threads, asynchronous control,memory management, interpreter and compilation.

2.2.4. Real-Time Threads and Handlers

In RTSJ the 4 main classes (RTT, NHRTT, AEH, BAEH) are subjects of scheduling and implements the Schedulable interface. It means, that they have more precise scheduling parameters than standard Java SE classes. This classes are the most important ones for developing real-time applications with RTSJ. Standard JLT threads can be still used, but the real-time task are executed within real-time threads a handlers. Some basic explanations of their function is needed for good understanding of benchmark implementation. The main principles do not much differ from JAVA SE, but especially the use of scheduling, release and next RTSJ parameters does using this classes a little more complicated. So they should be mentioned for better understanding of benchmark implementation.

Besides the threads supports the RTSJ events handling, it is not used only to deal with asynchronous events but for periodic events too. Schedulable objects are using event handlers for deadline miss handling. It is very important for hard real-time system to take appropriate action if a deadline miss occurs. The results of code execution within a handler or thread are the same. There is not a definition when is better to use thread or handler. Both has support for periodic operations through PeriodicParamateres for Threads and timers for event handlers.

2.2.4.1. RealTimeThread and NoHeapRealtimeThread Class

Class definition from RTSJ JavaDoc [4]: "Class RealtimeThread extends Thread and adds access to real-time services such as asynchronous transfer of control, non-heap memory, and advanced scheduler services."

Both classes can take as constructor parameter Runnable logic class or developer can create own classes extending these thread classes. There is no difference in code execution. RTT and NHRTT threads are the right place where to run critical application code. The RTT thread is almost same as JLT thread. The NHRTT has in its name, that it can not access the heap. This Schdedulable classes can take as parameters previously discussed scheduling parameters(priority, types of threads), memory parameters(memory within the thread runs), Both of classes have these semantics

- Dispatching – priority-based dispatching, according to Schedulable interface
- GC – The relation between RTT and collector is strictly based on their priorities. Collector has its own priority and preempts all the RTT with lower priority threads and all JLT threads. NHRTT is
- Synchronization – is basically same as JAVA SE, the priority inversion problem is solved by implementing priority inheritance
- Periodicity – for thread and its periodic behaviour is implemented method waitForNextPeriod(), which ensures that task is released precisely on the next`s period time, regardless on variable processing time of task
- Interruption – asynchronous transfer of control is used through the usage of AsynchronouslyInterruptedException (AIE) and Interruptible interface

Mentioned semantics could be discussed in more detail, but this is not a textbook of programming in RTSJ. Important functions will be explained when they are used in benchmark implementation. Start to using RTT is very simple, because it was one of the reason of using RTSJ for programming real-time applications. I describe simple usage of threads without defining all parameters.

```
/* Code example of simple instantiation of RTT thread and starting */

RealTimeThred rtt = new RealTimeThread() {
  public void run {
    //logic
  }
};
rtt.start();
```

The main difference from the view of developer is in the setting the memory within must the NHRTT run. There is more possibilities than to specify the MemoryArea in constructor, but it can be found in referenced literature.

```
/* Code example of simple instantiation of NHRTT thread within Immortal memory and starting */

NoHeapRealTimeThred nhrtt = new NoHeapRealTimeThread(null, ImmortalMemory.instance()) {
  public void run {
    //logic
  }
};
nhrtt.start();
```

More examples, samples of code and design patterns can be found in [8] and [5]

2.2.4.2. AsyncEventHandler and BoundAsyncEventHandler

The base element of asynchronous event handling is the abstract class AsyncEvent(AE), which itself represents occurrence of important system event. Concrete instance of AE can be associated with AEH and vice versa. AEH executes code when the fire() method of associated AE instance is called. The AEH has support for time events, which allows AEH execute periodic tasks. This support is accomplished through the usage of timers. RTSJ contains PeriodicTimer object, which periodically calls the AE fire() method. It is more obvious from the class diagram of AsyncEvent. The PeriodocTimer itself extends AsyncEvent and calls periodically the fire() method on itself. It is important to mention, that there is only one internal thread, which handles timer expiration events. This can cause release delay, when two events has same release time, because the releases are serialized.

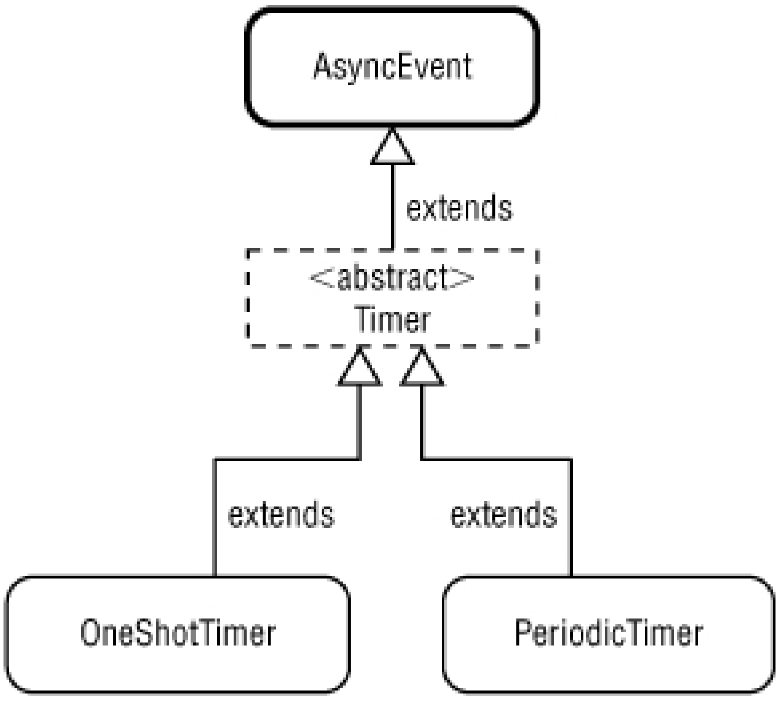


Figure 2.6. The class diagram of AsyncEvent objects from [5]

Class definition from RTSJ [4]: "An asynchronous event handler encapsulates code that is released after an instance of AsyncEvent to which it is attached occurs. It is guaranteed that multiple releases of an event handler will be serialized. It is also guaranteed that (unless the handler explicitly chooses otherwise) for each release of the handler, there will be one execution of the handleAsyncEvent() method...."

AsyncEventHandler can use RTT threads or NHRT for execution of logic contained in object, which implements Runnable class. AEH can be extended in the same manner like real-time threads. Then is the logic part of handler itself. There is nothing such NHRTT. Constructor of AEH accepts boolean parameter which defines usage of heap.

```
/* Simple code example of AsyncEventHandler usage */

public class Logic implements Runnable {
  public void run {
    //some execution code
  }
}

AsyncEvent event = new AsyncEvent();
Runnable logic = new Logic();

AsyncEventHandler handler = new AsyncEventHandler(logic):

event.addHandler(handler);//register handler to listen on this event

event.fire();//do logic in registered handlers
```

The difference between AEH and BAEH is not obvious from the name of BAEH or from looking into code. Real-time application often must process a lot of different events by different handlers and is not possible to connect all this logic with concrete running RTT threads. The RTSJ contains requirement, that it must be capable of having a big amount of handlers ready for processing events. This is accomplished thanks to dynamic assigning of RTT threads to fired events. RTT threads are in pool and they are assigned when the fire of event occurs. The BAEH is permanently bound to dedicated RTT thread which is created and handle all fired associated events - this behaviour does not suffer from latency and jitter opposite to AEH. But usage of BAEH is same as for AEH. The source code sample would differ only in usage of BoundAsyncEvetHandler class. Because of that i do not show code example for BAEH.

Chapter 3

Benchmark Description and Metrics

Benchmark in the terms of computer science is a set of tests used to compare hardware and/or software performance. Tests consists of different measurements. In tests are measured appropriate metrics, which are important indicators of how good tested system is. The purpose for running tests is to obtain valid data for comparison. Results of the tests are mostly measured data of different metrics. Then are resulting data represented by charts for more simple interpretation. For comparison of benchmark results is crucial to know exactly what is test designed, it means that the results of benchmark are dependant on the environment, where the benchmark is running. (OS, Hardware configuration)

The motivation for creation of a benchmark was the lack of information about performance of different real-time JVM implementations. Real-time Java is used only by small group of developers for science purposes or by companies to develop closed source systems especially in financial sector. JVM implementations are often not easily obtainable, which is one of the reasons for small amount of real-time JVM benchmarks. Vendors provides some information about their products, but it is not usable for direct comparison. It is caused by small spread of real-time Java usage. Selling real-time JVM is good business so the benchmarks are mostly a subject of business, too. There are some free and open source RTSJ benchmarks but the count is very low. Examples of other benchmarks are CDx - RTSJ Benchmark, jPapaBench, Sumaradu, Scheduler Test and other.

Benchmark developed in this thesis is a software benchmark. The important thing about software benchmarks is the possibility for changing the difficulty of particular tests and compare the results with each other. The behaviour of JVM implementation is specified by RTSJ, but it does not mean, that they are totally identical. Although benchmark the virtual machines, they are still software running on underlying OS. Generously benchmarking is very difficult task, especially when good metrics are not known. Determination of usable metrics is the most important part when writing benchmark application.

3.1. Robot Position Localization

Every benchmark has a set of tests, which measure needed metrics. These tests have to be based on some usage of tested software. So the first step was to choose an example application and adjust it for an implementation in RTSJ. My advisor is a leader of an

Eurobot team from Czech Technical University named Flamingos. This had led him to choose an application from the environment of robot competitions. He wanted to know if it is possible to use real-time Java for real-time computing on a robot. The benchmark consists of only small part of robot control - localization. The robot have to know his position on the playground to move successfully and perform competitive tasks.

This part of program used for robot localization is originally written in C language. The localization of the robot is based on Monte-Carlo (MCL) Localization algorithm, which will be shortly described.

My first task was to rewrite the C code to Java code. The C algorithm source codes are included in the jmclbench repository.

3.1.1. Monte-Carlo Localization

MCL is a type of robot position localization method for solving localization problems. MCL is often used for localization of the mobile robots such as in this case. The localization algorithm used by this method is relative simple for implementation. Another advantages of this method are the usability in wide range of localization problems and good performance. Thanks to this MCL is very popular and used especially by developers specialized in robotics.

Robot position localization method is used to estimate the actual position of a robot in a given time-step. Two main parts are global position estimation and local position tracking. Global position estimation is ability to determine the robot position on a given map. Local position tracking is estimating the robot position from the motion of robot over time. The next estimated position is somewhere in the likelihood of the current estimated position.

Estimating the current state of robot, given knowledge about the initial state and all measurements up to the current time. The state vector contains the position and orientation of the robot. This estimation problem is an instance of the Bayesian filtering problem.In the Bayesian approach, this probability density function is taken to represent all the knowledge we possess about the state of the robot, and from it we can estimate the current position. [9]

Basic principle of the robot localization methods is recursively solving two main subproblems. This two subproblems correspond to two main phases of robot localization algorithms:

- predict phase: In the first phase we use a motion model to predict the current position of the robot in the form of a predictive probability function, taking only motion into account. We assume that the current state is only dependent on the previous state ([13]) and a known control input and that the motion model is specified as a conditional density.

- update phase: In the second phase we use a measurement model to incorporate information from the sensors to obtain the probability. We assume that the measurement is conditionally independent of earlier measurements, and that the measurement model is given in terms of a likelihood . This term expresses the likelihood that the robot is at location of position given that measurement was observed. The density over probability of position is obtained using Bayes theorem.

The knowledge of previous robot state is provided as density of probability of previous estimated position.[9]

MCL is sampling-based method in which is the density represented by set of samples(particles) that are randomly drawn from it. Representation as particles allows the usage of the particle filter. In each time-step is generated set of particles from density of probability.

1. The creation of new set of particles is based on the previous estimated position(previous set of particles). The motion model is applied to each particle by sampling from the density. A new set of particles is obtained.

2. According to measurement are weighted each particles. The particles which are closer to measurement has greater weight.

3. The particles are resampled. Higher probability particles are selected which results in creation of new set of particles.

These parts of algorithm are repeated recursively for each time-step.

Pieces of Informations needed for updating the position are provided by the sensors. Each sensor update changes the probability of correctness for each particle. This is performed using statistical model of the sensors and Bayes theorem. Similarly, every motion the robot undergoes is applied in a statistical sense to the hypothetical configurations based on a statistical motion model.

Domain of possible inputs is defined by the robot playground and its parameters are described later in this chapter. In this method a large number of hypothetical current configurations(positions of a robot) are represented as set of particles. Initially are these particles randomly scattered in configuration space(robot playground). Set of particles is generated according to domain distribution or the starting position is known.

The complexity of calculation depends on the count of particles as well as the precision of estimation. With rising count of particles rises the time of computation as well. This is very important for benchmark, when we can choose a set of particle count for testing.

The main advantages of MCL according to [10]:

1. In contrast to existing Kalman filtering based techniques, it is able to represent multimodal distributions and thus can globally localize a robot.

2. It drastically reduces the amount of memory required compared to grid-based Markov localization and can integrate measurements at a considerably higher frequency.

3. It is more accurate than Markov localization with a fixed cell size, as the state represented in the samples is not discretized.

4. It is much easier to implement.

In the view of benchmark development are important the point 4, which do this method suitable for use in a benchmark application. Easy implementation means easier understanding for users of benchmark.

3.1.2. Robot Playground

The Robot's playground is a rectangular area 3 meters long and 2 meters wide. On the figure below are the letters W and H. These letters corresponds to W – width and H – height but it is only for the purposes of mapping robot position into concrete configurations. The position of robot is mapped on the playground by defining Cartesian coordinates. Left side of rectangle represents the y axis and bottom side of rectangle represents the x axis. Units on axes are in meters. X axis is limited from above by 3 and y axis is limited from above by 2, which corresponds with the rectangles parameters. The origin is located on intersection of the axes in the left bottom corner of rectangle and has coordinates [0,0]. This characterizes the first part of state space input domain.

