Real Java for Real Time -- Gain and Pain

Nilsson, Anders; Ekman, Torbjörn; Nilsson, Klas

Published in:
Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems

DOI:
10.1145/581630.581680

2002

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Users may download and print one copy of any publication from the public portal for the purpose of private study or research.

You may not further distribute the material or use it for any profit-making activity or commercial gain

You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
ABSTRACT
The Java programming language, being a portable and safe object-oriented language, has gained much interest among embedded and real-time systems developers. However, standard Java implementations exhibit problems with performance, memory footprint, and predictability. The question is then, are these limitations inherent in the technology?

Reviewing run-time aspects and the possibility to compile Java to native code, reveals some real limitations as well as common misconceptions. Investigation of the real limitations shows that for implementing real-time Java on small embedded platforms, native compilation via C is an appropriate solution for many platforms and applications.

A revised technique for Java-compatible memory management is proposed to reduce latencies, and linkage of externally generated (C) code with natively compiled Java is considered in a prototype that has been implemented. Based on application demands and experimental verification, we find that real-time Java can, and should, retain the standard simple Java memory model to the programmer.

Categories and Subject Descriptors
J.7 [Computer Applications]: Computers in Other Systems—Real-Time; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms
Languages, Performance, Experimentation

Keywords
real-time Java, embedded systems, rtJ

1. INTRODUCTION
Properties such as portability and safety\(^1\) motivate a great interest in using Java for implementation of embedded software. However, Java has not yet been widely used for programming embedded systems due to a number of reasons. Some of these we suspect are more or less misconceptions and easily solvable, whereas others may be real problems.

When talking about Java, one could mean the language, the virtual machine running byte-codes, or the complete set of standard classes that comprise the Java platform. For small embedded systems there are reasons to believe that the Java language (with a subset of) the standard classes comprises a sufficient basis for developing Java-based systems. In the sequel, we therefore make use of the Java language, possibly with enhanced but Java-compatible semantics.

Real-Time Java Objectives
Considering application and development needs, there exist a number of criteria that all need to be fulfilled for Java to be an appropriate programming language for real-time systems. Such systems range from small embedded devices (such as micro-controllers), and up to large scale systems including both enterprise computing and real-time machine control (such as automation systems). In order for Java to be a widely applicable language in the embedded world, all of the following objectives need to be considered:

\begin{itemize}
\item Memory footprint For most embedded devices, especially mass-produced devices, memory is an expensive resource. A tradeoff has to be made between cost of physical memory and cost savings from application development in higher level languages.
\item Performance CPU performance, and in some cases power consumption, is also an limited resource. The cheapest CPU that will do the job generates the most profit for the manufacturer. The same tradeoff as for memory footprint has to be made.
\item Determinism Many embedded devices have real-time constraints, and for some applications, such as controllers, there might be hard real-time constraints. Computing in Java needs to be as predictable as current industrial
\end{itemize}

\(^1\)By safe language we not only mean that it has strict typing, but also that all possible executions are expressed by the source code. This in contrast with for example C and C++, or C# when the keyword unsafe is used.
practice, that is, as predictive as when programming in C/C++.

Latency For an embedded controller, it might be equally important that the task latency, i.e. the time elapsed between the event that triggers a task for execution and when the task actually produces an output, is sufficiently short and does not vary too much (sampling jitter).

Safety The so called "sandbox model" supported by the safety of the Java language should be maintained. This is particularly important in flexible automation systems where configuration at the user's site is likely to exhibit new (and thereby untested) combinations of objects. Specifically, APIs or language extensions for real-time programming should be Java compatible and without violating safety (by providing raw memory access and the like).

External code The Java application, with its run-time system, does not alone comprise an embedded system. There also have to be hardware drivers, and possibly library functions and/or generated code from high-level tools. Examples of such tools generating C-code are Real-Time Workshop composing Matlab/Simulink blocks [14], generation of real-time code from declarative descriptions such as Modelica [16], object-oriented DAE models, or computations generated from symbolic tools such as Maple [13]. Note that assuming these tools (resembling compilers from high-level descriptions) are correct, programming still fulfills the safety requirement.

