

Bringing the full power of Java Technology to embedded realtime applications.

Fridtjof Siebert
siebert@aicas.com
aicas GmbH
Haid-und-Neu-Straße 18
76131 Karlsruhe, Germany
<http://www.aicas.com>

Java technology brings a variety of advantages for the development of software for embedded systems. Nevertheless, features like automatic memory management (garbage collection) in Java pose challenges that need to be solved by a Java implementation that is to be used in realtime systems.

This presentation gives an overview over different techniques used by Java implementations and their usefulness for realtime system development. It will be explained how a deterministic Java implementation can work and how it can bring the full power of Java's features even to time critical code.

1. INTRODUCTION

The use of Java technology is becoming more and more popular even in areas well beyond the original scope of this programming language. Even for critical applications in domains like automotive, avionic, industrial automation, telecommunication and medical, there is a need to profit from the advantages Java technology has brought to desktop and Internet applications.

These advantages include the higher productivity of the development process compared to traditional languages like C and C++, the platform independence that brings independence of hardware suppliers, the high reliability of software developed in Java and the flexibility Java brings through mechanisms like dynamic class loading.

Nevertheless, there are a number of drawbacks that prevent the use of Java in deeply embedded realtime or safety critical systems. The main problem areas are the high demand of memory and the poor runtime performance that is provided by current implementation and the lack of realtime guarantees that are required in critical systems.

2. THE BIGGEST PROBLEM: GARBAGE COLLECTION

The biggest problems for the application of Java in time critical systems are caused by the use of automatic garbage collection to provide safe memory management. Otherwise, the garbage collector is required since it forms the basis of the safety and security mechanisms.

The garbage collector automatically detects mem-

ory that cannot be referenced by the application program and hence can be freed and reused for future allocations. This provides a solution to the problem of memory leaks due to forgotten *free()*s and dangling referenced that can cause accesses to memory regions that were *free()*d too early.

The garbage collector in a Java system is typically implemented as a cyclic process that consists of several phases. This process is illustrated in Figure 1.

The four phases of a mark-sweep-compact garbage collector are explained in the following sections.

2.1 GC-Phase 1: Root Scanning

In this phase, all objects on the Java heap that are referred to by references stored outside of the heap, e.g., references in local variables, are marked, such that their memory won't be reclaimed during the current garbage collection cycle.

2.2 GC-Phase 2: Mark

Once all object referenced from outside have been marked, the mark phase continues to mark objects that have not been marked yet and that are referenced by already marked objects. Consequently, a wave of marked objects advances through the heap during the mark phase until there are no remaining unmarked objects referenced by an already marked object.

2.3 GC-Phase 3: Sweep

Once the mark phase finishes, no unmarked objects are referenced by any marked objects, hence all un-

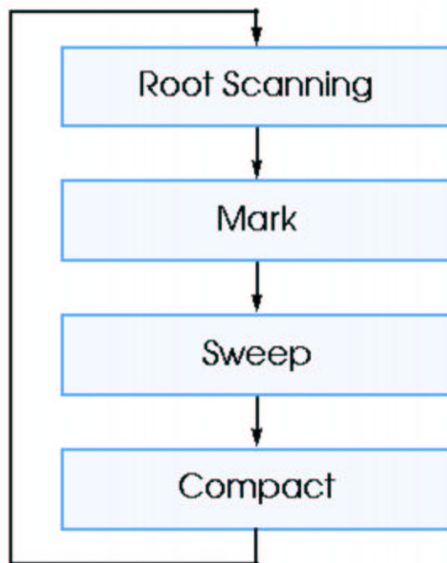


Fig. 1. Phases of a cyclic garbage collector.

marked objects cannot be reached through any sequence of referenced originating in the root references. The unmarked objects can no more be accessed by the application and their memory can be freed.

During the sweep phase, the memory of the remaining unmarked objects is consequently added to the free list.