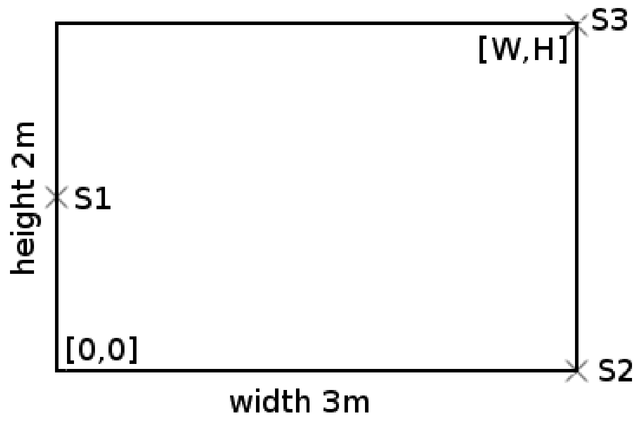


Figure 3.1. Robot playground

The robot can move forward only in one direction like a car so it has to turn right or left. It causes the need of knowing orientation of the robot. The orientation is defined as an angle in radians from the x axis. A value of 0 corresponds to the direction the robot motion perpendicular to the right side of the playground. The angle value is the second part of state space domain. The position of robot is characterized by the coordinates(x,y) and the angle. So we will work with three-dimensional state vector x = [x,y,θ] These data represents the members of particle set used by MCL.

Robot uses for localization one laser beacon and sensing the reflections of this lasers beam. This beacon is placed on top of the robot.

- Laser reflection sensing
  The robot detects only the reflections from three beacons located on the playground. The position of beacons is drawn on figure above(S1,S2,S3). This sensor does 3 rounds per second.

- Odometry sensing
  The robot has sensors, which can measure rotation of the wheels. Measured data represents maximal traveled distance. Deviations may arise when robots wheels slip or robot moves over uneven terrain.

Position of the robot is calculated using data from the laser sensor. Positions of the beacons are known in advance. This allows to estimate robot's position and rotation from the set of particles. The robot records the angle of the laser when receiving the reflection from individual beacons. It is not known from which beacon is the reflection. From position of the concrete particle are calculated the angles between robot and all beacons. After this is calculated the weight of particle by comparison of the calculated angle with measured angle. From the difference between angles is generated part of weight using probability distribution function(Gaussian) according to angles similarity. This is done for each beacon and the parts of weight are summed. Then the particle with best weight is selected as estimated position. After this is the set of particles resampled. It means that the particles with low weights are thrown away and they are replaced by particles with higher weight.

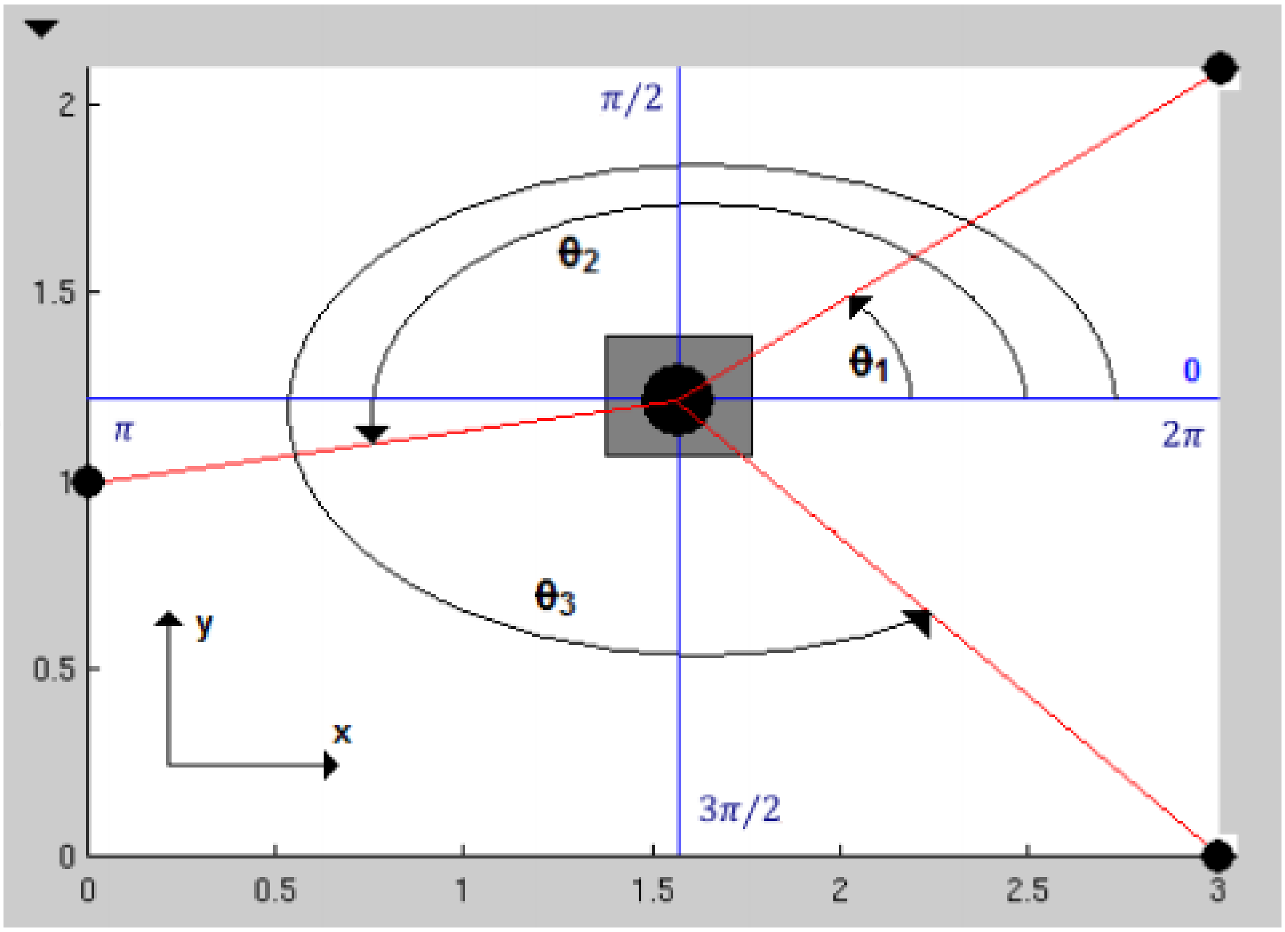


Figure 3.2. Description of parameters from [12]

3.2. Benchmark Description

As mentioned in the previous chapters, benchmarks are usually composed of several different tests. Because the benchmark's main purpose is to compare the performance of the real-time JVMs, testing methods are chosen in this context. MCL, as such, does not allow much modification, which have a direct impact on the real-time behaviour. This is mainly due to the calculations that are not dependent on time. The purpose of benchmark is not to tune and optimize the MCL algorithm for use on real-time JVM implementations.

Every JVM implementation have to run the same code on same machine and same OS, so the conditions are same. The way of creating the benchmark using MCL algorithm is based on its requirements. The area for benchmark development is above between the phases of MCL algorithm. Receiving data from sensors is the main part where could be benchmarking done. The benchmark have to deal with the control of algorithm and control of data receiving. And the benchmark have to provide simulation of events which represents the incoming data from sensors.

The most important features for choosing MCL algorithm are:

- lot of allocation and deallocation of memory

- resources shared between threads
- real-time requirements

The data used for localization are meaningful only when they are measured. The time is not used in this method's computation. But it is obvious that the correct sequence of data processing is critical for successful functioning of localization and as well as the time is one of the most important parts, which can be used in benchmark.

The robot consists of estimated position and set of particles. This set is changed by each run of algorithm. So in this is a lot of allocation and deallocation of memory.

In terms of real-time programming is one iteration of a localization algorithm represented as set of different task. This localization of robot is divided into 2 tasks. Each task covers part of the MCL according to the phase described in previous section. These tasks have to share the robots data. This creates the requirement for separated independent threads which represents tasks.

Tasks according to phases of algorithm:

- predict task

  This task performs predictive part of the algorithm. It uses data retrieved by the odomoetry sensors. These data represent increments to the individual parameters of the robot position. Increments are added to the estimated position without noise and to all the particles with random noise to represents sampling from probability distribution. This task should be executed every time robot retrieves data from the sensors. Data are retrieved every 50 milliseconds. This means that it is a recurring task with fixed period. Next part of this task is recording the estimated position and processing times.

- update task

  This task performs update and resample part of the algorithm. To update task are used data measured by the laser sensor. These data include rotation angle and time. The measured angle is used to estimate new position of robot. After this is set of particles processed by resample method. Time of receipt of data is dependent on the current position of the robot. This means that this task is recurring, too, but not have defined period. Next part of this task is recording the estimated position and processing times.

In the terms of real-time programing the predict task is a periodic task and the update task is an aperiodic task. Task have to be performed when an event occurs.

In this application, an event is represented as data is received from the sensor. Following figure shows diagram with possible occurrences of events and time of processing.

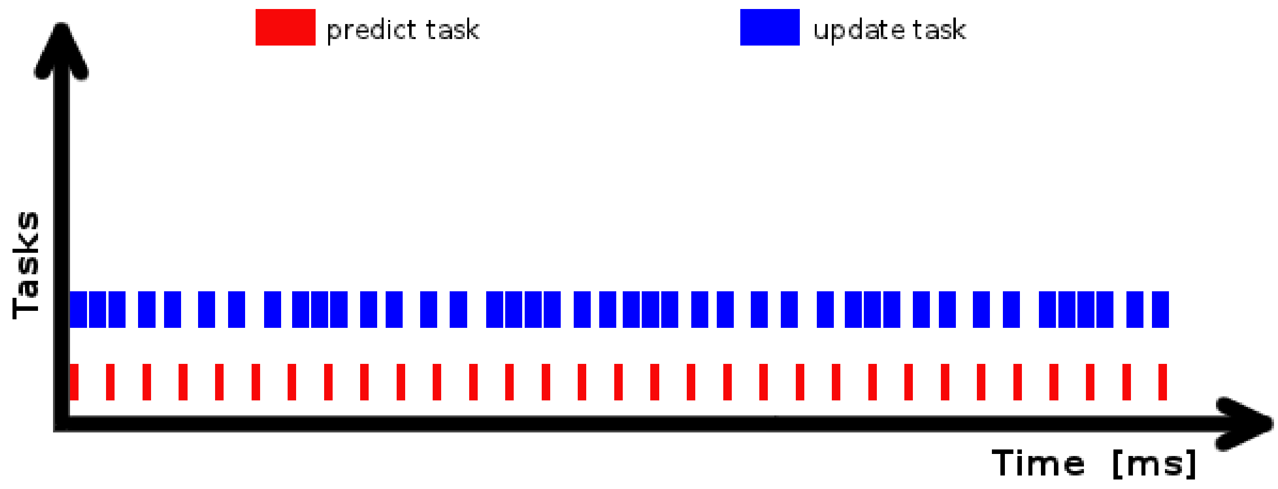


Figure 3.3. Occurrence of events

The tasks are not independent because they are performed over the same set of samples. It is possible that it happens a new data should be processed when the previous task has not been completed yet. This can happen both at a different types to the same types of tasks. With this application must somehow cope. Various approaches to solving this problem are to use different policies. In the application are referred to as tests.

3.2.1. Skipping of Jobs

One way to solve the problem of overlapping events is skipping. When it comes to the fact that the event is received if not completed the previous task. The application have to decide what do with this new event. Whenever a new event is received application logic have to verify whether there is still some tasks running. If a task is still being processed, the new event is simply ignored. Otherwise when no task is running the event is processed as usual.

3.2.2. Termination of Jobs

Termination of events solves the same problem as skipping of events. But if a task is still being processed, the new event is not ignored. Currently processed task is terminated. And the new event is processed just as if the termination not occurred.

3.2.3. Format of Input Data

For the development of the benchmark application has to be known in advance the course of a robot's movement around playground. Without this data can not be proven the correct function of the localization. The real robot's position can be used for better understanding and evaluating the different JVM implementations. Data for benchmark was provided by modified simulation of real robot's movement. The simulation was the subject of this paper [10] and method used in benchmark comes out of it.

The data are provided in simple text files.

- Odometry data [*time x-increment y-increment angle-increment*] In first column is time when the increments of parameters are in following columns.

  ```
  4.0000000000000000e+00 2.9508011417671016e-02 1.8086984431529840e-05 -1.2259031562840694e-03
  ```

Example 3.1. Line of odometry data example

- Laser data [*time angle*] First column is time of receiving laser reflection and second column is the angle between the robot heading and the reflection from beacon.

  ```
  3.0873606372492779e-01 5.8195376981787801e+00
  ```

Example 3.2. Line of laser data example

- Real position data [*time x-position y-position angle*]
  Real position data are formated same as the odometry data and file looks similar. The difference is, that the values are not additions but actual values of robot parameters.

3.3. Metrics

Metrics are the most important part of the application in terms of interpretation of benchmark results. Defining metrics was not easy due to the fact that have to be verified the relevance and validity of metrics in the context of real-time applications. It is then possible to interpret the measured data in different ways to calculate them into various other metrics.

Thanks to experience with the development of various applications and knowledge in the field of computer science was possible to predict what metrics might be useful. Because for real-time applications is the most important constraint satisfaction on the precise timing, are interesting metrics the latency and delay. Another important indicator to compare various implementations is the number of unfinished tasks. The MCL localization itself provides the simulation of the robot motion from which is obtained the trajectory. The results of simulations can be compared among themselves. The trajectory consists of a set of points that must be adjusted for easier and exact comparison. Other metrics are represented by results of statistical processing of different simulation results.

3.3.1. Deadline Miss (missCount)

Processing tasks in the context of the benchmark have a predetermined relative deadline. For the Duration of the task processing depends on complexity of computation. The complexity is defined by the number of particles. It is not precisely known how much time task processing will take. Restrictions on the completion of tasks is determined by the arrival of another event. The deadline for task is different for each type of task.

The relative deadline for predict task and is 50 milliseconds which is the fixed period of odometry sensor reading. As mentioned before, the robots laser turn at 3 rounds per second. The maximal and minimal angles between beacons are reached when the robots position is on position [0,0]. The maximal angle is between beacons S1 and S2 and the value is 270°. The minimal angle is between beacons S2 and S3 and value is approximately 35°. The maximal relative deadline for laser is 250 milliseconds and the minimal relative deadline is 32,407 ms.

At the beginning of one test run is deadline miss count equal to zero. When is a task terminated or skipped the count is incremented by one. The comparison of deadline misses provides good view of the JVM implementations performance and it is one of the most important metrics.

3.3.2. Latency

It is possible to measure the duration of the task processing. The time when the task started to be performed is recorded. Then the time when is job processed is recorded. The difference between these two times is the duration of the task. Each of the two types of tasks has a different duration. Tasks are processed repeatedly, so there is a large amount of data. From these data, can be calculated the average duration of a task, which is also an interesting indicator. The processing time is smaller, the more efficient implementation of the JVM.