Real Java If possible, real-time programming in Java should be supported without extending or changing the Java language or API. For instance, the special and complex memory management introduced in the RTJ proposal [3] need to be abandoned to maintain the superior portability of standard Java, as needed within automation and other fields.

In the sequel, the safety and real Java objectives will be implicitly considered as constraints on any appropriate solution, and referred to as a need for flexibility, whereas the other aspects will be covered in one section each. Our aim is to find a strategy with reasonable trade-offs, making Java suitable for design and implementation of embedded software.

Most important is to unveil misconceptions due to experiences from current implementations, and to find inherent limitations (contradictory objectives, etc.) that cannot be fully avoided. Several systems and publications claim to provide Real-Time Java [3, 4, 23, 6, 20]. In one form or another that can be true, but considering actual application demands as expressed in the objectives above, no real real-time Java platform has been proposed or accomplished so far. We bring the analysis of the situation one step further, which together with fully implemented prototypes forms the basis for the novelty of this paper.

2. MEMORY FOOTPRINT

A standard JVM needs memory in tens of megabytes and is obviously inappropriate for the vast majority of embedded systems. There are a few JVM implementations available that are targeted at embedded systems, see for example the Java 2 Micro Edition (J2ME) [24] from Sun Microsystems Inc. The J2ME incorporates a virtual machine called KVM. The minimum static memory footprint of the KVM is about 40KB and for running applications, the platform should have at least 128KB of memory. It must also be noted that J2ME is not 100% pure Java, but has significant limitations.

Another minimal VM, the JVM [10], is being designed to be even more economical with memory. The interpreter occupies about 30KB and the heap can be in the range of a few tens of KB. The JVM is designed for predictable behavior and run-time flexibility but not performance, and is thus not a generic solution.

One could also consider implementing a JVM directly in hardware, as shown for example by [6] and [1]. However, we would be limited to choosing one of those few platforms, unable to take advantage of the multitude of embedded processors on the market.

But, does Java have to run as byte-codes in a virtual machine? By natively compiling the Java application, the virtual machine can be replaced with a small kernel and garbage collector occupying less than 10KB of ROM and 1KB of RAM, as shown in [16].

3. PERFORMANCE

Interpreting Java byte-codes in a virtual machine typically runs at 10% of the speed of a corresponding C++ program. Many attempts have been made to reduce this speed gap using different just in time (JIT) compilers that analyze the interpreted byte-codes and adaptively compiles them into native code during run-time. The best JIT compilers today generate code that can close to, or in some cases even outperform, C++ code.

As usual, there is a price to pay. Not only will a JIT compiler require a substantial amount of memory which violates our criterion on memory footprint, but it is also difficult to schedule the compiler in such a way that it will not violate the criteria on determinism and latency.

Also here the problem can be reduced by natively compiling Java. In [18], we have shown that performance of our natively compiled Java code is close to that of C++. In a much more comprehensive study, Fitzgerald et al. [5] show similar results.

4. CHOOSING STRATEGY

Taking both the memory footprint and the performance criteria into account suggests that natively compiled Java could be the best tradeoff between flexibility and resource efficiency for small embedded systems.

Compiling Java to native code could be done either directly, or via an intermediate language. Using C as intermediate language has many advantages. Since there are C compilers available for virtually all usable CPUs there is no need for supporting multiple back-ends, and linking the application
to hardware drivers and other external code, see Section 6, becomes easier. However, garbage collection handled in C code (or rather, in the output from a Java to C translator) introduces some interesting problems which are addressed in Sections 5.2 and 7.

Let us take a look at what a Java to C translation could look like.

A Small Example
Consider the Java class in Figure 1, with a method that takes two arguments (one of them a reference), has two local variables, and makes a call to some other method before it returns. Compiling this class into equivalent C code yields

class AClass {
    Object aMethod(int arg1, Object arg2) {
        int locVar1;
        Object locVar2;
        Object locVar3 = new Object();
        locVar2 = arg2.someMethod();
        return locVar2;
    }
}

Figure 1: A small example Java class.

something like what is shown in Figure 2. Note that the referred structures that implement the actual object modeling are left out.