2.4 GC-Phase 4: Compact

After the sweep phase, the memory of unused objects was freed. However, the memory might be scattered through the heap in small chunks that cannot be used for larger allocation requests. It is hence necessary to defragment, i.e., to move all allocated memory such that the free memory forms a contiguous range available to allocation of arbitrary sizes.

2.5 GC execution

The whole garbage collection process needs to be executed while the actual Java application is running. The garbage collector is hence typically run as a separate thread, but an incremental garbage collection might also be performed within the application threads.

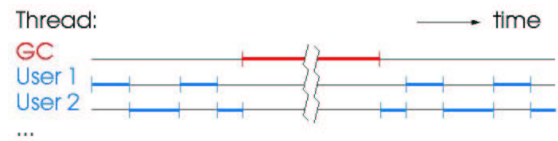


Fig. 2. User threads interrupted by a garbage collector thread.

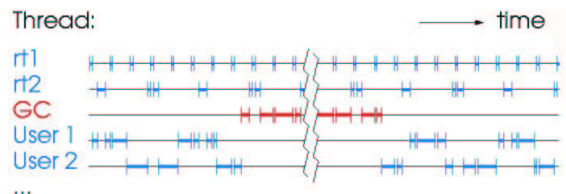


Fig. 3. 'Real-time' Java extensions providing realtime threads with priority higher than that of the garbage collector.

3. THREADS IN JAVA IMPLEMENTATION

Classical Java implementations that use a separate thread for garbage collection permit this threads to stop all user threads of a running application to perform garbage collection work. The resulting pause is illustrated in Figure 2.

3.1 'Real-time' Java extensions

To enable the development of realtime software even in the presence of a garbage collector thread that can interrupt the execution of user threads in an unpredictable way, extensions the Java specification such as the Real-Time Specification for Java were defined. These specifications define new thread classes like *RealTimeThread* or *NoHeapRealTimeThread* that are less affected by the preemption through the garbage collector thread. These threads run at priorities logically 'higher' than that of the garbage collector. Figure 3 illustrates this.

This solution permits the development of time critical applications, but it also provides a number of difficulties for the developer. The application needs to be separated strictly into a realtime and a non-realtime part, while the communication between these parts is very restricted. Realtime threads cannot use memory operations like allocation of objects that use the normal garbage collected heap. According to the Real-Time Specification for Java, threads that are not preempted by garbage collection activity (threads of class *NoHeapRealTimeThreads*) can't even access the normal Java heap.

Synchronisation between the realtime and non-realtime threads is not straightforward either. The Java synchronisation mechanism through the *synchronized* keyword will cause realtime threads to be

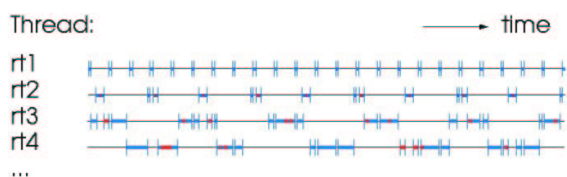


Fig. 4. *Realtime Garbage Collection enables all threads to behave like realtime threads.*

potentially blocked by garbage collection activity if they synchronise on an object whose monitor might be entered by a non-realtime thread.

3.2 Realtime Garbage Collection

In a system that uses true realtime garbage collection, the distinction between realtime and non-realtime threads is not required. Instead, no threads are blocked by garbage collector activity. Threads are scheduled according to their priority or other scheduling mechanism that is used in the system. Figure 4 illustrates this behaviour.

The realtime garbage collector performs its work in a predictable way within these threads whenever memory is allocated. Garbage collector activity must be interruptible any time to permit pre-emption by more urgent threads.

To implement realtime garbage collection mechanisms, a number of technical challenges must be overcome. The most difficult parts are the root scanning phase of the garbage collector, the memory fragmentation problematic and the determination of worst case execution times for successful memory allocation.