3.3.3. Jitter

The simple definition of jitter is irregular time variation of period signal properties, such as small, unpredictable delays in scheduling. In the context of benchmark development it is delay of dispatching the task. The precise time when the task have to start is known. The real start of the task is measured and compared to he required start. The difference between real and required start time is the value of jitter. Ideally, both times would be the same and value of jitter would be zero.

3.3.4. Average Error of Estimated Position

The simulation is possible by using previously known position of the robot trajectory. There are precise data about real robot position in given time corresponding with the times of odometer sensor reading. From these data we can determine the deviation of the coordinates and angle from the real position.
In each step, which is adjusted estimated robot position is available set of particles. From the set is selected new position of the robot and this selection is based on probability function.

Chapter 4

jMCLBench Application

Designing and programing an application is very complex process. This process is divided into some steps and each step needs different approach. Usually, analysis of the problem comes first, but I had to learn something about real-time environment and real-time programing. I described it briefly before. After analysis comes implementation. It is very hard to write all parts of the application at once. So I have to write simpler versions of application and proceed slowly to final version. Of course, it is important to test and optimize single parts of the code for better performance and application robustness. Whole implementation is set of iterations of adding new functions, testing and debugging.

The application consists of different parts. I have to mention building the application, because I want to distribute application for public. In this case the development is a little bit easier because there is no need for any graphics user interface (GUI). On the other hand, a very important part is the data visualisation, which is not very easy to implement. The application can be ran in the integrated development environment or in a built state from console. The most important part is the representation of the data. It is implemented by using graphs.

4.1. Technology

An integral part of software development are supporting programs. Each programmer has a favorite tool to use as well as I. When developing applications, very important is the choice of programming language. The subject of this work implies the use of the Java. Developing of applications in the Java language is not very complicated. I personally have much experience with Java programing, so I have favourite tools. It does not mean, that this tools are the only right ones. I prefer open source software. As my OS am I using GNU/Linux, the Ubuntu distribution. I do not have much experience with real-time programming or using real-time Java.

For realization of single parts, I will try to mention more programs, which can be used in similar way. There is a lot of software for doing same things. The choice of used software depends on the knowledge, experience and preference of the concrete program developer. Following section briefly describe technology that I used.

4.1.1. Integrated Development Environment

There is a lot of possibilities how to develop new applications. Important things for developer are integrated development environments (IDE). It is possible to write whole application in notepad-like application in shell terminal, but it is not so comfortable. I personally used free open source Eclipse Helios. Eclipse is one of the most popular IDE for developing Java applications. Like any IDE provides support for project management, controlling the code and debugging. Using plug-ins provide support for versioning systems. Other possibilities are the NetBeans or the IntelliJ IDEA.

4.1.2. Version Control System

One of the goals of this work is to publish benchmark application as an open source project. In open source development is very important the project management. First requirement of free open source project is the access to the source codes. Then everyone can see how the applications works. On dependency on licence type of concrete software, can be source codes or its parts used by developers in other projects. If someone is interested in similar subject, it can easily participate in development of this open source project, because he has everything, what he needs for it.

On the big and complex projects are there a lot of people working. Each of them working on different part of project, or small groups work on same parts. Everyone do changes in the source codes and there always has to be a function version of application. So for this purpose exist version control systems and versioning. Versioning is a way of storing history of all changes made to ordinary digital information. There is an evidence of changes implemented in single versions during the stages of open source project development. It is possible to version all types of files and data. The information about changes contains who, where and how modify which lines of codes. This full view of precise data state makes possible to return to an old version of application, when the new version contains errors.

The version systems do not store whole state of version, but they store the differences between single revisions. Tools like the diff is used for comparison. Information value is similar but the size of data is small.

Me and my advisor had decided to use git for jMCLBench development. We prefer remote communication so it was very important to use any version system. We can use the repository for publication of jMCLBench. I am familiar with CVS and SVN, but is exciting to try some new type - distributed version system. Source codes of application are not very complicated, but it is comfortable to have unlimited access to up-to-date source codes. We can share the results of benchmark using it. Some information about using git [15]

4.1.3. Types of JVM

The purpose of the benchmark is to measure the performance of particular JVM implementation. First I had to get some real-time implementation of the JVM. Implementation of real-time JVM are often paid, but for research purposes or for testing versions are provided under evaluation license or academic license.

The first option considered was the implementation provided by Java originator, Sun Microsystems. Now there can not be downloaded on the website stated that the evaluation program was terminated. Oracle focuses more on the JRockit JVM. Real-time version can be downloaded from http://www.oracle.com/technetwork/middleware/jrockit/downloads/index.html?ssSourceSiteId=ocomen, but this version is not used.

Another major player in providing real-time implementations of Java and Java support in general is IBM. Available for use is the hard and soft real-time version. Hard real-time version must run on the RTOS, while soft real-time version can run on standard operating systems. I could not find a link to download a free version that I used. Now, IBM offers WebSphere Real Time version 3 of its JVM, I used the version 2. It seems that it's only in the paid version.

The third implementation is tested JamaicaVM, which is proprietary but is easily available in limited release. This implementation I used longer than the time limit. Communication with the company aicas was smooth and my license was extended. Of course I had to use a standard implementation of Java for implementing the initial version.

List of used JVM implementations:

- Sun Java Real-Time System 2.2 (Oracle) - http://www.oracle.com/technetwork/java/javase/tech/rts-142899.html
- IBM WebSphere Real Time for Linux version 2 SR 3 (soft)
- IBM WebSphere Real Time for RT Linux version 2 SR 3 (hard)
- aicas - JamaicaVM http://www.aicas.com/jamaica.html
- Oracle Java SE (build 1.6.0_26-b03)

4.1.4. Work with Charts

Running the robot simulation produces large amounts of data. They are mostly numerical data. Interpretation of these data would be very difficult. Comparison of different implementations of the JVM would be almost impossible because values are located in different files and there are many of them. The solution is to display the resulting data in graphs.

Because I like to use very things connected with Java, I tried to find a suitable Java possibility for creating graphs. In the early days of development I have chosen JFreeChart. http://www.jfree.org/jfreechart/. Note that JFreeChart is a class library for use by developers, not an end user application. It provides support for many different types of graphs. Graphs can be exported into files as images (PNG, JPEG) as well as vector graphics (PDF, EPS, SVG). Another advantage is the use of a swing component for interactive viewing in an application, but which when used in the benchmark is not needed.

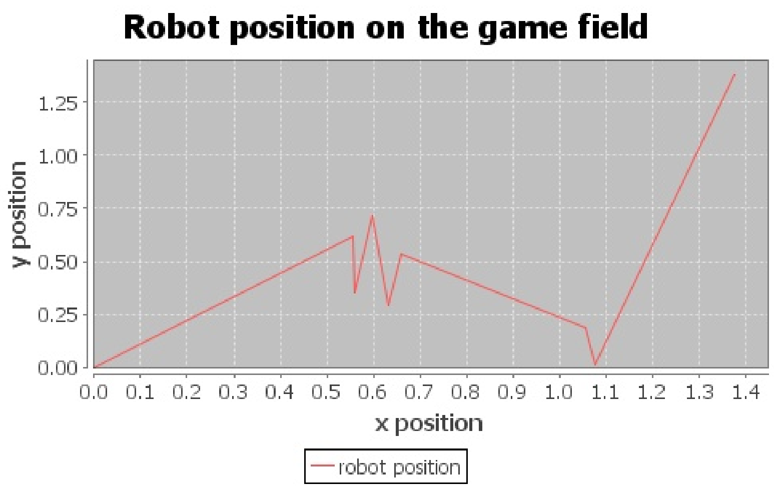


Figure 4.1. Example of chart generated using JFreeChart

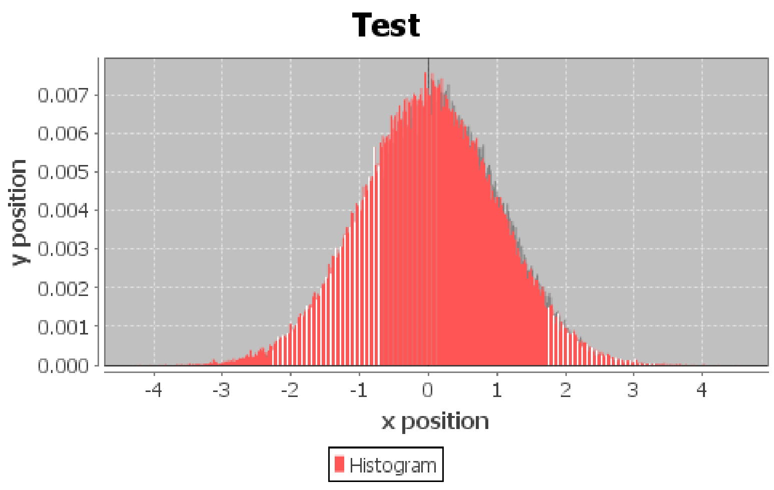


Figure 4.2. Histogram verifying Gaussian distribution.

On the recommendation of my advisor I chose as the main tool used to generate graphs the gnuplot. Gnuplot is command-line driven graphing utility supporting almost all platforms. Originally developed for interactive charting but supports export to a large number of output formats, including formats for printing and layout as the LaTeX. Gnuplot is distributed freely but source codes are copyrighted. Further information about gnuplot can be obtained from the this site: http://www.gnuplot.info/[1].

[1] ???

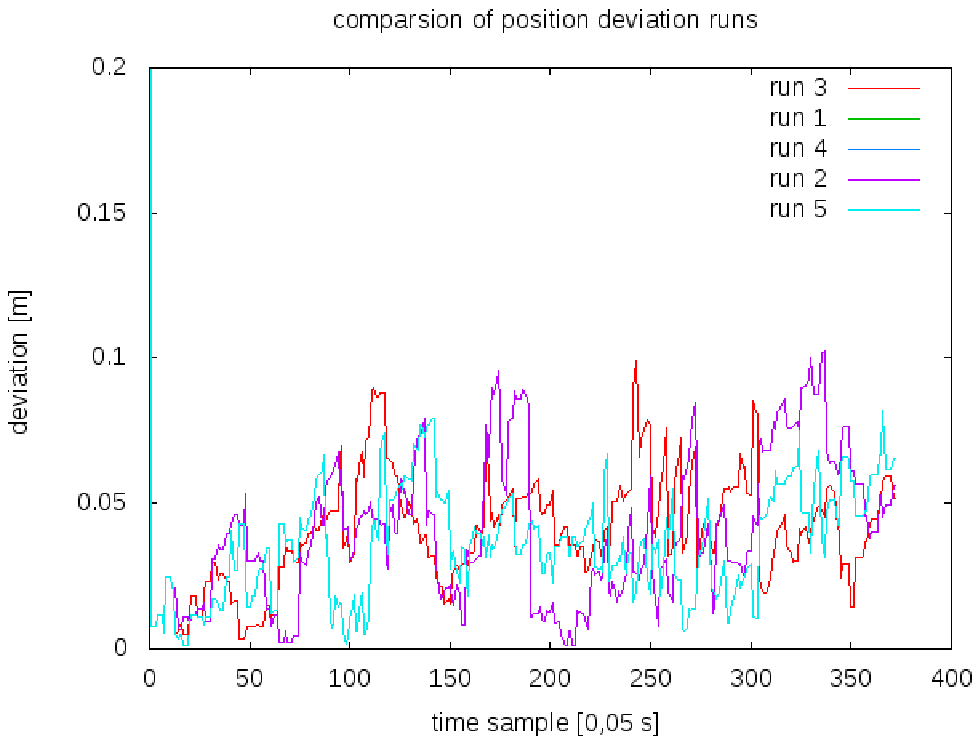


Figure 4.3. Example of chart generated by gnuplot

I got a recommendation for the use of R language for creating graphs. R is a free software environment for statistical computing and graphics. It runs on almost any operating system. But given that I did not know this programming language, so I did not use it. In addition, I had learned to work with gnuplot. Maybe I will use it sometime in the future.

4.1.5. Presentation Layer

The benchmark generates large number of diagrams, so is needed some way to compare the graphs of each other. Gnuplot provides the possibility to export multiple graphs into a single image. But this is only the possibility of static comparisons. Development of another software application used for comparison of graphs would be unnecessarily complex. I had to devise a simpler solution.

Each has a web browser, so I decided to use the power of WWW for the charts comparison. So technologies that I used for presentation layer are HTML and CSS. In order to do the comparison dynamically i used the scripting language PHP.

4.1.6. Real-time OS

RTOS is an operating system that guarantees certain capability within a specified time constraint. There are many of RTOS systems. Some of them are open source and even free of charge. Of course there are proprietary and payed solutions from different vendors. Support of different platforms is very variable. I chose RTLinux, because it is easily obtainable. There are more RTOS systems in use [2] some examples with supported platforms:

- RTLinux
  DEC Alpha, ARM, AVR32, Blackfin, ETRAX CRIS, FR-V, H8/300, Itanium, M32R, m68k, Microblaze, MIPS, MN103, OpenRISC, PA-RISC, PowerPC, s390, S+core, SuperH, SPARC, TILE64, Unicore32, x86, Xtensa
- Windows CE
  x86, MIPS, ARM, SuperH
- VxWorks
  ARM, IA32, MIPS, PowerPC, SH-4, StrongARM, xScale
- QNX
  IA32, MIPS, PowerPC, SH-4, ARM, StrongARM, XScale

The choice of OS used depend partly on the real-time requirements and partly on my preferences.In introduction I mentioned some RTOS, which are usable for running real-time application. Writing the source code does not have special demands on used OS. I personally use GNU/Linux as my main OS, specifically the Ubuntu 11.04 (natty narwhal) distribution. Running hard real-time implementation of the JVM is subject to the use of RTOS.

For normal work, of course, like most people I do not use RTOS. Before starting work on this thesis I do not have any experience with using RTOS. The GNU/Linux is easily turned into RTOS by installing the real-time linux kernel. Before I tried to compile a custom kernel, but for the older 32-bit version of Ubuntu. I am currently using 64-bit OS version, so I tried to compile a real-time kernel for it. I spent quite a lot of the time but I was not successful. Using information from the forums on the use of real-time kernel in Ubuntu I found that there is a precompiled package that is ready to be installed. I managed to run the real-time kernel on the 32-bit version of Ubuntu 10.04(Lucid Lynx).

