Towards a High Integrity Real-Time Java Virtual Machine

Hao Cai and Andy Wellings

Department of Computer Science, University of York, UK
{haocai, andy}@cs.york.ac.uk

Abstract. This paper defines a run-time architecture for a Java Virtual Machine (JVM) that supports the Ravenscar-Java profile (RJVM). This architecture introduces an early class loading and verifying model that can facilitate the predictable efficient execution of Java applications, detect program errors at the initialization phase and prevent errors occurring during the mission phase. A pre-emptive fixed priority scheduler is provided which supports the immediate ceiling priority protocol to facilitate efficient resource usage. An economical predictable memory management model based on heap, immortal and linear time scoped memory (LTM) is presented. The integration of our proposed runtime architecture and the KVM is discussed.

1 Introduction

The combination of object-oriented programming features, the lack of any mechanisms for memory reclaiming (thereby necessitating the virtual machine to undertake garbage collection) and the poor support for real-time multi-threading are all seen as particular drawbacks to the use of Java in high integrity real-time systems (HIRTS). The Real-Time Specification for Java [1] (RTSJ) has introduced many new features that help in the real-time domain. However, the expressive power of these features means that very complex programming models can be created, necessitating complexity in the supporting real-time virtual machine. Consequently, Java, with the real-time extensions as they stand, seems too complex for confident use in high-integrity systems. Ravenscar-Java [3][4][5] has been proposed in order to counter these effects.

1.1 Ravenscar-Java

A Ravenscar-Java program consists of a fixed number of schedulable objects (real-time threads and asynchronous event handlers) that are created immediately after the program begins its execution. A program is considered to have

- an initialization phase — where all classes are loaded and all the schedulable objects (and all other permanent objects) are created; typically there are no hard time constraints associated with this phase of execution,
- an execution phase — where all schedulable objects execute under time constraints.

Schedulable objects in Ravenscar-Java do not terminate and, hence, the program does not terminate. All schedulable objects have unique fixed priorities and they are executed using pre-emptive priority-based scheduling. Sporadic entities are released by a single event that can be either software generated or hardware generated (by an interrupt).

Communication between schedulable objects is via shared data. Where data cannot be accessed atomically, it is encapsulated by synchronized methods that enforce mutual exclusion. Ceiling priority inheritance (called priority ceiling emulation by the RTSJ) is used to bound the time that a high-priority schedulable can be blocked by a low-priority schedulable object accessing shared data. Ravenscar-Java requires predictable memory management, hence, only LTM can be used for dynamic object creation.

1.2 J2ME

To address the demands of embedded systems and consumer electronics, the Java 2 Platform Micro Edition (J2ME) [7][8] has been introduced by Sun. This defines the three layered architecture illustrated in figure 1 [4][8]:

- a virtual machine layer which usually is implemented on top of a host operating system,

- a configuration layer which defines the set of Java language features, a minimum set of virtual machine features and the available class libraries that can be supported by a particular implementation platform (for example a mobile phone),
- a profile layer which defines a minimum set of Application Programmers Interfaces (API) targeted at a particular application domain.

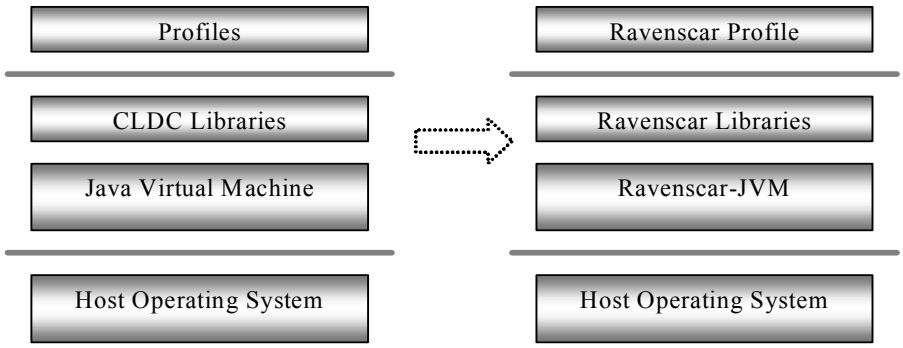

Figure 1 The architecture of J2ME and Ravenscar-Java

A configuration layer, called Connected, Limited Device configuration (CLDC) [8] has been defined for small, resource-constrained mobile devices (mobile phones, pagers, personal organizer etc.) typically with a memory capacity of up to 512 KB. The K (kilo bytes) virtual machine (KVM) is a virtual machine specifically designed to support the CLDC. The restrictions imposed on the Java language and the virtual machine include: no support for floating point operations, no native interface, no user-defined class loaders, no thread groups and daemon threads, no object finalization, etc.