3.2.1 Root Scanning. Root scanning typically involves stopping of all threads such that the references stored within the runtime stacks of the threads can be found. This requires stopping of the application for times that are not acceptable in demanding realtime environments.

A means to completely avoid the root scanning phase overhead is to use a Java compiler and a virtual machine that creates additional code that ensures that all root references are marked for the garbage collector and the root scanning phase is not required any longer.

3.2.2 Dealing with Fragmentation. A reliable system needs to actively fight fragmentation of memory to ensure that it can run correctly for long periods of time. Any system that allocates memory of arbitrary sizes might otherwise fail due to fragmentation of memory.

However, defragmenting the heap through compaction of allocated memory is not easy, objects

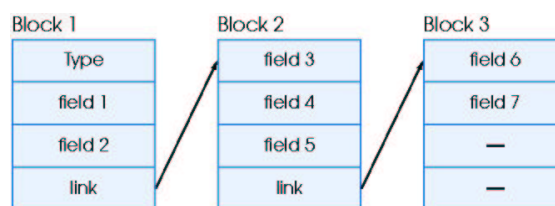


Fig. 5. *Avoiding the need for memory compaction using fixed-size blocks.*

typically need to be moved atomically causes long pauses since objects can be fairly large (e.g., Java objects can be arrays of arbitrary sizes).

A solution to the fragmentation problematic that completely avoids the need to compact memory is the use of fixed-size blocks that are never moved. Java objects can then be constructed out of these blocks even if they are not contiguous in memory. Figure 5 illustrates how an object can be constructed out of several blocks. Surprisingly, the use of fixed-size blocks results in performance similar to the use of a compaction mechanism since there is no need to provide for changing object locations.

3.2.3 Configuring a Realtime Garbage Collector. To be able to determine worst-case execution times for memory allocation operations in a realtime garbage collector, one needs to know the memory required by the realtime application. Automatic tools can help to determine this value. The heap size can then be set selected to give sufficient headroom for the garbage collector, while a larger heap size ensures a shorter execution time for allocation. Tools like the analyzer in the JamaicaVM help to configure a system and find suitable heap size and allocation times.

4. FURTHER CHALLENGES FOR A REALTIME IMPLEMENTATION

Even though the garbage collector is the most difficult obstacle for a realtime implementation, there are a number of other features in the Java language that need care.

4.1 Dynamic Calls and Type Checking

The dynamic type system and frequent use of dynamic calls and dynamic type checks (that are implicit in cast and array store operations) requires a deterministic implementation. It is possible to implement these features efficiently with constant overhead even for interface types that can be used for multiple inheritance, but they are not consequently used in classical Java implementations. A Java implementation to be used in realtime systems hence must use these techniques.

4.2 Synchronisation

Every Java object can be used to synchronise a sequence of Java statements, so every object must be equipped with a monitor that can hold the required information. To save memory, some implementations delay the allocation of special data structures for the monitor to the point when they are first needed. This technique can save memory, but causes difficult to predict execution times for monitor operation. A realtime implementation must instead preallocate the required information and use monitor inlining techniques to keep the memory overhead minimal.

5. REDUCING MEMORY DEMAND

For the use of Java technology in mass market embedded applications, the required memory needs to be minimal. A number of techniques have been developed to reduce the memory demand of Java application.

5.1 Using Configurations and Profiles

A number of Configurations and Profiles have been defined that provide subsets of the Java libraries and the Java language that are tailored for specific application domains and that remove functionality that is not needed in these domains to save memory.

The most important configurations are CLDC (connected limited device configuration) that provides a minimal subset of the Java language and APIs without features like graphics or 64-bit arithmetic. This configuration is targeting systems with less than 1MB of memory.

A more complex configuration was defined as CDC (connected device profile) that is to be used in larger systems with several megabytes of memory. It includes the full Java language and most of the standard class libraries, but not the `java.awt` graphics package.