ObjectInstance* AClass::aMethod(JINT arg1, ObjectInstance* arg2) {
    JINT locVar1;
    ObjectInstance* locVar2;
    ObjectInstance* locVar3;
    locVar3 = Object();
    locVar2 = arg2->class->methodTo1.someMethod();
    return locVar2;
}

Figure 2: The method of the small Java example class translated to C, neglecting preemption issues.

The code shown in Figure 2 will execute correctly in a sequential system. However, garbage collection, concurrency and timing considerations will complicate the picture.

5. DETERMINISM
If the embedded system has (hard) real-time constraints we must make sure that both the application and the run-time system are completely deterministic, i.e., there is a guaranteed worst case execution time (WCET) for all tasks with hard deadlines in the application.

In a standard JVM, there are two main features that deteriorates the deterministic behavior: dynamic class loading and garbage collection. Other problem sources, such as allocation of lock objects, priority inversion, class initialization, non-deterministic implementations of type coercion and/or interface invocations have been dealt with, and solved, by many authors.

5.1 Dynamic Class Loading
In traditional Java, every object allocation (and calls to static methods or accesses to static fields) pose a problem concerning determinism, since we can never really know for sure if that specific class has already been loaded, or if it has to be loaded before the allocation (or call) can be performed. Using our solution all referred classes will be loaded before execution starts since they are statically linked with the application. This ensures real-time performance from start.

Application-level class loading does not require real-time loading, but when a class has been fully loaded, it should exhibit real-time behavior just like the statically linked parts of the application. This is related to ordinary dynamic linking, but class loaders provide convenient object-oriented support. That can, however, be provided also when compiling Java to C, using the native class loading proposed by Nilsson et al. [19]. Using that technique, we can let a dedicated low-priority thread take care of the loading and then instantaneously switch to the cross-compiled binaries for the hard real-time parts of the system. Dynamic code replacement can be carried out in other ways too, but the approach we use maintains the type-safety of the language.

5.2 Garbage Collection
The presence, or absence, of automatic garbage collection in hard real-time systems has been debated for some years. Both standards proposals for real-time Java [3, 4] assume that real-time GC is impossible, or at least not possible to implement efficiently. Therefore they propose a mesh of memory types instead, effectively leaving memory management into the hands of the application programmer. Some researchers, on the other hand, work on proving that real-time GC actually is possible to implement.

Henriksson [9] has shown that, given the maximum amount of live memory and the memory allocation rate, it is possible to schedule an incremental compacting GC in such a way that we have a low upper bound on task latency for high priority tasks.

Siehert [21] chooses another strategy and has shown that, given that the heap is partitioned into equally sized memory blocks, it is possible to have an upper bound, though varying depending on the amount of free memory, on high priority task latency using an incremental non-moving GC. The varying task latency related to the amount of free memory m in a way that \( \lim_{m \to 0} l = \infty \) when using Siehert’s dynamic GC scheduling. By performing the static memory analysis of Henriksson we can get rid of this annoying, and also dangerous, variation.

Example with Compacting GC
Using an incremental compacting GC in the run-time system, the C code in Figure 2 will not suffice for two reasons. The GC needs to know the possible root nodes, i.e. references outside the heap (on stacks or in registers) pointing into the heap, for knowing where to start the scanning
phase. Having the GC to find them by itself can be very
time-consuming with a very bad upper bound, so better is
to supply them explicitly. Potential root nodes are refer-
ence arguments to methods and local reference variables.
Secondly, since a compacting GC will move objects in the
heap, object references will change. Better than searching
for them, is to introduce a read barrier (an extra pointer
between the reference and the object) and pay the price of
one extra pointer dereferencing when accessing an object.
The resulting code is shown in Figure 3.