Ravenscar-Java can be considered as a profile layer for the high integrity real-time application domain. It is supported by a Ravenscar-JVM (RJVM) which is based on KVM. The RJVM can not only preserve the portability of KVM but also targets the temporal non-deterministic execution problems of KVM. The Ravenscar-Java profile defines a set of APIs for HIRT Java applications and also defines a reliable and analyzable computational model. This results in the following restrictions on the RJVM:

- No dynamic class loading during the application execution phase,
- Pre-emptive priority-based scheduling with immediate ceiling priority protocol (ICPP),
- No garbage collection during the execution phase.

This paper discusses how the KVM can be modified in order to support Ravenscar-Java. Section 2 analyses the run-time architecture of KVM and identifies which features are unsuitable. A run-time architecture for the RJVM is then proposed, in Section 3, which targets these problem areas. Sections 4 and 5 describe a prototype implementation of the architecture and an experimental evaluation of its performance. Finally, our conclusions and future work are given in Section 6.

2 The Run-Time Architecture of K Virtual Machine (KVM)

In this section, the run-time architecture of KVM is presented. The three main unpredictable features of KVM: the dynamic class loading and verifying model, the threading model and the memory management model are discussed in detail.

2.1 The class loading model

The KVM supports a dynamic class loading, linking and verifying execution model, illustrated in Figure 2. Each instance of the KVM has a class loader that is responsible for loading the classes of an application. It then passes the loaded data to a linker, which parses the constant pools, fields, methods and other attributes contained in the loaded classes into the run-time class structures. The execution engine of KVM, which is an interpreter, starts as long as the essential classes (this includes some essential system classes and the main class) are loaded. When executing the application, the interpreter may access some instructions that refer to the constant pool that

contain symbolic references. The resolver asks the class loader to dynamically load the referenced types and replace the symbolic references in the constant pool to direct references. When the interpreter first uses a class, it needs to be verified. Substantial effort is needed to verify every Java byte code of all methods in the class to ensure their type safety at run time.

The execution of instructions resulting in dynamic class loading and class verifying, are some of the main sources of unpredictability and inefficiency in the execution of Java applications. To increase predictability and efficiency, these dynamic loading and verification features should be removed.

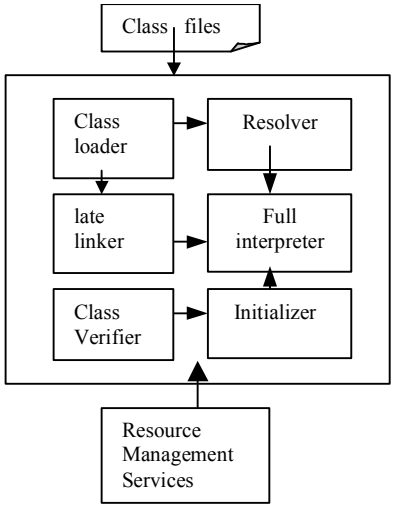

Figure 2. A dynamic class loading and verification run-time architecture in Java

2.2 The threading model

KVM has a simple pre-emptive, variable quantum, round-robin scheduling model, in which all the active threads are stored in a circular linked list. It supports 10 priority levels. When a new thread arrives, it is linked to the end of the list and is scheduled to run. Each thread in the list is given a priority, which indicates to the interpreter the maximum number of bytecodes the thread may execute during its quantum. Each thread also has four runtime variables (a stack pointer, a frame pointer, a local pointer and an instruction pointer) and a stack which store its execution context. When a thread comes into existence, its runtime context is loaded into the four virtual registers of the interpreter: and also its quantum (timeslice) is set to `1000*its priority`.

When a running thread is blocked for a period of time, it is put into a timer queue. The threads in the timer queue are sorted by their wakeup times. When a thread switch occurs, first, the timer queue is checked from its first thread. The threads in the timer queue with their wakeup times due are moved from the timer. If a thread in the timer queue was blocked inside a monitor, the thread will be put into the wait queue for the monitor lock. If the thread in the timer queue was blocked by sleeping for a period of time, it will be put into the runnable queue.

KVM has an approach that attaches real monitor objects to object instances only when they are really needed. The header of an object instance stores information about the monitor that is associated with that object. There are four possible types of access to a synchronized object. First, no synchronization is required for the object. Second, the object is locked and accessed by only one thread once. Third, the object is locked and accessed by only one thread multiple times. Fourth, the object is locked by one thread and is requested by other threads. Only the latter requires a real lock to be used. Consequently, when multiple threads try to execute synchronized method calls or `MONITORENTER` bytecodes, a monitor object illustrated in figure 3 is created to attach to the associated object.

Each monitor contains a pointer that refers to the current holder of the monitor and also holds a wait queue and a condvars queue in which those threads are unconditionally or conditionally waiting for the monitor to be released. When a thread tries to access an object associated with a monitor, a check will be performed. If the thread holds the monitor, it continues to execute. If the monitor is held by another thread, this thread is queued on the wait queue. A first-come-first-serve scheduling model is used for controlling the wait queues. A thread may request a timeout whilst trying to acquire a monitor lock, this thread is first put into the timer queue and the condvars queue of the monitor. After the period, the thread is moved to the wait queue.

