Deterministic Java in Tiny Embedded Systems

Anders Nilsson
Torbjörn Ekman
Department of Computer Science
Lund University, Sweden
{andersn | torbjorn}@cs.lth.se

Abstract

As embedded systems become more and more complex, and the time to market becomes shorter, there is a need in the embedded systems community to find better programming languages that let the programmers develop correct code faster. The programming languages used today—typically C and/or Assemblers—are just too error-prone. The Java technology has therefore gained a lot of interest from developers of embedded systems in the last few years.

We propose an approach based on compiling Java into native machine code via C as an intermediate language. The C code generation process should also add close interaction with a fully pre-emptive incremental garbage collector and a small and efficient real-time kernel. Tests performed on a small 8-bit microprocessor show that it is possible to use a modern object-oriented language with automatic memory management—such as Java—and yet generate fully predictable code that can be run in very small devices with severe memory constraints.

1. Introduction

Embedded systems are traditionally programmed in C or Assembler, but as the systems grow more complex and the time to market decrease, the need for a more secure and structured language has increased.

In the last few years, the programming language Java [11] has gained more and more interest among embedded systems developers. The object-orientation, strict type checking, and automatic memory management, are features that make it easier to write large and complex systems with less risk of introducing difficult bugs. It's C-like syntax also appeals to C programmers as it makes it easier for them learn, as well as makes it easier to port legacy C code to Java. In the area of automatic control, where domain-specific languages and run-time systems [9, 1] are preferably used, our work supports more robust porting of such systems to even small embedded devices.

However, some serious problems arise when one wants to use Java in small embedded systems, where the most significant ones have to do with the inevitable speed- and memory constraints imposed by using a Java Virtual Machine (JVM). If hard real-time demands are to be fulfilled, there are also problems with most garbage collectors not being predictable with regard to non-preemptable execution time. This can result in too much jitter in the accomplished sampling interval of high priority threads, or even missed samples or deadlines.

1.1. Problem area

Our work is directed towards a certain class of applications and systems where a traditional Java environment cannot fulfill our application demands. Examples of such applications can be found in tiny embedded control devices used in industrial processes. The demands on that kind of embedded system are:

Correctness: The system must not crash due to some bug causing a memory leak or pointer arithmetic failure, for instance once every three months. Of course a programming error can result in too much memory being consumed, but in such a case controlled error handling (exceptions) should enable graceful degradation.

Hard Real-Time: The physical process may be very sensitive to jitter in the sampling interval of the controller. Too much jitter may cause the process to perform badly, or even possibly become unstable[2].

Speed: Applications, such as a servo control, may need a sampling interval down to a few milliseconds to perform well.

Cost: To be able to sell such systems, in some cases, we cannot use anything more powerful than, say, a simple 8 bit micro-controller with less than 128KB of RAM.

The **Correctness** demand indicates that we should use a modern language with strong typing and automatic memory management, and only use C and assembler for isolated functions and hardware interfaces. Of course an application needs to be correctly written and tested in order to really be correct. Using a safe[1] language drastically improves the development process. The **Hard real-time** demand requires predictability in both run-time and garbage collector. The **Speed** and **Cost** demands indicates that we must look for a reasonably effective solution so that cheap micro-controllers can be used.

The correctness demand and the other three demands have so far been contradictory. The topic of this paper is to resolve that contradiction! On one hand, we want to use Java, representing a safe object-oriented programming language with many other nice features. On the other hand, Java in it's traditional—interpreted byte code—form is neither predictable regarding timing nor can it be run on small micro-controllers with very limited amounts of RAM.

We are also committed to not making any extensions to the Java language as that would make it more difficult to simulate the software with standard Java tools on a standard workstation where all sorts of debugging tools are available.

From a pessimistic point of view, the Java language does not support predictability and real-time programming, and Java VMs and standard class libraries does not fit into very small systems. From an optimistic point of view, however, the Java language supports concurrency and it does not explicitly hinder utilization of real-time support from the underlying system, and for development of embedded systems we are free to use an appropriate (possibly our own) runtime system. The following is based on the optimistic approach.

1.2. Approach

We propose an approach consisting of three parts working together:

Compiled Java: By compiling Java—via C as an intermediate language—we should be able to make typical applications sufficiently fast and memory effective.