#define REF(x) (x **)          
#define DEREF(x) (* x)          

REF(ObjectInstance) AClass::Object::Method(
    REF(ObjectInstance) this, JINT arg1,
    REF(ObjectInstance) arg2) {         
    JINT locVar1;
    REF(ObjectInstance) locVar2;
    REF(ObjectInstance) locVar3;
    GC_REG(arg2);
    GC_REG(locVar2);
    GC_REG(locVar3);

    locVar3 = Object();
    locVar2 =
        DEREF(arg2)->class->methodTable.someMethod();
    GC_POP(3);
    return locVar2;
}

Figure 3: Handling of compacting GC added to the
small Java example class.

The REF(x) and DEREF(x) macros implement the needed
read barrier while the GC_REG(x) and GC_POP(x) macros
respectively register a possible root with the GC, and pops
the number of roots that was added in this scope.

Example with Non-moving GC
The generated code in Figure 3 is valid also for a non-
moving GC, but since objects are never moved and refer-
ences changed, the read-barrier penalty can be avoided by
redefining the corresponding macros as shown in figure 4

#define REF(x) (x *)          
#define DEREF(x) (x)          

Figure 4: Macro definitions for non-moving GC.

6. EXTERNAL CODE
Every embedded application needs to communicate with
the surrounding environment, via the kernel, hardware de-
vice drivers, and may be with various already written library
functions and/or generated code blocks from high level pro-
gramming tools (such as Matlab/Real-Time Workshop from
The MathWorks Inc.). As mentioned, native compilation
via C simplifies this interfacing. Sharing references between
generated Java code and an external code module, e.g. a
function operating on an array of data, has impact on the
choice of GC type and how it can be scheduled.

When using a compacting GC, one must make sure that the
object in mind is not moved by the GC while referred to
from the external code since that code can not be presumed
to be aware of read barriers. If the execution of the exter-
nal function is sufficiently fast, we may consider it a critical
section and disable preemption during its execution. More
on this topic in Section 7. A seemingly more pleasant al-
ternative would be to mark the object as read-only to the
GC during the operation. Marking read-only blocks for ar-
bitrarily long periods of time would however fragment the
heap and void the deterministic behavior of the GC.

For non-moving GCs, the situation at first looks a lot better
as objects once allocated on the heap never move. How-
ever, as a non-moving GC depends on allocating memory in
blocks of constant size to avoid external memory fragmenta-
tion in order to be deterministic, objects larger than the
given memory block (e.g. arrays) size have to be split out
over two or more memory blocks. Since we can never guar-
ante that these memory blocks are allocated contiguous,
having external functions operate on such objects (or parts
thereof) is impossible.

However, if we do not depend on having really hard timing
guarantees, the situation is no worse (nor better) than with
plain C using malloc() and free(). Memory fragmenta-
tion has been argued by Johnstone et al. [12] not to be a
problem in real applications, given a good allocator mecha-
nism. Using a good allocator and a non-moving GC, the nati-
vely compiled Java code can be linked to virtually any external
code modules. The price to pay is that memory allocations
are no longer strictly deterministic, just like in C/C++.

7. LATENCY AND PREEMPTION
Many real-time systems depend on tasks being able to pre-
empt lower priority tasks to meet their deadlines, e.g. a spo-
radic tasks triggered by an external interrupt needs to sup-
ply an output within a specific period of time. Allowing a
task to be preempted poses some interesting problems when
compiling via C, especially in conjunction with a compact-
ing GC. How can it be ensured that a task is not preempted
while halfway through an object de-referencing, by the GC?
The GC then moves the mentioned object to another loca-
tion, leaving the first task with an erroneous pointer when it
later resumes execution. And what about a "smart" C
compiler that finds the read-barrier superfluous and stores
direct references in cpu registers to promote performance?

Using the volatile keyword in C, which in conjunction with
preemption points would ensure that all root references ex-
ists in memory, is unfortunately not an answer to the lat-
ter question since the C semantics does not enforce its use
but merely recommends that volatile references should be
read from memory before use. Though many C compilers
for embedded systems actually enforce that volatile should
be taken seriously.