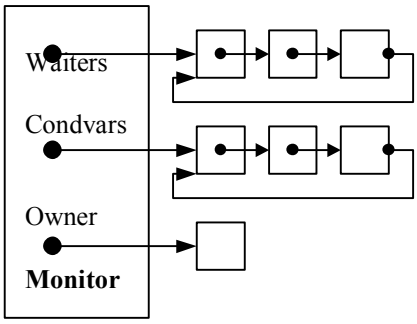

Figure 3. The structure of a monitor [10]

2.3 Analysis of the threading model in KVM

All runnable threads are put in the runnable thread queue and are scheduled by the policy "first come first served". The priority just simply enables the high priority threads to get a larger quantum. Priority inversion may frequently occur.

Table 1 Example Thread set for KVM.

Thread	Priority	Execution sequence	Release Time (instructions)
a	1	EQQE	0
b	2	EEEE	2000
c	3	EQQEE	2000

To illustrate priority inversion in KVM, consider the executions of three threads: *a, b,* and *c*. Assume they arrive in the order of *a, b, c* and thread *a* and thread *c* share the resource (synchronized object), denoted by the symbol Q. The priority of thread c is the highest and that of thread *a* is the lowest. Table 1 gives the details of the three threads and their execution sequences; in this table 'E' represents the execution of 1000 bytecode instructions and 'Q' represents the execution of 1000 instructions whilst holding Q's lock.

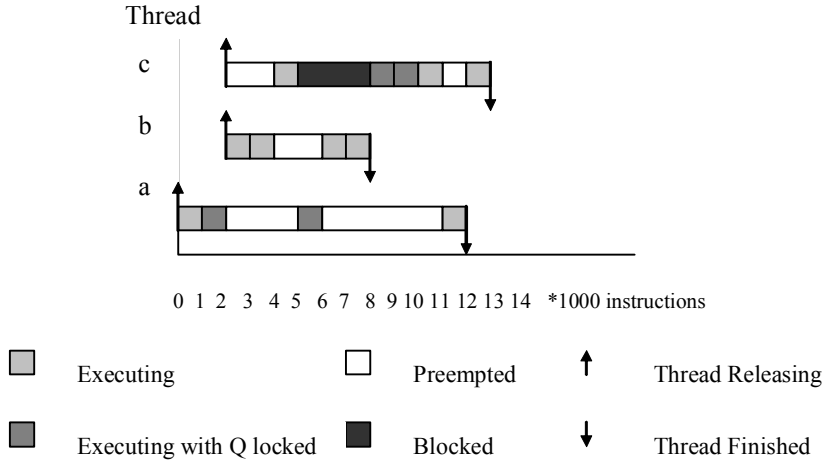

Figure 4 Example of priority inversion with a shared resource in KVM

Figure 4 illustrates the execution sequence for the three threads. The quantum of threads are decided by their priority, so thread *b* has the double quantum of thread *a*, thread *c* has triple quantum of thread *a*. At each turn of running, thread *a* will execute 1000 instructions, thread b will execute 2000 instructions and thread c will execute 3000 instructions. Thread *a* is released first, executes and locks Q. Thread *b* and c are released at the same time. However, b arrives earlier than c, so b is scheduled to execute and finishes its 2000 instructions. Thread *c* is scheduled to run, it executes for 1000 instructions and then it is blocked when it tries to acquire the lock of Q, which is currently held by thread *a*. At the second iteration of the three threads, thread *a* executes

1000 instructions, releases Q and it is pre-empted by thread *b*. Thread *b* executes another 2000 instructions and finishes at time 8000. Thread *c* is scheduled to run, it acquires the lock of Q and executes its 3000 instructions. It is pre-empted by *a* after it finishes its quantum. At the third iteration of running, thread *a* executes another 1000 instructions and finishes at time 12000. Thread c then finishes its last 1000 instructions at time 13000. Priority inversion is very severe in that the lower priority threads *a* and *b* both finish before the highest priority thread *c*.

The threading model in KVM has the disadvantages of severe priority inversion and frequent context switches. It is not suitable threading model for high-integrity real time systems.

2.4 The memory management model

The KVM uses a simple traditional garbage collection (GC) algorithm called Mark-and-Sweep collection [10][12]. At run time, the KVM is initialized with a chuck of memory called the heap. Stage 1 in figure 5 illustrates the status of an initialized heap. The newly created objects are allocated from the heap during program execution. Each object has a 32-bit header of which 24 bits is for the object size, 6 bits is for lock types and 1 bit is for the mark. Stage 2 in figure 5 illustrates the status that some objects have been allocated in the heap. When the heap is full, the GC starts to mark the objects in the heap. It starts marking the global root objects that consist of all the loaded classes, all threads and all thread stacks and some temporary root objects. For each live object, the GC recursively marks all those objects that are reachable from it. After marking, the GC sweeps the heap and all the free memory spaces are linked into a list of small chucks which is illustrated in stage 3 of the heap in figure 5. The linked chucks are used for later memory allocation. If the largest chunk is not large enough for a memory request, a compaction occurs. This moves all the live objects together, upgrades pointer in the live objects and links all the free small chunk into a large free block, then the large block is used for later memory allocation.