Installation of real-time kernel in itself is not enough to run a hard real-time implementation of JVM. It was necessary to adjust the required dependencies, and even the dirty way by creating symbolic links to other versions of libraries. Specifically, linking particular concern libcap.so.1 library. Due to demands on OS scheduling capabilities has hard-real time implementations of JVM embedded tests of system. I have to solve unresolved dependencies. Real-time distribution which is often used to run real-time applications, OS RHEL from Red Hat. There is a rpm package that contains a script named rtcheck. Ubuntu does no contain this script and the packaging system does not contain it, too. Some guidance for the use of JVM implementations described as a solution to create a program that returns 0, which is the correct indication of the test. I downloaded a rpm package that contained rtcheck. Then I compiled it and tried to ensure that all conditions of the test are met. Example of rtcheck output follows:

```
RTCheck 0.7-4 - Linux Real-Time Environment Checker
--------------------------------------------------
```

```
RTCheck Initialization: > Locking all memory: ok
Setting up real-time scheduling: ok

System Tests:
Looking up boot_id (20e0572a-8007-4639-829d-abb3882b7ff7): ok
Checking for out-of-tree RT extensions: failed
Kernel was not built with CONFIG_PREEMPT_RT=y
Checking for robust (PI) mutex support: ok
Testing for acceptable hrtimer resolution (<=100us): ok
Reporting 60us
Testing for acceptable clock resolution (<=200us): ok
Reporting 1ns
Caching results in /var/cache/rtcheck: ok

User Permission Tests:
Trying to lock memory: ok
...
```

It was necessary to adjust the limits for allocating memory. This can be done by changing the value in the configuration file limits.conf. Must be set: * - *memlock unlimited*. Current settings can be tested using *ulimit-l.*

Another test for proving real-time capabilities of used system is cyclictest. Cyclictest is designed specifically to locate and identify latencies in a real-time system [17] The description of use is located on the web [16]. Results of cyclictest for 10000 iterations, 10000 us interval, priority of threads 80, smp – symetric multi-processing uses *cloc_nanosleep* and number of threads is same as *max_cpus*:

```
janousekm@Ample-Michal:~/cyclictest/rt-tests$ sudo ./cyclictest --smp  -p 80 -i 10000 -l 10000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.13 0.16 0.09 1/358 7165

T: 0 ( 7164) P:80 I:10000 C:  10000 Min:      6 Act:   15 Avg:   16 Max:      30
T: 1 ( 7165) P:80 I:10500 C:   9523 Min:      7 Act:   16 Avg:   15 Max:     120
janousekm@Ample-Michal:~/cyclictest/rt-tests$ sudo ./cyclictest --smp  -p 80 -i 10000 -l 10000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.30 0.21 0.11 1/355 7178

T: 0 ( 7168) P:80 I:10000 C:  10000 Min:      6 Act:   15 Avg:   15 Max:     549
T: 1 ( 7169) P:80 I:10500 C:   9524 Min:      6 Act:   15 Avg:   15 Max:     287
janousekm@Ample-Michal:~/cyclictest/rt-tests$ sudo ./cyclictest --smp  -p 80 -i 10000 -l 10000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.05 0.14 0.09 1/354 7181

T: 0 ( 7180) P:80 I:10000 C:  10000 Min:      6 Act:   15 Avg:   15 Max:     283
T: 1 ( 7181) P:80 I:10500 C:   9524 Min:      6 Act:   15 Avg:   15 Max:      29
janousekm@Ample-Michal:~/cyclictest/rt-tests$ sudo ./cyclictest --smp  -p 80 -i 10000 -l 10000
```

```
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.04 0.12 0.09 1/354 7186
@
T: 0 ( 7185) P:80 I:10000 C:  10000 Min:     6 Act:   15 Avg:   15 Max:      28
T: 1 ( 7186) P:80 I:10500 C:   9524 Min:     7 Act:   14 Avg:   15 Max:      27
```

The most important part of the results is in the last column. The maximal latency achieved during the test. The values are in microseconds. These values are sufficient, because the latency of the tasks in th benchmark is in units of milliseconds.

The used kernel name from the use of command *uname -a* : 2.6.31-11-rt #154-Ubuntu SMP PREEMPT RT Wed Jun 9 12:28:53 UTC 2010 i686 GNU/Linux. The notebook which was used for development and benchmarking is lenovo Thinkpad R500. The hardware configuration is as follows

- 2 x Intel Core 2 Duo CPU (P8700 @ 2.53 GHz)
- 4 GiB RAM - 2.9 GiB avaliable because of 32-bit architecture of system

4.2. Description and Subprojects

A good practice to develop complex applications is a division of the project into smaller subprojects, so the application was split into multiple parts. At the beginning there were only algorithm source code written in C programming language and files containing data needed for simulation. I had to study the functioning of the MCL localization and source code written in C programming language. Then I started working on the first version, which ran only in Java SE. It made no sense to engage in the development of real-time features did not work until the localization algorithm.

Plain Java version only served to test the correct functionality of the MCL rewritten in Java programming language. Even back then but had to be solved a problem retrieving the source data for the simulation and generate output data for comparison. Another version of the project have used the RTSJ. The problem is that if a newer version of a program developed so that it overwrites the old version directly in the same project, it is possible to quickly obtain results to compare the two versions. History of development is indeed stored in the versioning system and can be undone at any time, but development is very impractical. In addition, using different versions of classes contained in the Java development kit (JDK) can cause conflicts. It would be impractical to maintain real-time classes in the same project as classes written in plain Java.

The main logic of MCL along loading input and generating output data were placed in a separate project. Other projects include the implementation of event processing and individual tests. All versions are not used to generate results that are included in the final comparison. Gradual expansion of the project can be easily seen in the history of the git repository.

4.2.1. Algorithm Rewriting

I had some experience in using the C programming language, but mainly from school projects related to hardware, where it was necessary to use it. With the help of my advisor I ran C version for testing and I could start programming the first Java version. Rewriting is generally very boring but it is easy to make mistakes that are hard to detect.

First I had to design a basic directory structure of the project and the distribution of packages. The structure does not differ from the standard Java SE project but contains other directories with the necessary parts of the benchmark. Source package I called as in custom Java. First goes the country, then the company or developer's name and the name of project. This results into: **cz.janousekm.jmclbench**. This package contains two sub packages. One of them the **Io** contains classes for support of input and output data. Second package called **mcl** contains classes representing the MCL logic.

The C files and the other files related to simulation are placed in the repository folder named **c**. This part is managed by the advisor of my work. Logic of MCL algorithm written in C programming language was divided into multiple files. It was not easy for me to navigate through these files, but I understand how they are interconnected, and I chose names to represent different parts in Java. The main class is the **cz.janousekm.jmclbench.mcl.RobotIface** which describes the main methods of the MCL. The robot consists of the estimated current position, the necessary constants required for calculations, fields containing individual particles, their weight and other variables.

Another important extracted parts were the representation of input data types. It was necessary to model the laser measurement, the date from odometry and the position of robot. In the data types packages are contained following object classes: **Measurement**, **RobotPosition** and **LaserState.** Coordinates and angles are placed in double and time is represented as a long representing the millisecond. For the benchmarking purposes is there integer value called index which is used for counting deadline misses. They are simple objects which encapsulates primitive data types. Measurement is used for representation of incoming data from the laser sensor. RobotPosition is used for the representation of robot's position. The RobotPosition is used for the additions received by odometry sensor, too. LaserState is used for the representation of the particles.

Important constants for the description of the playground were grouped into the one class. The name of the class is **PlayGround** and contains the height and width of the playground and the position of reflection beacons. Class contains a public inner class to represent the position of the beacon with two double values, which correspond to coordinates on the playground.

Writing the Java code was not so time-consuming but it was worse with tuning and debugging. Java version did not work correctly on the first attempt so it had to contain errors. Find an error in nondeterministic algorithm is not easy especially because of the use of random number generator. Together with my advisor, we divided the algorithm into smaller parts, which we tested separately to verify that it returns the correct results. A mistake in the wrong evaluation of the likelihood of the individual parts, was removed.

4.2.2. Projects Structure

The development was divided into multiple projects. To understand the proper functioning of the benchmark is important to understand what different sub-projects contain. Brief description of the Java projects contained in the distribution:

- GraphsGenerator - independent project for generating gnuplot graphs from output data
- jmclbench - main logic of MCL and shared classes
- jmclbench-pj - projects containing plain Java versions
- jmclbench-rtsj - projects containing RTSJ versions

The content and purpose of individual project will be described below. Some projects have been already mentioned and does not make sense to discuss them again. All files can be found in the electronic annex to this thesis.

4.2.3. Repository Structure

The repository contains not only the Java project directories. Are there also other directories containing parts needed for proper use benchmark. The whole structure is as follows:

- c C programming language codes and simulation data
- GraphsGenerator Java projects for generating graphs
- jmclbench Project contains the main logic.
- jmclbench-pj-v1 Version one of benchmark logic in plain Java.
- jmclbench-pj-v2 Version two of benchmark in plain Java.
- jmclbench-rtsj-v1 Use of aperiodic and periodic RTT thread.
- jmclbench-rtsj-v2 Use of Test thread as periodic Task a event handler for laser event.
- jmclbench-rtsj-v3 Testing interruptible capabilities of asynchronous thread termination and control transmission.
- jmclbench-rtsj-v4 Using handlers for task processing.
- jmclbench-rtsj-v5 Test of using another memory models.
- jmclbench-rtsj-final Java project containing final version used for generating presented results.

- www - files used for web presentation

Important are only directories jmbclbench, jmclbench-rtsj-final, GraphsGenerator, and www. Other directories include the development versions, which may not be functional.

4.2.4. Plain Java Versions

The first version transcribed from the C run only in one thread and not address the problems with the coming of events at the same time or coming events even when the previous process. It represented an ideal state which is actually an infinitely fast computing unit and has unlimited time to do needed computations for each event. If such a machine existed, it would always met all deadlines and using real-time programming and real-time implementations of the JVM would be unnecessary.

Retrieving data was provided by **DataReader** class from package cz.janousekm.jmclbench.io. The class constructor receives three String parameters which are the paths of files containing the data required for the simulation. These are laser data odometry data and data of the real position of the robot on the field. Data is loaded into memory and the access of them is possible via getters of the object. This is the final version of the class, but I used implementation which reads data from file by line but it could cause some I/O latency. So data is loaded directly to the memory.

The next step was to write multi-threaded version of the algorithm. For each type of processing task was used a single thread. The release of threads to start process an event was driven by using semaphores. Java.util.concurrent package contains Semaphore class which is used. Each thread has his own semaphore and waits for the release using the semaphore's method acquire(). In following example is the use of semaphore :

```
/* Example of Semaphore usage */
while (run) {

try{
 semaphore.acquire();
 synchronized(robot){
   robot.laserUpdate(meas);
   robot.laserResample();
}

semaphore.release();
```

Threads have inside infinite while loops, which depends on boolean value. When threads have to end the boolean value is set to false and when they are released using semaphores the loop ends and as well as whole thread. Loops are interrupted from the control thread by setting the run boolean of the thread. Control threads are in application named as test and are represented by the Test classes. The tests are objects which handle the control of tasks and measure some of the metrics. Test implements different approaches of event processing which is described in third chapter. The names of the test classes are **SkipTest** and **TerminationTest**. The name of termination class was shorten to **TermTest** in RTSJ implementations.

Tasks in plain versions are represented directly by Thread classes. The names of classes are UpdateThread and PredictThread. Names match the encapsulated parts of MCL. Part of the task is to record the state of the robot after performing the MCL update or predict depending on the task. It has to be done because it is needed for comparison. This data could be written into files or stored in memory.

In the plain versions was the processing of the results part of the tests, which means that the task were a little bit more demanding on computation time. Which was subject to change in RTSJ versions.

The two plain Java projects were used for better and faster comparison of tests results. They are now same it differs only in used implementation of Robot class, which was tunned to avoid instantiation in the MCL methods for use in Immortal memory. The work with plain Java version is generously faster, so i used it for development of new features and tuning of the main logic residing in jmclbench project which contains only plain Java classes.

4.2.5. RTSJ Versions

RTSJ offers more options for solving the time-dependent processing of individual tasks. It is possible to use the RTT or AEH class for implementation of tasks. Basic usage of these classes is previously described in the chapter 2.

The main project contains more subprojects of versions written for RTSJ as well as Java SE. Particular versions are used for testing different functions of real-time Java because RTSJ offers more options for implementing same things. I have tested the use of periodic and aperiodic RTT and the use of AEH and BAEH classes. I tried to use non-standard memory models but I do not have any success with it.

The final version is using for the event processing BAEH class. Usage of BAEH class is same as the usage of AEH class because the BAEH extends AEH. Implementation of BAEH class differs from AEH in events handling. BAEH creates one dedicated thread for handling events bounded to this handler. The overhead is in bounding one thread for use for only one task, but the advantage is smaller lateness of dispatching. Thank to use of EventHandler are the events handled sequentially in the right order according to FIFO(first-in-first-out) principle. So BAEH allows faster dispatching of incoming events and because of that is appropriate for use in this case. In addition event handler are usable in another implementations of real-time Java as Safety-Critical Java which I would test in the future.

Subproject contains 4 implementation of tasks. They are located in **cz.janousekm.jmclbench.rtsj.handlers.tasks** package. Reason for so much implementations is the problem with task termination. Because of the handlers do all the work with task management and encapsulate the event processing it is very difficult to interrupt threads bounded to them. RTSJ provides mechanism for Asynchronous Thread Termination (ATT) and Asynchronous Transfer of Control (ATC).

ATT is achieved by the use of the **javax.realtime.Interruptible** interface which is connected with AsynchronouslyInterruptedException(AIE). Interruptible has run()

method which throws AIE. Within this method can be executed code which can be asynchronously interrupted by calling the AIE fire() method. The object which implements Interruptible interface have to be associated the instance of AIE. The instance of AIE can be shared within more threads. When the fire() method is called and there is a RTT executing his interruptible run method associated with this AIE, the execution of run method is interrupted. Interruptible interface provide interruptAction method which serves for handling the interruptions. Tasks used in TermTest implement Interruptible interface and uses interruptible run method. If thread is interrupted it is equal to termination of task.