Compacting GC
One possible solution is to explicitly state all object refer-
ences as critical sections during which preemption is disa-
allowed, see the example code in Figure 5. This can be a
valid technique if the enabling/disabling of preemption can
be made cheap enough. On the hardware described in Sec-
where preemption is disabled for as short periods of time as possible. If one considers overturning this assumption and instead have preemption generally disabled, except at certain "preemption points" which are sufficiently close to each other in terms of execution time, some of the previous problems can be solved in a nicer way, see Figure 6. The

```c
REF(ObjectInstance) AClass_Object_Method(
    REF(AClassInstance) this, JINT arg1,
    REF(ObjectInstance) arg2) {
    JINT locVar1;
    REF(ObjectInstance) locVar2;
    REF(ObjectInstance) locVar3;
    GC_REG(locVar2);
    GC_REG(locVar3);
    ENABLE_PREEMPT();
    DISABLE_PREEMPT();
    locVar3 = Object();
    ENABLE_PREEMPT();
    DISABLE_PREEMPT();
    locVar2 =
        DEREF(arg2)->class->methodTbl->someMethod();
    ENABLE_PREEMPT();
    DISABLE_PREEMPT();
    GC_POP(3);
    return locVar2;
}
```

**Figure 5:** Regard all object references as critical sections to accomplish preemption points.

Figure 9.2, for example, it only costs one clock cycle. Using this technology, the only possible way to ensure the read barrier will not be optimized away, is not to allow the C compiler to perform "destructive" optimizations. It may seem radical but the penalty for not performing hard optimizations may be acceptable in some cases. As shown by Arnold et al. [2], the performance increase when performing hard optimizations compared to not optimizing at all is in almost all cases less than a factor of 2.

However, there are still many possibilities to optimize the code. The optimizations that have the greatest impact on performance are mostly high-level, operating on source code (or compiler-internal representations of the source code), and are best performed by the Java to C compiler which also can perform object-oriented optimizations. Some examples which have great impact on performance are:

**Class finalization** A class which is not declared final, but have no subclasses in the application is assumed to be final. Method calls do not have to be performed via a virtual methods table, but can be made direct calls.

**Class inlining** Small helper classes, preferably only used by one or a few other classes, can be in-lined in their use classes to reduce reference following. The price is larger objects which may be an issue if a compacting GC is used.

For a more comprehensive listing of object-oriented optimizations, see for example [5]. Both optimizations described above may interfere with dynamic class loading, if used (see Section 5.1), requiring some extra care.

In the last example, Figures 5, we assumed that preemption of a task is generally allowed except at critical regions where preemption is disabled for as short periods of time as possible. If one considers overturning this assumption and instead have preemption generally disabled, except at certain "preemption points" which are sufficiently close to each other in terms of execution time, some of the previous problems can be solved in a nicer way, see Figure 6. The

```c
REF(ObjectInstance) AClass_Object_Method(
    REF(AClassInstance) this, JINT arg1,
    REF(ObjectInstance) arg2) {
    JINT locVar1;
    struct {
        REF(AClassInstance) this;
        REF(ObjectInstance) arg2,
        REF(ObjectInstance) locVar2;
        REF(ObjectInstance) locVar3;
    } refStruct;
    refStruct.locVar3 = Object();
    PREEMPT(&refStruct);
    return refStruct.locVar2;
}
```

**Figure 6:** Using explicit preemption points gives some advantages.

struct refStruct holds all local variables and arguments of a method. To handle scoped variable declarations, the names are suffixed in order to separate variables in different scopes that can share the same name but have different types. Registration of GC roots (with the GC_REG(x,n) macro) is simplified to passing the address of the struct and the number of elements it contains, compared to registering each root individually.

The PREEMPT(x) macro checks with the kernel if a preemption should take place. Such preemption point calls are placed before calls to methods and constructors, and inside long loops (even if the loop does not contain a method call). By passing the struct address, we utilize a property of the C semantics which states that if the address of a variable is passed, not only must the value(s) be written to memory before executing the call, but subsequent reads from the variable must be made from memory. Thus we hinder a C compiler from performing (to us) destructive optimizations.