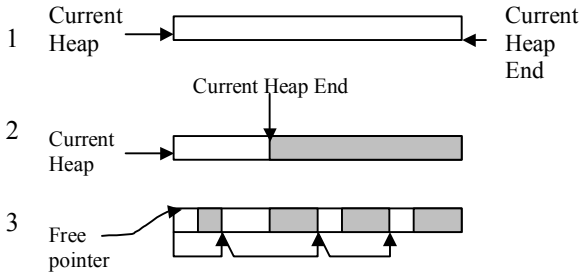

Figure 5. The three stage of a heap in KVM

The mark and sweep GC in KVM traces live objects in the mark phase and sweeps and compacts linearly throughout the entire heap. Let M be the size of the heap, R be the amount of live objects and a, b be constants. The time complexity of the mark-sweep collector can be approximated by [12]:

$$t = aR+bM \quad (1)$$

The amount of space recovered by a garbage collection is:

$$m=M-R \quad (2)$$

Define the efficiency, e, as the amount of memory reclaimed in a unit time [12]

$$e=\frac{M-R}{aR+bM}=\frac{1-r}{b+ar} \quad (3)$$

where r = R/M is the residency of the program. Let *gc* be the amount of time to perform garbage collection (GC), *overall* be the execution time of the applications and *c* be the allocation rate (memory allocated in a unit time). Define the GC overhead, o, as:

$$o= gc / overall = \frac{c \cdot \frac{1}{e} \cdot overall}{overall} = \frac{c}{e} = c \cdot \frac{b+ar}{1-r} \quad (4)$$

The mark and sweep GC in KVM has the disadvantage that it is not efficient and is highly unpredictable. The larger the heap, the longer the GC pauses as shown in equation (1). This is because the larger heap the more live objects it holds and the larger the space that needs to be swept and compacted. The behavior of GC is highly unpredictable and it depends on the status of virtual machine and dynamic features of Java programs, for instance, the residency of the program, r shown in the equation (3)(4).

3. Run-Time Architecture of a RJVM

We have discussed the problems of adapting KVM so that it supports Ravenscar-Java. Dynamic class loading and verifying provide enormous uncertainty for the application's execution. The thread model has frequent context switches and suffers from priority inversion which prevents efficient resource usage. The high overhead of GC and long GC latency make it unsuitable for HIRTS. In this section, we will present our solutions to these problems. A new class loading and verifying model, a real-time thread model and a predictable memory management model for a RJVM are proposed.

3.1 A class loading and verifying model of a RJVM

In the initialization phase of a Ravenscar-Java application, a static `NoHeapRealTimeThread` executes first and performs the initialization of the application before the mission phase is carried out. It includes initialization of all real-time threads, memory objects, event handlers, events and scheduling parameters. In the mission phase, the application is executed and multithreading is dispatched based on the imposed scheduling policy. It is difficult to have a reasonable upper bound execution time for the Java primitive operations that may result in dynamic class loading and verifying during the application's mission phase. The Ravenscar-Java requires that all the classes needed in the application be loaded during the initialization phase.

A new run-time architecture is proposed to support this illustrated in figure 6. All the classes needed by the application are loaded, linked into the RJVM and also are verified at the initialized stage of the application. The virtual machine starts to load the essential system classes and then loads the main class of the application. All the classes that can be reached from the main class will be loaded before the interpretation. During linking classes, not only the syntax of classes but also every Java byte code of every method in the classes is verified [9][15]. During the verification of each Java byte code, the instructions that need dynamic resolutions of the constant pool entries are replaced with their correspondent fast instructions and these constant pool entries are resolved. For example, when verifying a `new` instruction, the constant pool entry requires that the `new` instruction will be resolved and the `new` instruction be replaced with `new_quick` instruction. During the execution stage of the application, no dynamic loading, linking and verification will be performed. All the instructions that may invoke dynamically loading and linking classes are replaced with their correspondent quick instructions. No dynamic verification of classes will be needed during the interpretation of Java byte code.

The early loading by the verifier simplifies the interpretation of Java byte code. All the instructions associated with dynamic class loading such as `getfield`, `new`, `invokevirtual` etc can be replaced with their correspondent quick instructions such as, `getfield_quick` `new_quick`, and `invokevirtual_quick` etc. 24 slow instructions can be removed from the instruction set of the interpreter. This model has the advantages that:

- The instruction replacement is performed at the initialization phase of the execution, so it is backward compatible with other Java applications for KVM and performance and predictability are greatly improved.
- The simplified instruction set can facilitate more accurate worse case execution time (WCET) analysis of Java bytecode execution [6].
- The Java primitive operation, which may result in dynamic class loading and verification, has been replaced with their correspondent quick instructions and can be given a reasonable upper bound WCET.
- Comprehensive type safety checks become possible because all the related classes are available for the verifier before the mission phase.