Tasks are represented as classes which implements the Runnable interface. They contain only references to shared objects and variables and the run method. Run method consists of calling robot's method and writing measured data to the memory. Sharing the Robot object between handler could case the conflicts in access to Robot's data. Only one thread can work with in a given time. Integrity of the Robot object's data is secured by use synchronized keyword. The calling of Robot objects method is encapsulated as critical section using the synchronized block. The robot methods are called atomically.

The problem is to find out if a task is running because the thread is encapsulated by event handler. If a new event arrives it has to be decided what to do. It can happen that another task is still processing then the behaviour depends on the type of test. But both tests need to know if a task is processed. The solution of this problem is the use of a class from java.util.concurrent.locks package, concretely **ReentrantLock**. This class has methods tryLock() and isLocked(). Before starting of execution the task is tried state of the lock. If is the lock locked it indicates that task is running. Thanks to isLock() is found out that a task is processed and tryLock() method founds out the state of lock and when it is unlocked the method locks it.

The control thread handles the dispatching of event a it is implemented as RTT. Both types of test extends the Test class. **Test** class contains the declaration and initialization of variables, which are same for both types of test. And it has method for processing the results after simulation. The priority of Test has to be higher than the Tasks priority value.

The tasks are executed within the TaskHandler class which extends BAEH. Constructor of TaskHandler accepts as parameter an integer value which is subtracted from the default value of priority. The default value of priority is obtained from Scheduler as maxpriority. Next important parameter is instance of Runnable. Used instances are Tasks.

Simulation of events is implemented as iteration through the events data. The times of arrival of event are compared to each other. There two arrays of events corresponding to the laser and odometry. They are sorted according to time. The earlier event is dispatched first. The logic of Test finds out the time of next event and then compares it the next event of the second type of event. Then the tread is suspended by using sleep() method. Sleep method in the RTSJ implementation accepts instance of HighResolutionTime class, so the thread can be suspended to precise absolute time. When the thread is awaken then fires the prepared event and then prepares new event and goes to sleep to the time of next event. Data for concrete are in classes shared between control thread and the handlers.

The final version of the benchmark is located in project named jmclbench-rtsj-final

4.2.6. Building Benchmark

For compiling the Java programs are mainly used apache ant or maven. Personally, I have experience with both programs. Using maven is a bit complicated but the advantage is the extensive management of dependencies. Management of large projects with many dependencies would be very difficult without maven. Maven has in addition to the program building and management more features such as support for creating documentation and reporting. Ant is very simple because it is used mainly to compile and to assemble programs.

The benchmark has not many dependencies but consists of several projects, so it is more suitable for Ant use. Using Ant consists of the targets and tasks that are defined in the XML file. The file resides in jmclbench project directory and its name is build.xml. Ant can be simple executed from console:

```
janousekm@Ample-Michal:~/jmclbench/jmclbench$ ant

janousekm@Ample-Michal:~/jmclbench/jmclbench$ ant rtvFinal -Djavac.path=<path to java>
```

Default value of java.path is set to /usr/lib/jvm/java-6-sun/bin/javac. Standard target builds version in plain Java. The result of building is a jar file containing the classes in byte code. Jar file is built and packed into ./build/jar and default name is jMCLBench.jar.

Assembling and building the benchmark will link jmclbench and another project with the name which specifies the implementation of test behavior. In dist folder are copied the source code files from the packages of the linked projects according to selected version. The name of the packages are same for all jmclbench projects. They are also copied the libraries from the lib directory. It is used only one external library containing the command-line parser. In Eclipse is to each project linked the jmclbench project containing the logic. So plain and real-time versions contain logic from the perspective of the IDE.

Such appropriately building allows using different compilers for compilation. For testing different JVMs is important know the paths. When you want to use real-time functions of real-time JVMs you have to build soft or hard real-time version. You have to use RT Linux kernel for the real-time version. Non real-time version can be ran on all the types of JVMs.

A more detailed description of how to build is located in a text file ReadMe.txt in the jmclbench project directory.

4.3. Benchmark output

In the benchmark is the generation of results separated from the simulation. Benchmark can write temporary results into files but it can cause possible I/O delay and had bad impact on the results. So the temporary results are only saved into memory, then they are processed. The graph generation was included in jmlclbench subproject.

Generated results are saved into files. The generation of data is provided by using countOutputData() method in Test class. The temporary measured data are used for calculating the metrics into file named data.txt. For calculation is used the Math class which implements needed mathematical methods. The file contains these data:

- deviationXY - combined average deviation of position(coordinates x and y) in metres
- deviationX - average deviation of coordinate x in metres
- deviatonY - average deviation of coordinate y in metres
- deviatonAngle - average deviation of angle in radians
- deadline-missCount - the amount of occurred misses
- averageReleaseJitter - average value of the difference between required and real start time
- averageTaskTime - average time for processing of one task

and This is provided by The location of files depends on the type of test. Generated into files according to

The information about time are saved into object of **DataLine** class. This class contains integer value representing the index of event and two longs which represents the time. The position of robot is written to array in memory and then processed.

4.3.1. Folder Structure

Results are generated by following rules:

```
,/resulst/jvm/covariance/particlescount/typeoftest/numberoftest/

jvm : name of tested JVM
covariance: indication of counting the covariance
particlescount: the count of used particles
typeoftest: used test (TermTest, SkipTest)
numberoftest: the number of same test in a row
```

The files with measured data are located in "numberoftest" folder. The names of the files are laserData.txt, odoData.txt, position.txt and positionDeviation.txt. Files containing the times of event processing are laserData.txt and odoData.txt. In position.txt the data about position are stored and in positionDeviation are stored values of xy deviation. If covariance benchmark counts covariance data they are stored in covariance.txt.

4.3.2. Graphs

Generation of graphs is separated into its own subproject named GraphsGenerator. The main generation of the graphs is in the GraphsGenerator class. This utility searches the folders structure using depth first search (DFS) and generate graphs. The data files are mostly in the roots of directory tree. Position of files in the folders determines the descriptions of graphs.

The utility can be built using ant as well as benchmark. The usage is very simple. The only parameter is absolute path to the folder which contains results.

For each graph is generated text file with extension "gp". This file contains gnuplot commands for generating of corresponding graph. The graph is generated by calling gnuplot from Java with generated file as parameter. Some of the graphs especially histograms need another processing of the results before plotting. This is provided by classes which work similarly as the GraphsGenerator in the manner of DFS and aggregates the data from results. Then create a gnuplot files on higher levels in the directory tree. Types of generated graphs are

- XY lines graph
- XY points graph
- histogram
- histogram with error values

I used GraphsGenerator from IDE, but it can be build using ant. If the GraphsGenerator is executed from command line the path to the folder with results have to be specified. Graphs are generated to the location of their source files.

Examples of graph generated using GraphsGenerator

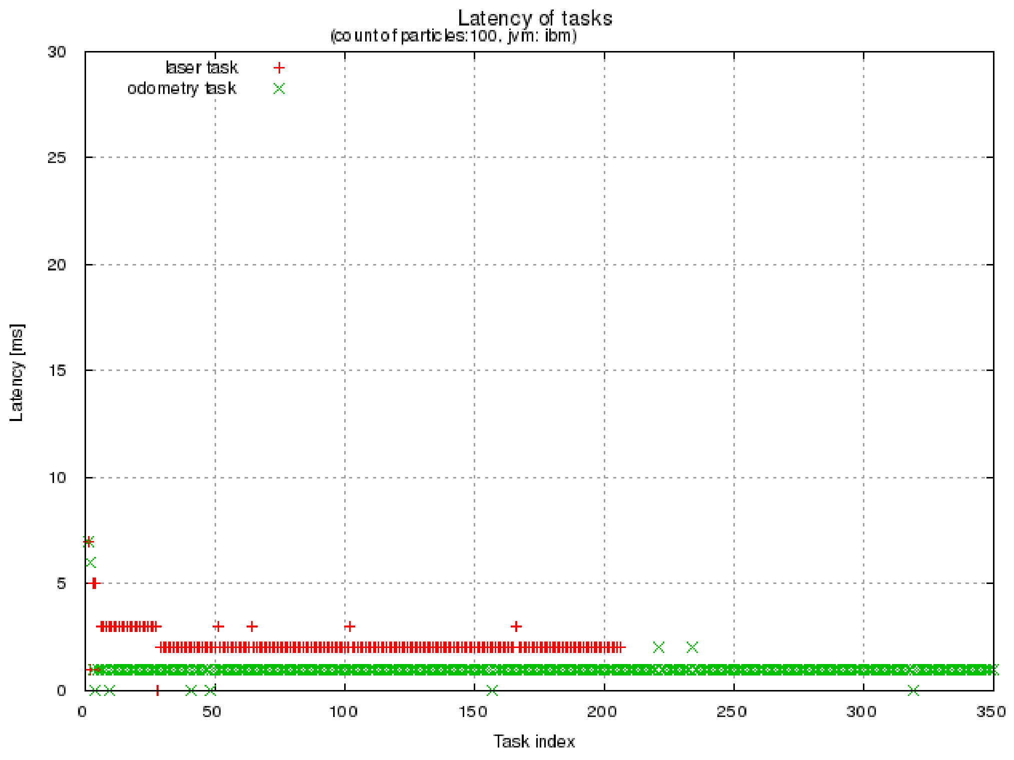


Figure 4.4. Example of XY points graph

Graph represents the distribution of task latency according to index. The index corresponds to the time-steps of simulation. For odometry task is the time-step 50 ms. For laser task is time-step variable. From this graph can

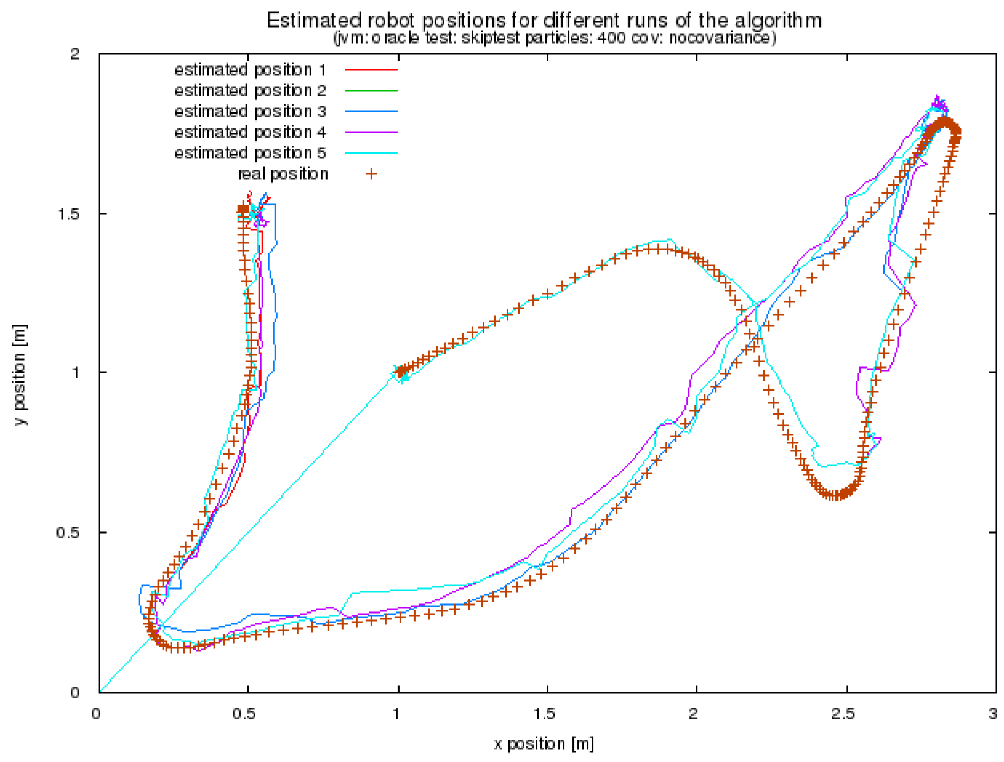


Figure 4.5. Example of XY lines graph

Graph for comparison of 5 five runs of SkipTest. The count of particles is 400 and tested JVM is oracle. This graph shows the deviation of estimated position from real position. Tests were run several times for the given parameters. It is used for verification of correct functioning of the algorithm and for comparison of differences between runs. Because it is important to repeat same results in benchmark application. The used MCL algorithm is nondeterministic. The pseudo random generator is initialized by same value for generation same sequences of random numbers which itself does not guarantee deterministic behavior. But the repeatability is better.

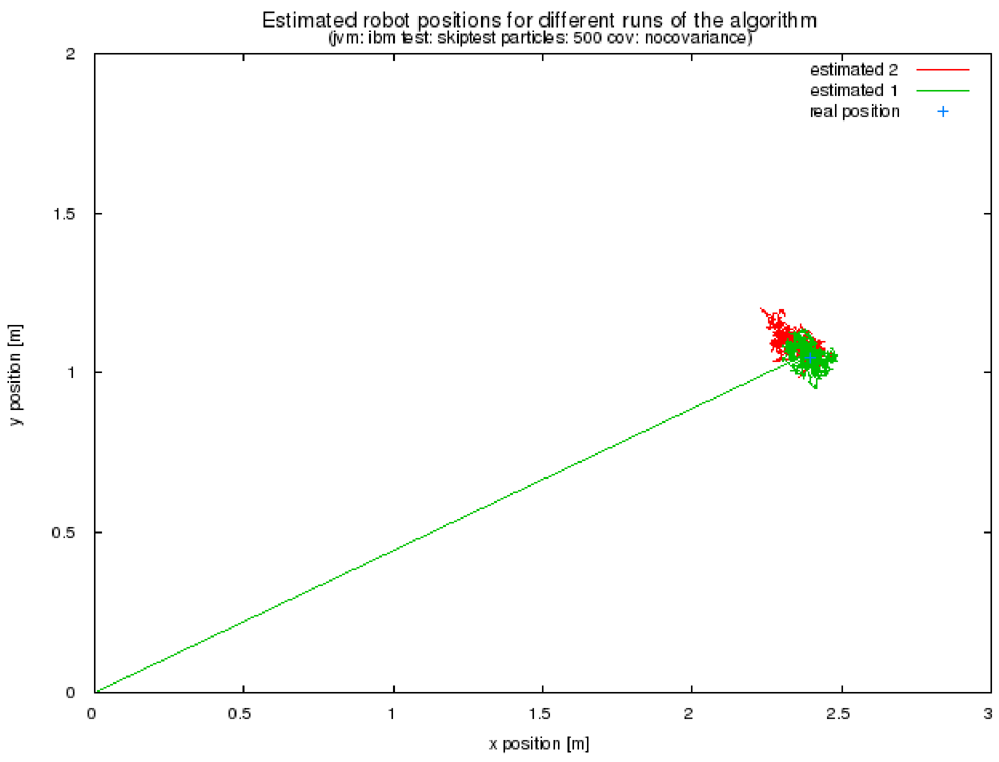


Figure 4.6. Example of comparison graph

This graph shows two runs of SkipTest with different trajectory. The robot did have to stay in the center of beacons. The cross represents the position in the middle of beacons. The algorithm has not implemented staying so it moves around the real position.