To prevent excessive penalty from the preemption points, a number of optimizations are possible. After performing some analysis on the Java code, we may find that a number of methods are short and final (in the sense that they make no further method calls), and a preemption point before such method calls may not be needed. Loops where each iteration executes (very) fast, but have a large number of iterations, may be unrolled to lower the preemption point penalty. The penalty we are left with after all optimizations is the cpu performance price we have to pay for
accomplishing hard real-time Java. Depending on the timing demands of an actual application, a suitable Latency ⇔ Average CPU performance tradeoff must be made.

**Non-moving GC**

Since reference consistency is a much smaller problem with non-moving GCs, the situation is simplified. In fact, no visible changes have to be made to the code in Figure 3 for maintaining reference integrity. Most macros will be dummies as for the BEBEP macro in Figure 4, and therefore average performance will be improved. However, when dynamically allocating several object sizes the allocation predictability will be as poor as in C/C++.

8. **GARbage COLLECTOR INTERFACE**

Different types of garbage collectors need different support in the generated code, as is seen in previous sections. A universal garbage collector interface (GCI) [11] is therefore being developed in our group. With a universal interface to the garbage collector, the same generated code can be tested and used with different GC algorithms without regeneration. The small Java example generated against the GCI can be seen in Figure 7. The implementation of the GCI macros allows the code to be executed with both compacting and non-moving GCs, as well as enabling general compiler optimizations.

```java
GC_PUSH_ARG0(result); // register with GC
GC_PUSH_ARG0(locVar2);
GC_PUSH_ARG0(locVar3);
GC_PUSH_ARG0(this); // invoke method
java_lang_Object super=Method(syncBody);
{
  struct java_lang_Object_Class *tmp_method;
  GET_method(tmp_method, arg0, class);
  GC_PUSH_ARGS(...methodTM->someMethod, GC_PASS(arg0));
}
GC_SETBackingField(result, locVar2);
GC_PUSH_ARG0(locVar2);
GC_PUSH_ARG0(locVar3);
GC_PUSH_ARG0(locVar3);
GC_PUSH_ARG0(result);
GC_PUSH_ARG0(locVar2);
GC_PUSH_ARG0(locVar2);
GC_PUSH_ARG0(this);
GC_PUSH_ARG0(arg2);
GC_PUSH_ARG0(java_lang_Object, AClass_method_list_Object);
```

Figure 7: The example method using the generic garbage collector interface example.

The various GC macros ensure the atomicity of all operations that may mutate the reference graph, and code in Figure 7 resembles that in Figure 5, with minimal latency. Using explicit preemption points, as in Figure 6 would result in simplified implementation of many macros, and possibly generation of explicit preemption points in the code where needed.

The sometimes intricate reasons for, and implementation of, each GC macro is outside the scope of this paper, see [11].

9. **IMPLEMENTATION AND EXPERIENCE**

Prototyping for experimental verification of our chosen strategy has been carried out. A prototype for a specific platform basically consists of three parts: a Java to C translator, a (small) run-time system, and a subset of the standard Java classes with enforced semantics to better suit real-time demands. But note, real Java.

9.1 **Java-to-C Translator**

The Java-to-C translator is built on a parser which is generated with JavaCC [15] and generates C code from an Abstract Syntax Tree (AST). After the parser has generated an AST, the translator traverses it in two phases. The translator is built in a very modular fashion using aspect-oriented programming (AOP) techniques and tools [8] in conjunction with tools for incorporating reference attributed grammar (RAG) [7] techniques. RAG techniques are very powerful for passing attributed information up and down an AST. Using AOP makes it possible to build a very flexible modular translator where different functionalities are programmed in different aspects. Creating name and type analysis, calls to the GC, read and write barriers, and so on, as different aspects makes it very easy to add or remove functionality in the translator.

During the first pass, the AST is attributed with name and type analysis so that all uses of variables have a reference to their respective declaration, and all declarations have a list of references to all uses of that declaration.