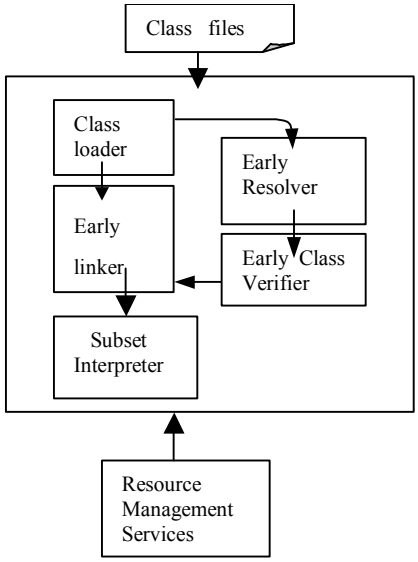

Figure 6. A early class loading and verifying runtime architecture in RJVM

3.2. A scheduling model for a RJVM

To solve priority inversion problem in the KVM, the thread system and the lock system need to be modified. Inside our RJVM, instead of a single runnable queue, a list of runnable queue is created and each priority has a runnable queue [16]. The scheduling of threads starts from the highest priority queue to the lowest priority queue. The scheduling policy of the same priority queue is pre-emptive round-robin scheduling. The newly created threads are put into the end of their correspondent runnable queue. The next thread to run when a context switch occurs is the head of the non-empty highest priority queue.

The monitor wait queue also needs to be modified from a "first come first serve" queue to a priority-based queue. The highest priority threads in the wait queue of a monitor will hold the monitor instead of the first waiting thread when the owner releases the monitor. The immediate ceiling priority protocol [13][14] is also provided to avoid the priority inversion when shared resources are involved between threads.

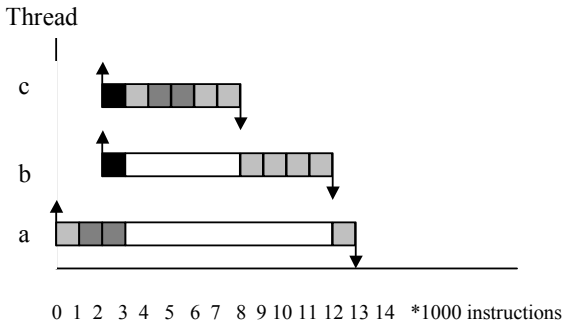

Figure 7 Example of modified thread system in KVM

With the new scheduling, the execution sequence of the three threads in figure 4 will change to that illustrated in Figure 7. The shared resource Q will be given a ceiling value of 4, which is higher than the maximum priority of processes that use it. When thread *a* is locking the resource Q, thread *a* inherits the ceiling value of resource Q, here thread *a* is scheduled to run with dynamic priority 4. This ensures that thread *a* is not pre-empted by the threads with priority 3 when thread *a* holds lock of Q, so *a* can finish and release Q as soon as possible. When thread *a* is releases the lock of Q, it restores its previous dynamic priority and thread *c* acquires the lock of Q and is scheduled to execute. After c finishes, *b* is scheduled to run and finish, then *a* executes and finishes. No priority inversion occurs.

3.3 A predictable memory management model for a RJVM

A predictable memory management model based on heap, immortal and LTM [2][5] is shown in Figure 8. This model addresses Ravenscar-Java concerns for efficiency and predictability and yet the programmer can benefit from the Java object-oriented programming style while avoiding the side effect of garbage collection. At the initialization phase of a virtual machine, a heap and a block of memory for linear time memories (LTM) are allocated. The heap is used to allocate objects that include the loaded classes, the String objects and some temporary objects etc. During this phase, the heap is subject to garbage collection. Just before the start of interpreting, the heap is compacted and is transferred to immortal memory, which is not subject to garbage collection. During the initialization of Java applications, all new created objects including LTM objects are created in immortal memory. However, the memory space that LTM objects refer to are allocated from the memory for LTMs. During the mission phase, all the new created objects are allocated either in immortal memory or linear time memory which are not subject to garbage collection. Each thread is associated with a memory area, either a LTM or the immortal memory area at any instance. When a thread is scheduled for execution, its memory area will become the current memory area and all objects dynamically created are allocated in the area during the program execution.

This approach has the advantages of more efficient and predictable memory usage compared with the model of KVM. The garbage created at the initialization phase can be removed by using a traditional garbage collected heap. A substantial memory could be wasted without collecting garbage objects created during the loading of classes. By the early loading of classes and collecting and compacting the heap at the end of the initialization phase, this memory will be saved for use by the application. Experiments indicate that the memory space saved is around 1.3 times of the application size. The early loading of all classes can facilitate the efficient memory usage because a large amount of garbage objects can be collected after finishing the class loading. For devices with limited memory, the collection of the heap is essential during the initialization phase of the application.