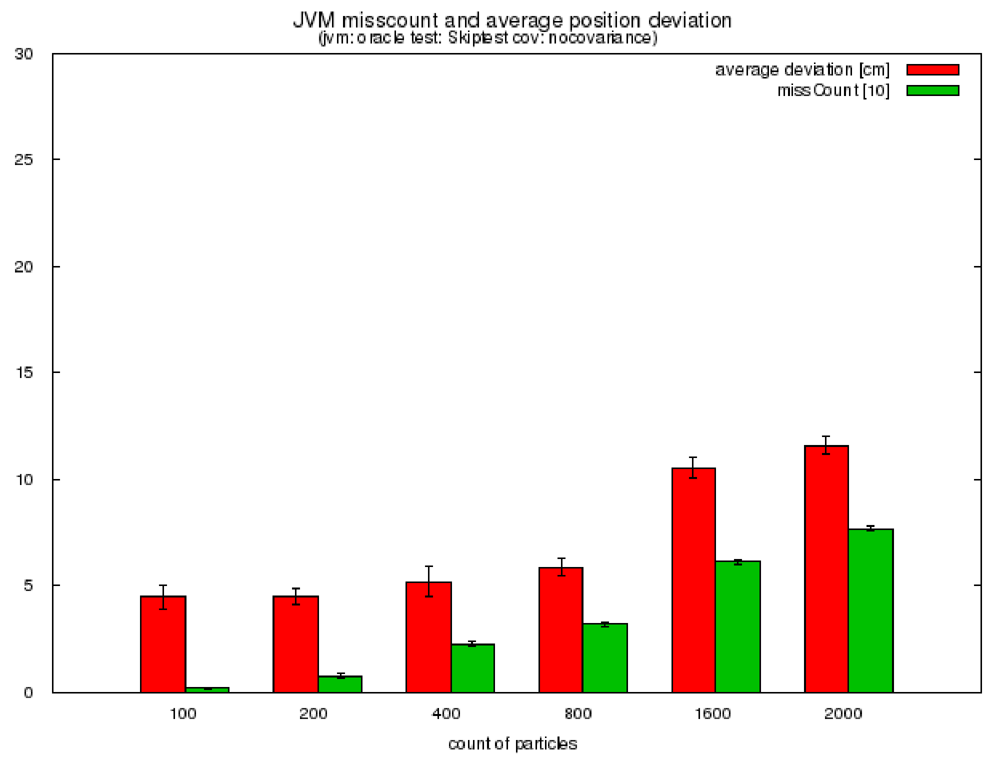


Figure 4.7. Example of histogram with errors

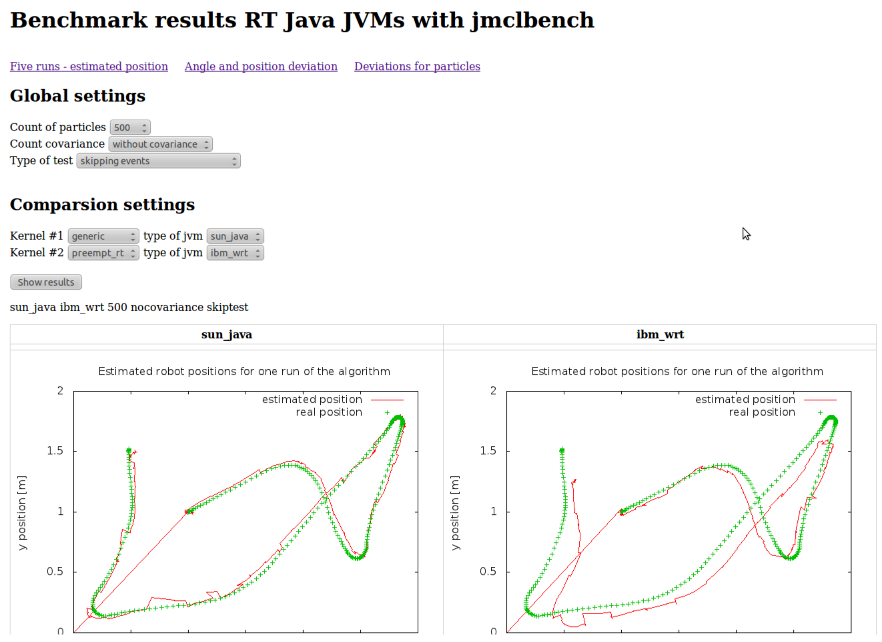
Example of classical histogram used to display the dependence of missCount and average position deviation on count of particles. For better comparison is missCount value in tens.

4.4. Presentation and Testing

For better comparison of results and simpler testing was necessary to develop some support scripts and documentation for users.

4.4.1. Web layer

The presentation layer is simple web page written in PHP. The website is used for direct comparison of some generated graphs according to parameters used for simulation. The graph images and data files are read from the results folder of jmclbench subroject. But only the results are copied to a web server.



Benchmark results RT Java JVMs with jmclbench

Five runs - estimated position   Angle and position deviation   Deviations for particles

Global settings

Count of particles 500
Count covariance without covariance
Type of test skipping events

Comparsion settings

Kernel #1 generic type of jvm sun_java
Kernel #2 preempt_rt type of jvm ibm_wrt

Show results

sun_java ibm_wrt 500 nocovariance skiptest

sun_java	ibm_wrt

Estimated robot positions for one run of the algorithm
estimated position
real position
y position [m]
2
1.5
1
0.5
0
Estimated robot positions for one run of the algorithm
estimated position
real position
y position [m]
2
1.5
1
0.5
0

Figure 4.8. Screenshot of web application for comparison

4.4.2. Testing

Successful development of application means a lot of time spent for testing a debugging. In earlier versions are more simulations ran through one initialization of JVM, but in the final version is for each test JVM initialized again. The purpose of this was better management of the tests.

The basic usage of provided console application is described below;

```
usage: jmclbench [-c] [-h] [-j <arg>] [-laser <arg>] [-n <arg>] [-O <arg>]
      [-odo <arg>] [-p <arg>] [-real <arg>] [-T]
 -c             count covariance
 -h             print this message
 -j <arg>       name of tested jvm
 -laser <arg>   file containing laser data
 -n <arg>       number of test
 -O <arg>       dir where to generate data
 -odo <arg>     file containing odometry data
 -p <arg>       count of particles
 -real <arg>    file containing real position data
 -T             test for termination of unfinished jobs
```

For more comfortable testing are used shell scripts for calling the tests. They are located in ./script/test folder of jmclbnench subproject. The names of the script contains names of tested versions. Example of shell script used for testing :

```
ant

testnonrt() {
for i in 1 2 3 4 5
 do
   java -verbose:gc -jar build/jar/jMCLBench.jar -j sun-java -p $1 -n $i
 done
}

for n in 500 1000 2000 3000 4000
do
  testnonrt $n
done

exit 0
```

Chapter 5

JVM Evaluation

In this chapter are presented and discussed the results of benchmark. Few graphs of different tests are discussed for each tested JVM implementation. All the graphs can not be included in this thesis and they are located on the appended CD with other results. In particular, I chose graphs that display data from multiple measurements and compare them as metrics. The benchmark was ran on RTOS which was not fully loaded. The simulation ran without doing anything else on the OS. I used basic configuration of all JVM implementations. I only used switch for printing the info about GC on the standard output. The switch is "-verbose:gc".

5.1. aicas - JamaicaVM (Jamaica)

Jamaica is hard real-time JVM implementation provided by aicas company. It can be only ran on RTOS.

5.1.1. Deviation and MissCount

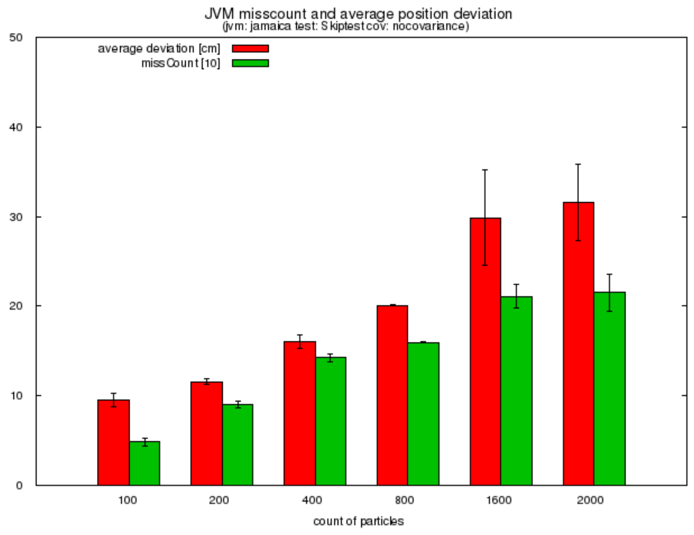


Figure 5.1. Histogram for skipping of tasks test

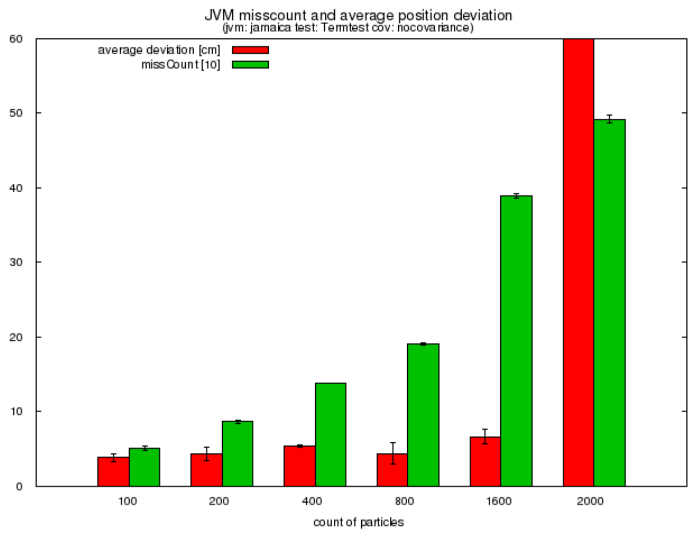


Figure 5.2. Histogram for termination of tasks test

This two graphs provide comparison of JamaicaVM performance in both tests. The first graph shows the results for SkipTest and the second graph shows results for TermTest. Graph shows the dependence of missCount and combined standard deviation for coordinates on count of particles. From the graphs it is evident that the JamaicaVM achieves better results in the test of termination results. In the both types of the test were the results for 1600 and 2000 particles very bad and the algorithm has stopped working. In the second graph the difference between the results for 100 and 200 particles was how was expected. That the missCount will increase with increasing count of particles and deviation will decrease to some point where the count of particles will be too big. The deviation for 400 particles is worse than for 800 particles. The precission of the algorithm should be better for larger numbers of particles. The missCount increase proportionately to the count of particles. In the test of skipping events the deviation and missCount increase proportionately to the count of particles. In both of graphs is the missCount very high. The error values are not too hight,

5.1.2. Task Latency and Release Jitter

This graph shows the average processing latency of task and average jitter. These values are aggregated from more simulations. On the y axis is the time in milliseconds and on the x axis are counts of particles.

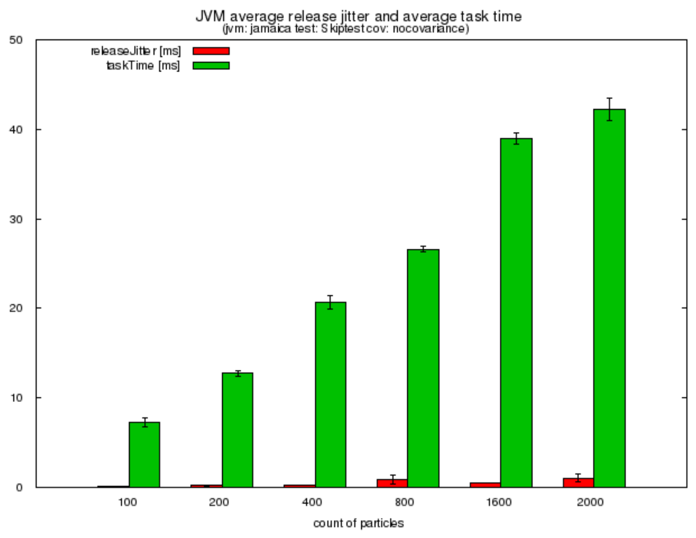


Figure 5.3. Histogram of average task latency and release jitter for SkipTest

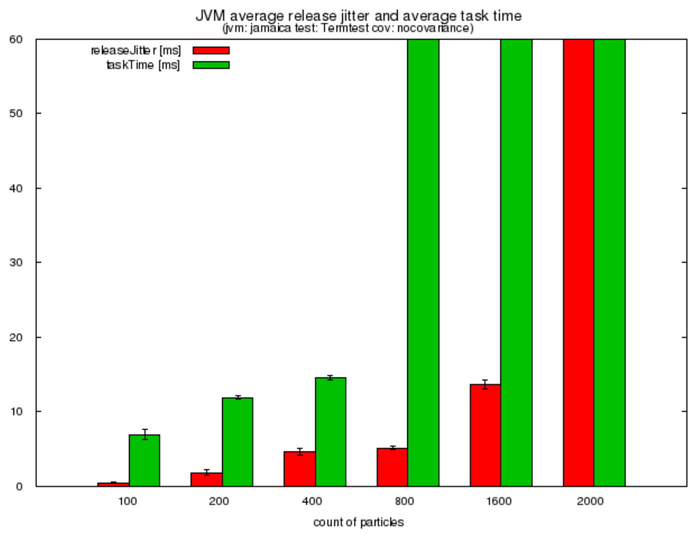


Figure 5.4. Histogram of average task latency and release jitter for TermTest

These three graphs show the average length of task. We can see when was the latency higher. The latency of both tasks increase proportionally to the count of particles. For TermTest were the average latency lower because of the termination of processing. In skipping test tasks ran without interruption. The release jitter of tasks is increasing like the task time but slower.