During the code generation pass, calls to the GC and kernel are inserted at appropriate positions. Write barriers are inserted at writes to reference variables, root references are registered with the GC at the beginning of each new code block and they are popped at the end of the corresponding code block.

9.2 **Run-Time System**

Several real-time run-time systems have been developed over the years. Of specific interest in the context of this paper is the implementation for the smallest type of processor.

The run-time system of this prototype consists of a small real-time kernel tailored for use with Java, and a tightly coupled incremental copying garbage collector.

The run-time system is implemented for an Atmel AVR 8-bit RISC CPU which has been used in experiments. This platform is very typical for tiny embedded systems, with 2A code block is defined as ‹Statements›. As variable declarations appear almost anywhere inside a block in Java, but only at the beginning of a block in ANSI C, new blocks may be inserted in the generated code.
only 128 KB ROM and 64 KB RAM. The processor runs at 4 MHz and approaches 1 MIPS per MHz. The fully preemptive kernel and the garbage collector has a footprint of less than 10 kbytes of ROM and 1 kbyte of RAM. Worst-case execution times of operations in the kernel are summarized in Table 1. See also [18].

<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution time in CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context switch</td>
<td></td>
</tr>
<tr>
<td>due to timer interrupt</td>
<td>(103 + 358 \cdot k)</td>
</tr>
<tr>
<td>Context switch</td>
<td></td>
</tr>
<tr>
<td>due to voluntary suspension</td>
<td>(749 + 7 \cdot r + 54 \cdot n)</td>
</tr>
<tr>
<td>Take a mutex</td>
<td>113</td>
</tr>
<tr>
<td>Give a mutex</td>
<td>(1024 + 12 \cdot k)</td>
</tr>
<tr>
<td>Create an object</td>
<td>(2594 + 18 \cdot s + 60 \cdot n)</td>
</tr>
<tr>
<td>Push a root</td>
<td>24</td>
</tr>
<tr>
<td>Pop j roots</td>
<td>12</td>
</tr>
<tr>
<td>Reference new</td>
<td>8</td>
</tr>
<tr>
<td>Read barrier</td>
<td>4</td>
</tr>
<tr>
<td>Write barrier</td>
<td>0.1</td>
</tr>
<tr>
<td>Process a root</td>
<td>5.6</td>
</tr>
<tr>
<td>Mark an object</td>
<td>(110 + 18 \cdot s + 118 \cdot n)</td>
</tr>
<tr>
<td>Sweep an object</td>
<td>(109 + 9 \cdot s)</td>
</tr>
</tbody>
</table>

Table 1: Measured performance with \(k\) priority levels and object size \(s\) bytes with \(n\) pointers divided into \(i\) groups. 1 CPU cycle is 0.25 \(\mu s\).

9.3 Class Library

On of the key factors for Java to be a good programming language also for embedded real-time systems is that the API to the standard classes should not be changed unless absolutely necessary. Retaining the syntactic API (or rather, a subset of it) makes it easier to develop and demonstrate applications in simulated environments, see Figure 8, and to deploy control classes on higher (non-real-time) levels of control running standard Java. Judging from industrial experience, the possibility of early software testing will significantly improve both software quality and time-to-market of the complete product since embedded hardware platforms are both notoriously hard to debug, and are sometimes not available until very late in the development process. At that time much of the software is already written but not tested.

The standard class library used in experiments consists of a subset of the Sun JDK 1.3.

The standard Java threads are known for having too weak semantics for being appropriate in embedded systems. There is also no dedicated periodic threads in java.lang.Thread. The thread package used in our experiments has been developed in an earlier project to better support real-time demands with enhanced semantics and extended classes for periodic and sporadic threads. In a JVM simulation environment, the threads are based upon java.lang.Thread while in the target environment, they retain their API's but are mapped (by our translator) on kernel threads in the run-time system. Thread synchronization is handled in a similar way.

The enforced semantics in the experimental thread and synchronization primitives coincide well with that of two real-time Java proposals, except for the memory management.