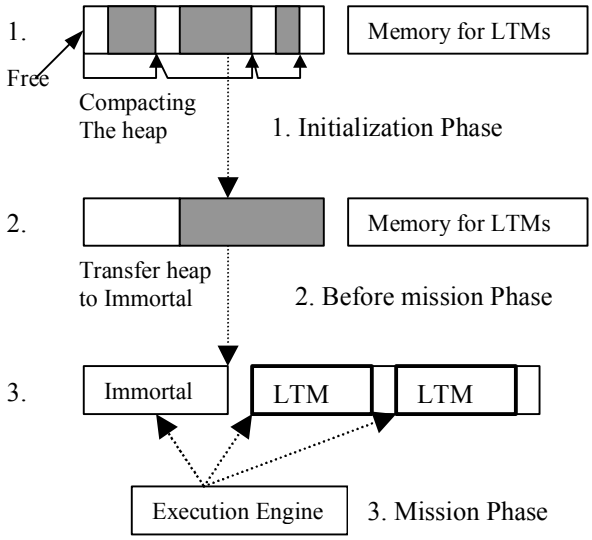

Figure 8. A predictable memory management model

During the execution phase, a linear time memory can be associated with and periodically reused by a periodic thread. The reason for separating the memory for LTMs from the immortal memory is to ease the runtime assignment check. It will become much easier to tell whether an object created is in the immortal or in the LTM block. This can simplify the runtime to check that an object created in LTM is not assigned to objects created in immortal memory or heap.

4. Implementation Issues

In this section, the integration of our proposed runtime architecture and KVM is illustrated in detail.

4.1 Implementation of ICPP

The Ravenscar-Java profile requires ICPP to be implemented in the run-time system. KVM has a lightweight monitor system discussed in 2.2 that creates real monitor objects to object instances only when they are really needed. To implement the ICPP, one attribute (`int ceilingPriority`) is added to each shared resource lock, not to every object and two attributes (`int org_priority[MaxOjectLocking], int CeilingCount`) are added to the thread class. Integer `ceilingPriority` holds the ceiling value of a shared resource which is greater than the maximum priority of the threads that can use it. An integer array `Org_Prioirty` stores a list of the ceiling value that the thread is inherited from the shared resources. `CeilingCount` is the number of the priority inheritance occurrences.

Each thread starts with its static default priority as its dynamic priority. When it tries to acquire an available shared resource, the ceiling priority of the resource is compared with the thread's dynamic priority. If it is higher than the thread's dynamic priority, priority inheritance occurs, the thread's current dynamic priority is stored into the `org_priority` array and the number of the `CeilingCount` is increased. The thread's current dynamic priority inherits the ceiling priority of the shared resource. The thread will execute with the dynamic priority until it releases the shared resource. When the thread releases the resource, its previous dynamic priority is restored and the `CeilingCount` is decreased. If a thread is trying to acquire a shared resource whose ceiling priority is lower than the thread's dynamic priority, a monitor holding error will occur. Ravenscar-Java does not allow a thread holding a higher dynamic priority to enter a lower ceiling priority monitor.

Two attributes (`int NoLock , int NoLockCount`) are also added to each runtime thread and one attribute (`int NoLock`) is also added to the resource monitor to support `NoLockPriorityCeilingEmulation` that prevents a thread suspending when it is holding `NoLock` shared resources. When a thread is holding a NoLock resource, the thread `noLock` flag is set to 1 and its `noLockCount` is increased. When the thread releases the `noLock` resource, the `noLockCount` of the thread is decreased. The `NoLock` flag of the thread will set to 0 when its `NoLockCount` becomes 0. A thread with the `NoLock` flag set to 1 is not allowed to suspend.

4.2 Implementation of the Memory Management model of RJVM

Ravenscar-Java only defines one type of scoped memory area, Linear Time Scoped Memory (LTM) and it also requires that access to LTM areas must not be nested and LTM areas must not be shared between schedulable objects. All threads may use immortal memory or LTM during the run time. This implies that each thread can only be associated with one memory area, either a LTM or the immortal memory at any instance. The run-time instance of threads is extended and the runtime instance of memory area is created. Their data structures are illustrated in Figure 9.