5.2 Classfile Compaction

Java classfiles are relatively large and are not directly executable out of ROM such that they need to be duplicated into RAM before execution on a small embedded device.

An obvious source for memory demand reduction is hence the use of a more compact classfile format that is executable directly out of ROM without the need to copy information into RAM. This technique typically reduces the required ROM for classes by 50% and avoids nearly all of the RAM classes are copied into.

5.3 Smart Linking

To further reduce the memory demand, smart linking can be employed. In this technique, the Java application is analysed to find all methods and classes that are used during runtime. All unused methods can be removed to reduce the memory demand.

The reduction in memory required for Java classes is significant, it is typically in the order of 80-90%. Using the smart linking features of the JamaicaVM in conjunction with classfile compaction as mentioned in the previous section, the classes needed by the embedded caffeine benchmark can be reduced from 334630 bytes to only 23280 bytes.

However, this technique cannot be used in Java applications that use dynamic class loading or reflection, since the smart linker cannot predict what methods might be accessed by classes that will be loaded dynamically or through reflection.

5.4 RAM requirements

Reducing the amount of RAM needed by an application is mainly the responsibility of the developer. Nevertheless, the Java implementation itself can take a number of measures to keep its overhead low. Typical Java applications use many very small objects. This means that the per-object overhead introduced by the virtual machine and the garbage collector must be minimal.

The virtual machine needs to save information like object type, monitor, location of references, garbage collection and finalization state of an object with every object. This information must be encoded such that a minimal memory overhead results.

6. EXECUTION SPEED

The concept of the Java virtual machine suggests the use of an interpreter for the execution of Java applications. This is nevertheless too inefficient for many realtime and embedded applications, such that compilation techniques need to be applied.

Classical Java implementations use just-in-time or adaptive compilation to speed up frequently executed methods. However, this technique causes unpredictable execution times and is not applicable in time critical systems. Also, it requires the RAM and ROM overhead of having a compiler on the target system.

An alternative is load-time compilation. Here, the code of all classes is compiled at the time the classes are loaded. All executed code is compile and predictable fast. The drawback of such a system is the RAM and ROM overhead of the compiler that is required on the target system, the high overhead

at class load time and a high overhead to store the compilation results.

For applications that make little use of dynamic loading, ahead-of-time compilation is an ideal means to speed up execution speed. Performance critical code can be compiled when the system is build and highly optimised machine code replaces the interpreted bytecode instructions.

The use of ahead-of-time compilation avoids the overhead and unpredictability caused by compilation on the target system.

However, the compiler must allow the mixing of compiled code and interpreted bytecode to permit the execution of code in classes that are loaded dynamically. This code does not profit from the compilation directly, but it profits from optimised compiled code in standard libraries that are present on the system.

An important aspect of compilation is the code size required for compiled code. Machine code is generally significantly larger than Java bytecode instructions. Furthermore, most applications spend the largest time of the execution in a small fraction of their code. Consequently it makes sense to compile only those parts of an application that have a sufficiently large part of in the overall execution time and leave the rest in smaller (but slower) bytecode instructions.

Tools like a profiler can help to find and compile the most important parts of an application automatically. As an example, we have run the embedded caffeine benchmark with different percentages of its methods compiled. The result is that it is sufficient to compiler 10% of the methods to obtain the same performance as when all methods are compiled. However, the total code size of the application is 100kBytes larger when all code is compiled.

7. CONCLUSION

Even though there are a number of technical challenges that needed to be solved, a Java implementation for embedded realtime systems can provide a solution to make Java deterministic, small and efficient.

When using such an implementation, the developer can profit from the advantages that made Java technology so popular in other application domains.

8. FURTHER INFORMATION

Additional Information is available online:

Technical Papers:

<http://www.aicas.com/publications.html>

Free download of JamaicaVM for Linux:
<http://www.aicas.com/jamaica.html>