5.2. IBM WebSphere Real Time for RT Linux (IBM)

Second tested hard real-time JVM implementation is from the IBM. It uses own GC Metronome. The special configuration for this JVM is the mandatory use of this GC.v The switch is "-Xgcpolicy:metronome".

5.2.1. Deviation and MissCount

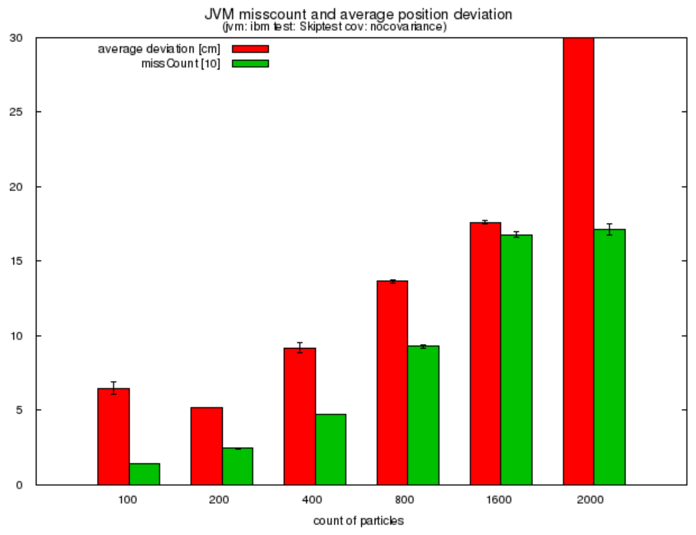


Figure 5.5. Histogram for skipping of tasks test

There are only results of skipping tasks because the real-time JVM cannot handle the termination of tasks properly. Probably it si casue by the use of AsynchronouslyInterruptedException, which could cause unexpected behavior of algorithm. The usage of termination cause a deadlock. This deadlock breaks down the whole system because of the high values of thread priority, The results of skipping task test are better than for JamaicaVM, but the miss count increases slowly and the deviation was better for all of the counts of particles in skipping test in comparison with JamaicaVM. For 200 particles is the deviation lower than for 100 particles. The missCount is constantly increasing.

5.2.2. Task Latency and Release Jitter

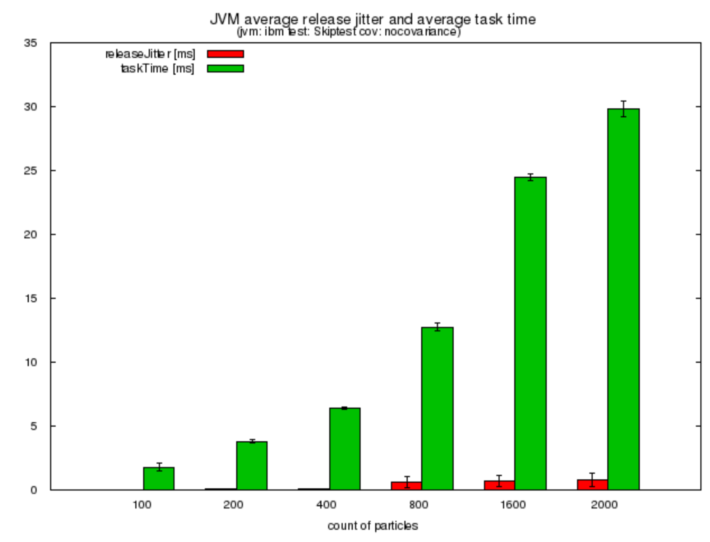


Figure 5.6. Histogram of average task latency and release jitter for SkipTest

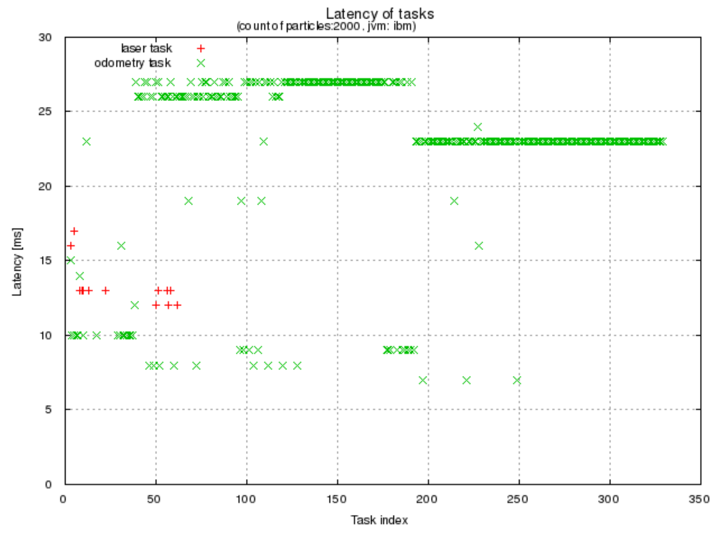


Figure 5.7. Graph of task processing distribution of 2000 particles

Second graph shows the processing length of task. Task are divided according to type because the complexity of computation is for each task different. On the y axis is the latency in milliseconds and on the x axis is index of processed task. On this graph is showed the maximal task latency which is 30 milliseconds. This is very long time for processing of one task. It is caused by the type of test, where the tasks are not interrupted and run longer.

The results of task latency and release jitter are better than for JamaicaVM. For the high count of particles are the times still better than for Jamaica. It means that the Websphere real-time implementation is better in the terms of meeting deadlines and the task are processed faster. The times for 2000 particles are higher for only few milliseconds. The algorithm is still working for 2000 particles, but the precision is very low.

5.3. Sun Java Real-Time System 2.2 (Oracle)

Sun's version of JVM implementation has no requirements on the underlying system. So it was possible to do the tests on the system without real-time kernel. I did the tests on real-time kernel too. This section describes the benchmark for runs on both types of kernels. The tests with larger numbers of particle launch GC. But the GC runs only in normal, which means that the tasks can not be preempted by GC.

5.3.1. Tests on Generic Kernel

The benchmarks tests were performed in the same manner as on the system with real-time kernel.

5.3.1.1. Deviation and MissCount

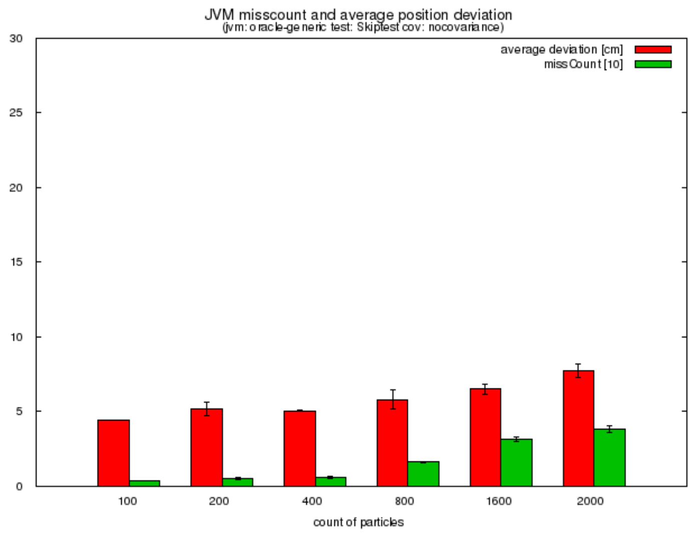


Figure 5.8. Histogram for skipping of tasks test

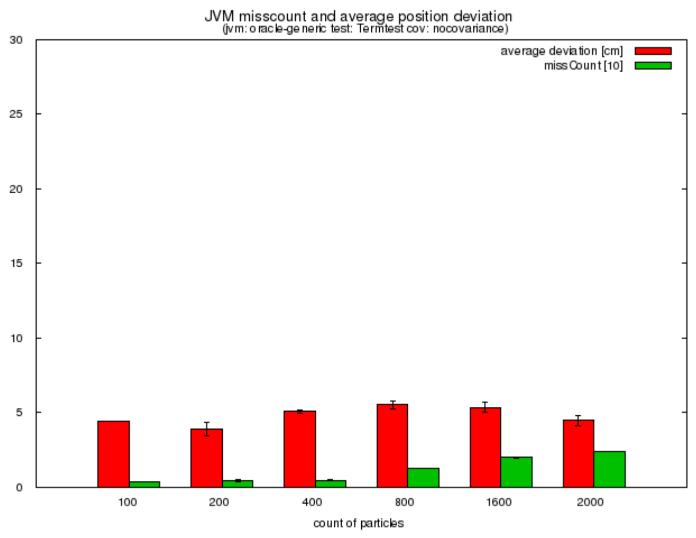


Figure 5.9. Histogram for termination of tasks test

The deviation of position increases very slowly and the results are better for the termination test. The comparison of the estimated trajectories of robot is almost the same. The graphs of five runs often looks, that there is only one curve. This is caused by running the benchmark without doing other things on the system.

5.3.1.2. Task Latency

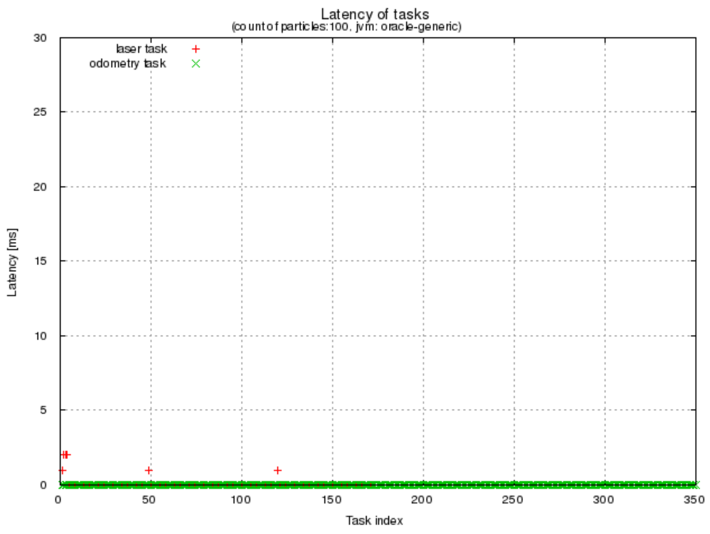


Figure 5.10. Graph of task processing distribution of 100 particles

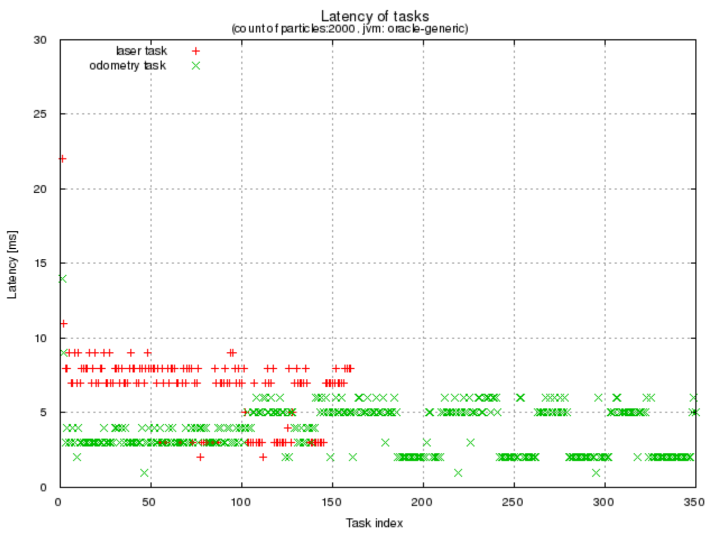


Figure 5.11. Graph of task processing distribution of 2000 particles

The latency of tasks is for 100 particles under 1 millisecond in the hundreds of microseconds. And the latency for 2000 particles is much better than at previously tested JVMs. I do not have the histograms of average task latency and average release jitter. I have done it only for the tests on real-time kernel.

5.3.2. Tests on RT Kernel

Tests on the real-time kernel are not much different from the tests on generic kernel. In general they are a little bit slower. The latencies are higher but not much so it is unnecessarily to show the results of latencies on real-time kernel.

5.3.2.1. Deviation and MissCount

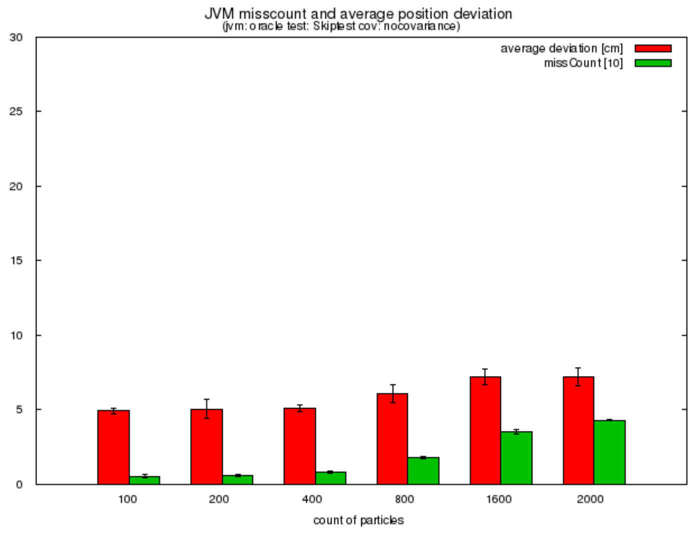


Figure 5.12. Histogram for skipping of tasks test

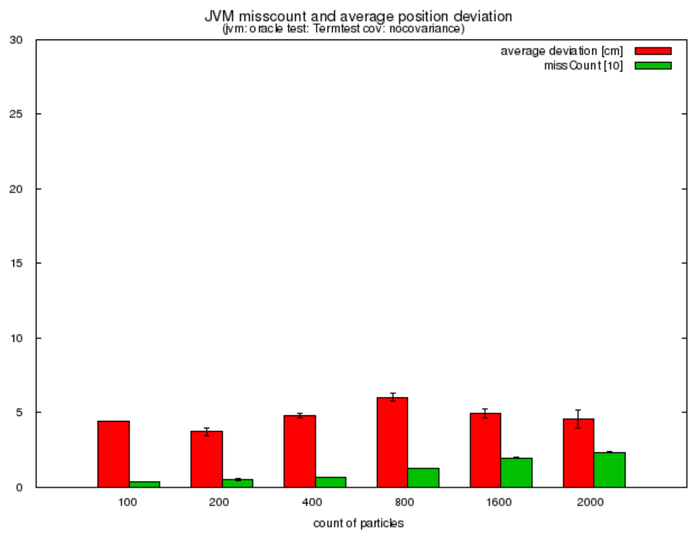


Figure 5.13. Histogram for termination of tasks test

The results of the tests are not very different. The values of average deviation of position and average missCount are lower. Termination of task has slightly better results. The deviation is lower for 2000 particles than for 800. The results are same for more different runs of benchmark. The missCount for 2000 particles is comparable with the missCount for the low values of particle count. The error value of showed metrics are low, which means, that the results for different runs do not differ much.