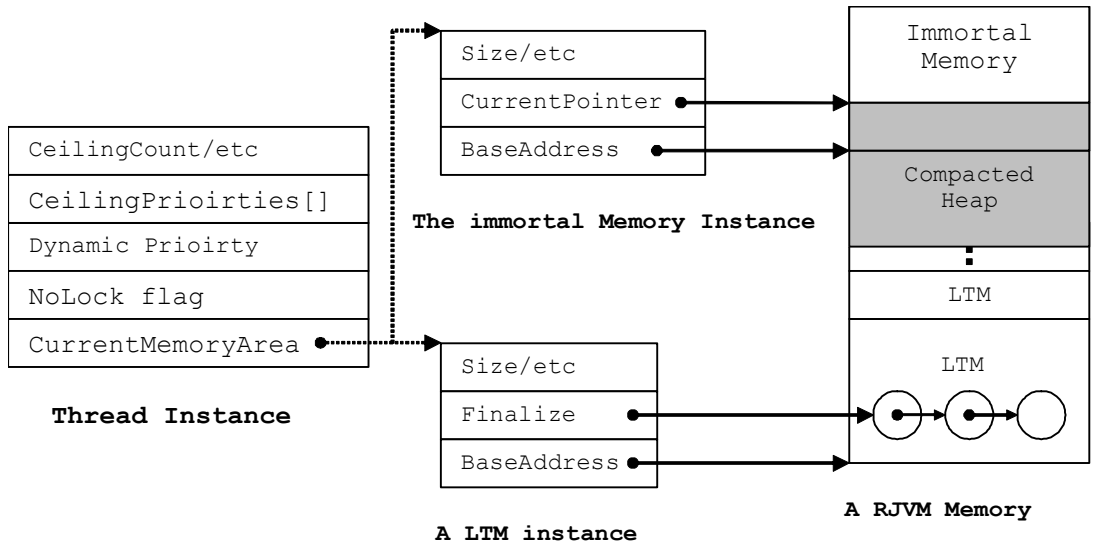

Figure 9. Runtime structures of thread and memory Area

Each runtime thread is extended to contain a memory area pointer (`MemoryArea*`) which points to the current memory area of each thread. The thread uses this to hold new created objects. The current memory area could be the immortal memory or a LTM. Each RJVM has only one immoral memory while multiply LTM areas could be created in the memory for LTMs. Each run-time memory area contains the information such as its maximum size, the free memory size, the pointer to a block of memory allocated from the immortal memory, the current free pointer that contains the address of the free memory etc.

Each LTM also contains a list of finalizable objects. When an object of a class which contains a finalize method is created in a LTM, the object will be put in the finalizable list of the LTM. When the memory of a LTM is to be reclaimed, the objects in its finalizable list are finalized first, then the LTM is restored to its initial state such as the current pointer points to its base address, remaining size is restored to the initial size of the LTM, the finalizable list is null etc. The LTM then can be reused.

5. Experiments Evaluation

This section evaluates the changes made to the KVM to form the basis of a RJVM. To evaluate the policy of collecting memory after loading all classes just before executing the application, four small applications are used. The size of the applications, memory collected and the ratio of application size with memory saved are illustrated in table 2. By compacting the heap before the start of the main method, around 1.3 times the application size memory could be saved before the mission phase of the applications.

Table 2 Memory reclaimed after loading all classes before the mission phase

Size of applications	Garbage collected	Ratio
2169	2976	1.37
2341	3128	1.33
4162	6196	1.34
5276	6884	1.30

A benchmark with three threads denoted by the symbol *a*, *b* and *c* is used to evaluate the thread model in KVM and in our RJVM. The priorities of the thread threads, their order arrived, their time released, their computation block in $1*10^6$ instructions and their microsecond response time both in KVM and in our RJVM are detailed in Table 3. No garbage collection occurs during the execution.

Table 3 Example thread set and their response time in KVM and RJVM

Thread	Priority	Order arrived	Time released	Computation block	Response time in KVM	Response time in RJVM
a	1	1	0	890	177846	177806
b	2	2	0	1780	177876	148194
c	3	3	0	2670	177896	88908

The scheduler in KVM has a very high context switch overhead which results in quite inefficient execution of the Java applications. It is not a suitable model for multithreads applications. Our RJVM provides a predictable and efficient multithreads execution environment.

The garbage collection benchmark used to evaluate the memory management in KVM and our proposed memory model is the tree benchmark which is an adaptation made by Boehm from the Ellis and Kovac benchmark [17]. It is modified to be compatible with the KVM. The allocation rate of the modified tree benchmark is about 13.75 K bytes per microsecond. The number of garbage collection pass, the microseconds spent in garbage collection, the microseconds spent in execution and the percentage overhead introduced by GC in KVM for this benchmark is illustrated in Table 4. From the equation: o= $c \cdot \frac{b + ar}{1 - r}$ in section 2.4, we can understand the behavior of GC in KVM shown in figure 10. Horizontally, for a single application, the smaller heap, the larger residency of the program, r, and then the larger GC overhead. Vertically, for different applications, the larger allocation rates, *c*, the larger GC overhead.