10. RELATED WORK

There are two proposals for a real-time Java standard published; one by the Real-Time Expert Group [3] and the other by the J Consortium [4]. They have in common the belief that there is no such thing as real-time garbage collection, and to avoid the non-determinism of normal GCs they propose extended memory models where some areas are manually managed. This approach does solve the GC problem, but leaves the memory management in the hands of the application developer. On the contrary, we believe that one of the main benefits with Java is the automatic memory management, which is very important in flexible real-time (such as automation) systems. We also believe in being able to use the same (source) classes for both real-time and non-real-time applications.

Aicas GmbH and IPD Universität Karlsruhe have implemented a combined JVM and Java bytecode-to-native compiler called Jamaica that is said to comply with hard real-time constraints, see [22, 21, 23]. The Jamaica VM is always responsible for garbage collection and the task scheduling, while some classes may be natively compiled and call the VM for services such as memory allocation. The GC principle used is a non-moving type with fixed memory block size for eliminating external fragmentation. The amount of GC work to do at each object allocation is scheduled dynamically with respect to the current amount of free memory, and task latency also for high priority tasks) will vary accordingly. Not only the varying task latency, but the need for a "booted" VM and the fact that the fixed-size memory block scheme makes linking with non-GC-aware code modules complicated, make the Jamaica system inappropriate for small embedded systems and for flexible hard real-time systems.

![Figure 8: The same application classes are used in a simulation environment with a standard JVM as in the natively compiled target environment.](image)

The memory footprint of the Jamaica VM is around 120 KB.
The PERC Java platform from Newmonics Inc. [17] is another example of a hybrid platform with alleged hard real-time capabilities, but it has a footprint of at least 256KB ROM and 64 KB RAM. Linking in external code is also not feasible.

11. CONCLUSIONS

In this paper we have discussed fictive and real limitations of real-time Java. Some of the presumed limitations, memory footprint and performance, can be overcome by natively compiling Java applications. The other discussed limitations; determinism, external code and latency, are partly contradictory.

By using a non-moving garbage collector together with a good allocator mechanism, we are left with the same restrictions for real-time Java as for (correctly programmed) C. That is, only operations that are not strictly deterministic is memory allocation, which is a situation that is good enough for most real-time systems. This configuration also allows linking the Java system with external code (either legacy or generated from high-level tools) without further jeopardizing the deterministic behavior.

If we set the aim even higher, to have strict timing guarantees for all operations, we have to do a tradeoff between two approaches. An incremental compacting garbage collector may be used, with the penalty of moving objects and the need for preemption points. The other viable alternative is to use an incremental non-moving garbage collector with fixed size memory blocks, where the penalty comes in the form of large objects (e.g. arrays) being split into several memory blocks.

Both alternatives are not usable with external code linking in the general case, for different reasons. In the compacting GC case, a call to an external function is a critical section for which the worst case execution time may inflict missed deadlines in high priority threads. In the non-moving GC case, the splitting of large objects renders external functions operating on such objects, i.e. arrays, useless.

Our conclusion is that the desired implementation strategy is to develop applications in 100% pure standard Java, following some programming conventions, with an adequate run-time system, but without external C/C++ code except for static parts such as device drivers.

Considering the objective to be able to use external code, the situation is more complex. For systems with really strict timing guarantees a tradeoff has to be made between determinism and the ability to link in externally generated code. Considering current industrial practices and required system properties (as expressed in the objectives listed) we think programming in standard Java (with enhanced semantics) in combination with the proposed support for selecting type of GC (e.g. non-moving for external code) permits the developer to accomplish the most appropriate properties (trading average performance for strict timing guarantees) of the system or product.

12. ACKNOWLEDGMENTS

This work has been carried out within the LUCAS4 applied software research center with financing from the Swedish Agency for Innovation Systems (Vinnova) and the Swedish Foundation for Strategic Research (SSF). We would like to thank Anders Blomdell at the Department of Automatic Control, for sharing his deep knowledge in compilers and real-time kernels, and many of our colleagues at the Department of Computer Science.

13. REFERENCES


http://www.lucas-lab.se