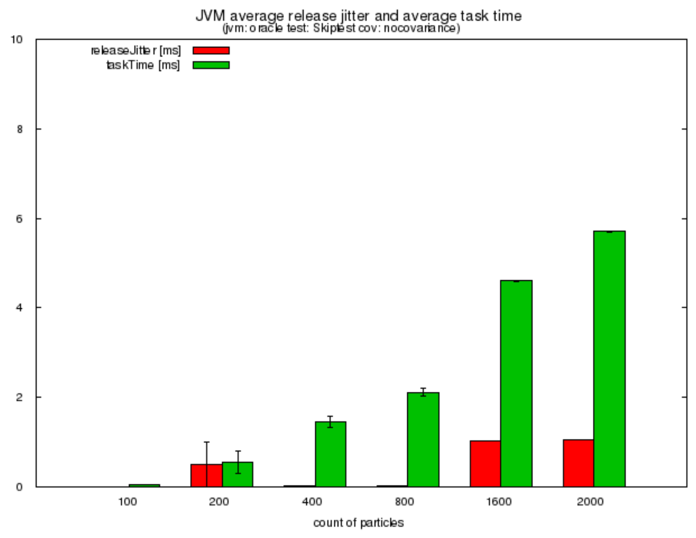


Figure 5.14. Histogram for skipping of tasks test

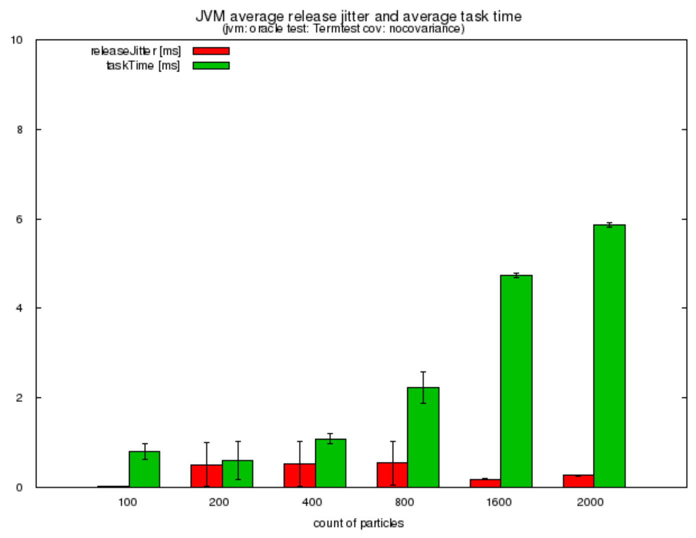


Figure 5.15. Histogram for termination of tasks test

The values of latency times and release jitter times are very low for both types of tests. The values of tasks latencies are not larger than 8 milliseconds. Form TermTest is the release jitter is in hundreds of microseconds.

5.4. Comparison of JVM

The most important result is the graph which summarizes all the JVM implementations in one graph. The average deviation is important parameter for testing the algorithm but most important metrics is the missCount.

5.4.1. Pure Hard Real-time JVM

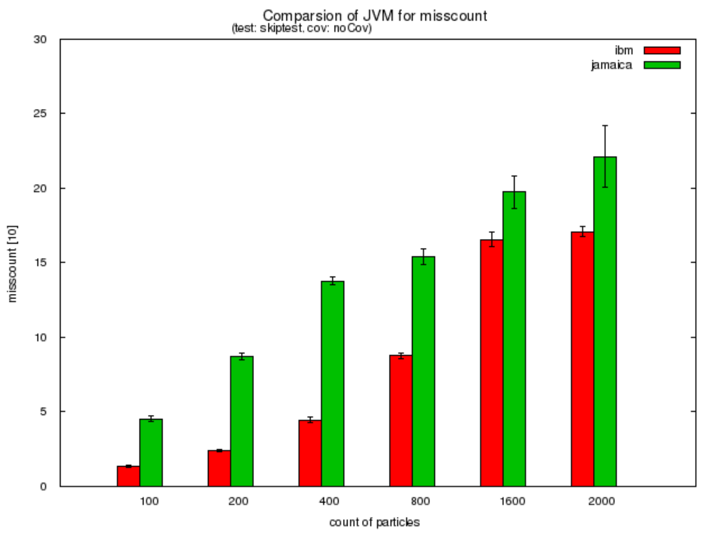


Figure 5.16. Histogram of average missCount

In the comparison of missCount are much better the results of IBM implementation. In proper hard real-time application should not be too much deadlines misses, which can cause failure of whole system. Both implementations have very high values of missCount for small number of particles.

5.4.2. Comparison of All JVM

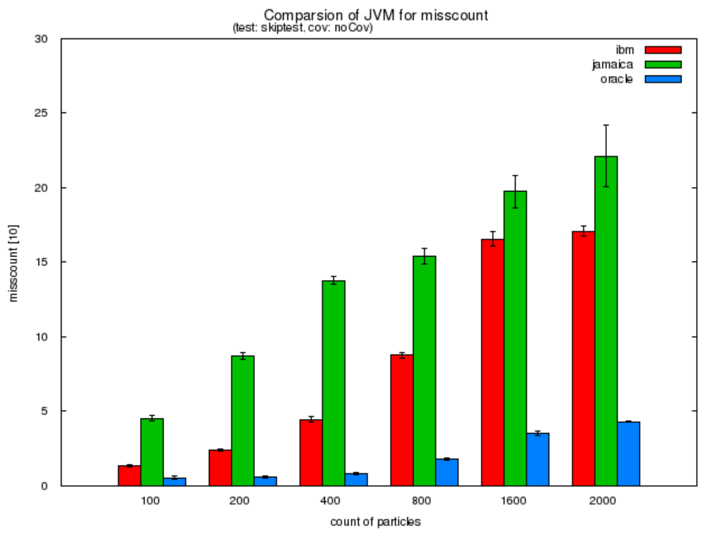


Figure 5.17. Histogram of average missCount

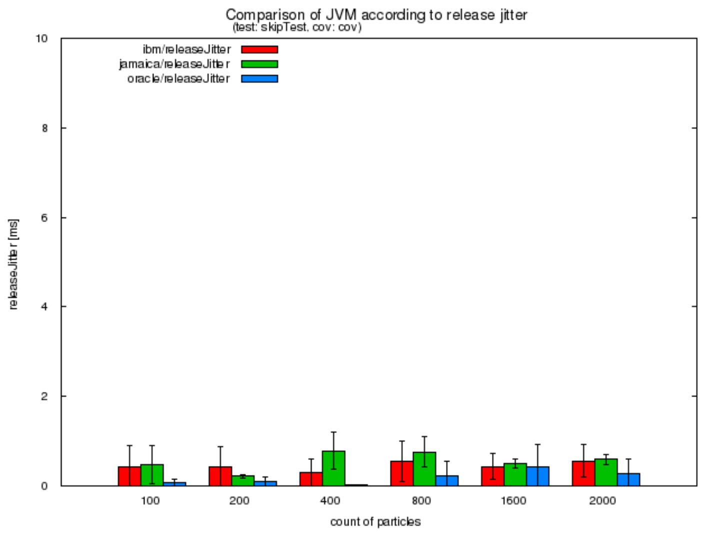


Figure 5.18. Histogram of average release jitter

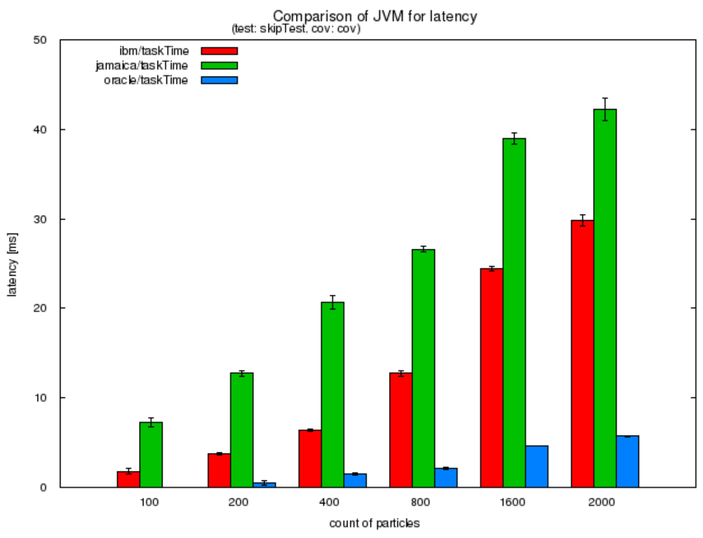


Figure 5.19. Histogram of average task latency

Best results are achieved by the JVM implementation of Oracle which results are diametrically different. Second is the IBMs implementation of hard real-time JVM, which can handle the interruption of unfinished tasks, but has good latencies of task processing. The Jamaica has the worst results in the measured metrics. Algorithm works only to the 800 particles and the deviation of position was high.

Chapter 6

Conclusion

I started working on this thesis when I did not have much experience and knowledge of programming for real-time systems and real-time systems in general. This topic interested me and I tried to understand it during work on the development of benchmark. At our college nobody using Java for programming of real-time systems. So that solving of problems,which are connected with RTSJ and programming practices, was not so easy.

I successfully rewrote the algorithm from C programming language to Java language and I demonstrated its functionality. I implemented the algorithm in several versions and tried to optimize it according to real-time programming principles. Along with my advisor, we have developed we have developed several metrics to find out which (or whether) they are suitable for comparison of real-time JVM properties.

From the measured results I created various graphs. Part of the benchmark is a project that is used to generate graphs automatically.

I wrote a web site providing the dynamic comparison of results which i used for evaluation.

During the work I have tested the real-time implementations from IBM, Oracle (Sun) and aicas. The performance on RTOS is worse than the performance on non real-time system. The latency, jitter and missCount of tasks is lesser for non real-time system. This was expected because of the higher overhead of real-time implementations of JVM. On the non real-time OS has benchmark better results but these results are not guaranteed. On a worst case scenario if the system is very busy or overloaded the results should be much worse than on RTOS.

The results of benchmark depends on the used algorithm and on my implementation of algorithm. In the comparison of JVM implementations has the best results Sun Java Real-Time System 2.2. The second was IBM Websphere for RealTime, but it can not handle the termination test. It could by caused by my poor implementation. The worst results has JamaicaVM from aicas because of high amount of missCount and average deviation. The task times and jitter were longest from the three tested JVMs. All the test was ran on RTOS which was not fully loaded. Only the benchmark ran along with standard programs of Ubuntu Linux and Gnome.

The results of this work and all source codes are public available in the git repository. The address is git@rtime.felk.cvut.cz:jmclbench.

Further development is necessary to provide another results. There is big space for impromevents of the benchmark especially in use of different GC configurations and doing the benchmark tests in fully loaded system. The benchmark should be expanded by implementation of algorithm in Safety Critical Java.

Literature

[1] *Programming Language Popularity (2011)*, <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

[2] Real-Time Operating Systems <http://en.wikibooks.org/wiki/Embedded_Systems/Real-Time_Operating_Systems>

[3] Brian Goetz, Robert Eckstein (2008), An Introduction to Real-Time Java Technology

[4] Gregory Bollella, Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, Mark Turnbull (2000), The Real-Time Specification for Java January 15 <http://www.rtsj.org/specjavadoc/book_index.html>

[5] Eric J. Bruno, Greg Bollella (2009), Real-Time Java Programming: with Java RTS

[6] Kanaka Juvva (1998), The Real-Time Systems Carnegie Mellon University, <http://www.ece.cmu.edu/~koopman/des_s99/real_time/>

[7] Greg Bollella, Kevin Russell (since 1998), JSR 1: Real-time Specification for Java <http://jcp.org/en/jsr/detail?id=1>

[8] Mark Stoodley, Mike Fulton, Michael Dawson, Ryan Sciampacone, John Kacur (2007), Cycle of articles about Real-time Java programming - Real-time Java, Part 1: Using Java code to program real-time systems <http://www.ibm.com/developerworks/java/library/j-rtj1/index.html>

[9] Frank Dellaert, Dieter Fox, Wolfram Burgard, Sebastian Thrun (1999), Monte Carlo Localization for Mobile Robots <www.ri.cmu.edu/pub_files/pub1/dellaert_frank_1999_2/dellaert_frank_1999_2.pdf>

[10] Dieter Fox, Wolfram Burgard, Frank Dellaert, Sebastian Thrun (1999), Monte Carlo Localization: Efficient Position Estimation for Mobile Robots, in Proc. of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)

[11] Sebastian Thrun, Dieter Fox , Wolfram Burgard , and Frank Dellaert (2001), Robust Monte Carlo Localization for Mobile Robots <http://robots.stanford.edu/papers/thrun.robust-mcl.pdf>

[12] Andrei Stanculescu (2009), Evaluation of the Monte Carlo Localization algorithm <rtime.felk.cvut.cz/dragons/repos/papers/Monte_Carlo_Localization.pdf>

[13] Wolfram Burgard (1998), Markov Localization, A Probabilistic Framework for Mobile Robot Localization and Navigation <http://www.cs.washington.edu/homes/fox/diss/diss.html>

[14] Maria Isabel Ribeiro, Pedro Lima (2002), Markov Localization <http://users.isr.ist.utl.pt/~mir/cadeiras/robmovel/Markov-Localization.pdf>

[15] Scott Chacon (2009), Pro Git <http://progit.org/book/>

[16] HOWTO: Realtime-Preempt Kernel <https://www.osadl.org/Realtime-Preempt-Kernel.kernel-rt.0.html#externaltestingtool>

[17] Cyclictest <https://rt.wiki.kernel.org/articles/c/y/c/Cyclictest.html>

Appendix A

Contents of CD

CD appended to this work contains following folders:

- janoumi8-2012-dip.pdf - Electronic form of this thesis in pdf format.
- project jMCLBench subprojects - All classes and source codes, sites, config files, results and pictures
- results - history of measured data with generated graphs
- c - source codes of MCL localization algorithm and data for benchmark
- www - codes used for web comparison