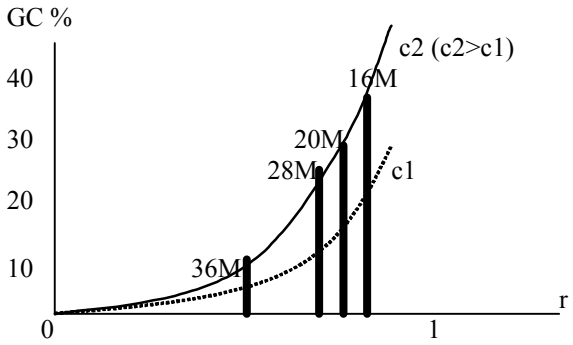

Figure 10 The behaviour of GC in KVM

Table 4. The garbage collection overhead of the tree benchmark

Memory heap	GC pass	Collection time	Execution time	% overhead
16M	4	1020	3365	30.31
20M	3	1011	3355	30.13
28M	2	961	3325	28.90
36M	1	341	2694	12.66
64M	0	0	2424	0

The behavior of the tree benchmark in our implementation of RJVM is illustrated in table 5. Predictable and efficient execution is achieved. During the mission phase, no garbage collection will occur. By separating immortal memory and linear time memory (LTM), it is relatively easier to check that an object created in LTM is not referenced by objects created in immortal memory. The runtime check overhead (RCO) is the amount of the time to perform runtime check with the execution time.

Table 5 The execution time of the tree benchmark in the RJVM

Heap/ Immortal	LTM	Execution Time without RC	Execution Time with RC	RCO %
16M	12 M	2504	2704	7.4

6. Conclusion

In this paper, a runtime architecture of RJVM has been presented. An early class loading and verifying model can detect the program errors at the initialization stage; it will reduce the failures of the system during the mission phase which might have catastrophic consequences. A more deterministic threading model reduces the context switch overhead of the KVM and facilitates efficient resource usage by adopting the immediate ceiling priority protocol. A memory management model based on heap, immortal and linear time memory provides efficient and predictable memory usage without sacrificing the automatic memory management features of Java. The experiments show great improvements on predictability and efficiency when executing Java applications in our RJVM compared with the KVM. A more reasonable upper bound WCET of Java Byte code can be obtained by removing the effect of garbage collection and dynamic class loading and verification.

Reference

[1] G. Bollela, B.Brosgol, P.Dibble, S.Furr, J.Gosling, D. Hardin, and M.Turnbull, “The Real-time Specification for Java”, Addison Wesley,2000

[2] G. Bollella, K. Reinholtz, “Scoped Memory”, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002
[3] P. Puschner and A.J. Wellings, “A Profile for High-integrity Real-time Java Programs”, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2001
[4] J. Kwon, A.J. Wellings, and S. King, “Ravenscar-Java: A High Integrity Profile for Real-Time Java”, Proceeding of the Joint ACM Java Grande – ISCOPE 2002 Conference, 2002
[5] J. Kwon, A.J. Welling, and S. King, “Predictable Memory Utilization in the Ravenscar-Java Profile”, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003
[6] G. Bernat, A. Burns, and A.J. Wellings, “Portable Worst-Case Execution Time Analysis Using Java Byte Code”, Proceedings of the 12th EuroMicro Conference on Real-Time Systems, Stockholm, June 2000
[7] Sun Microsystem®, Java™ 2 Platform, “Micro Edition (J2ME™) Technology for creating Mobile Devices”, white paper, http://java.sun.com/products/cldc/, accessed March 2003
[8] Sun Microsystems, “Connected, Limited Device Configuration, Specification Version 1.0a”, http://java.sun.com/products/cldc/, accessed March 2003
[9] T. Lindholm and F.Yellin, “The Java Virtual Machine Specification(2e)”, Addison Wesley, 1999
[10] F. Yellin,, “Inside the The K Virtual Machine (KVM)”, Presentation slice, http://java.sun.com/javaone/javaone2001/pdfs/1113.pdf , accessed April, 2003
[11] B. Venners, “Inside the Java Virtual Machine”, McGraw-Hill, 1999.
[12] R. Jones, R.Lins, “Garbage Collection-Algorithms for Automatic Dynamic Memory Management”, John Wiley & Sons,1996
[13] A. Burns and A.J. Wellings, “Real-time Systems and Programming Languages: Ada 95, Real-Time Java and Real-time POSIX”, Addison Wesley Longmain , 2001
[14] L. Sha, el al, “Priority Inheritance Protocols: An Approach to Real-time Synchronization”, IEEE Transactions on Computers, 39(9):1175-1185, September 1990
[15] X. Leroy, “Java Bytecode Verification: An Overview”, Proceedings of CAV'01, number 2102 in LNCS, pages 265--285. Springer
[16] T.J. Wilkinson and Associates, “Kaffe: A Free Virtual Machine to Run Java Code”, Technical report, http://www.kaffe.org
[17] H. Boehm et al, “Tree Benchmark for Memory Management”, http://www.hpl.hp.com/personal/hans_Boehm/gc/gc_bench.html, accessed April 2003,