Real-Time Kernel: A small real-time kernel which is tailored for small micro-controllers and object-oriented applications.

Predictable Garbage Collector: A predictable garbage collector, which is integrated with the kernel, helps us to fulfill also the hard real-time demand.

[1]By a *safe language* we mean a language that ensures that all possible executions are expressed by the program itself. Specifically C and C++ are unsafe, whereas Java is safe. C# is safe except where declared unsafe.

2. Compiling Java

To be able to meet the demands on speed and memory consumption[2], we need to compile our Java code into processor specific machine code. There are basically two ways to do this:

Native compiler: An ordinary compiler taking Java source code, or byte code, and producing machine dependent binary code.

Intermediate language: A tool for converting Java source or byte code to some intermediate language. The intermediate representation can then be compiled with a standard compiler for the specific machine architecture.

2.1. Native Compiler

Using a native Java compiler seems at a first sight to be the most straight-forward way to produce machine code from Java source. There are some native compilers available, both as commercial packages and as open source. Most of those are aimed at speeding up execution of large Java applications, especially on the server side in a client-server solution, see for example TowerJ3[25] or Jove[13]. WindRiver Inc. has developed a native Java compiler, TurboJ[28], that produces object files which can be linked to their real-time operating system VxWorks. They have however not dealt with the predictability problem incurred by the automatic memory management, and thus recommend that all critical real-time parts of an application should be written in C/C++ as usual.

2.2. Intermediate language

There are some tools available today which can translate Java to an intermediate language, usually C. Most of them, for example Toba[20] and Harissa[17] take Java byte code and converts it to C source code. The other type of converter—going from Java source to C source—is represented by jcc[23]. There are good things and bad things in both approaches. Converting byte code makes it easier to use pre-compiled Java libraries or applications which may not be available as source. On the other hand, the byte code is tailored for a generic stack machine with no hardware registers, which we think will harm the performance compared to generating C code from Java source, at least if no special optimization techniques are used. Converting Java source code also produces a somewhat more readable

[2]The Java byte codes actually occupies less memory than machine codes, but a JVM will take some memory space. Both in ROM for itself and some extra RAM for it's runtime. A JVM implemented in hardware would need significantly less ROM, but that solution has other drawbacks.

C code, which makes debugging feasible also for the intermediate code.

The biggest drawback of using Java source as input to the C code converter is that many of the available Java packages are only available as pre-compiled byte code. This introduces the limitation that we can only compile Java programs that are —in itself and for all dependencies—available as source. However, preliminary tests with Java decompilers, such as jad [15], shows that the byte code can quite well be decompiled into Java source, then making our approach feasible.

2.3. Our Compiler

For portability- and efficiency reasons we are focusing on going via C as an intermediate language for compiled Java, rather than implementing a compiler or adapt an existing compiler, for example GCC[7], to meet our needs. A tool that converts Java source to C can also generate object layout information and calls that makes predictable garbage collection possible.

A tool called Java2C has been developed. Given Java source as input, it generates the corresponding C code as well as the necessary GC administrative calls and information about the layout and size of objects.

The compiler: The Java2C tool is built in 100% pure Java2, so it can be used on any platform that can run a Java2 virtual machine. It also implies that the Java2C tool could convert itself for native compilation and better performance. A parser generator is used to build a parser for the Java formal grammar. The parser can then take any valid Java source code file and produce an abstract syntax tree (AST). From this AST, the C source code is then generated in one pair of files for each Java class or interface (one .h and one .c file).

The compiler-compiler: The JavaCC compiler-compiler [16] is a freely available parser generator written in Java. It was originally written by Sun Microsystems but is now freely available from Metamata Inc. Given a grammar in a BNF-like form, it builds a parser in Java and optionally also an AST from the parsed file. Each node of this AST consists of a Java class all inheriting a common ancestor `Simplenode`. This makes it fairly easy to traverse the syntax tree by using method overloading.

Code generation The generation of C code is accomplished by traversing the AST of a class in two passes. During the first pass information about inheritance, field- and method declarations is gathered whereas the actual code generation is accomplished during the second pass.

2.4. Object model

Some works has been carried out at the department on how to model compiled real-time Java [4]. An instance of any object is represented by a pointer to an object instance structure, see Figure 1.

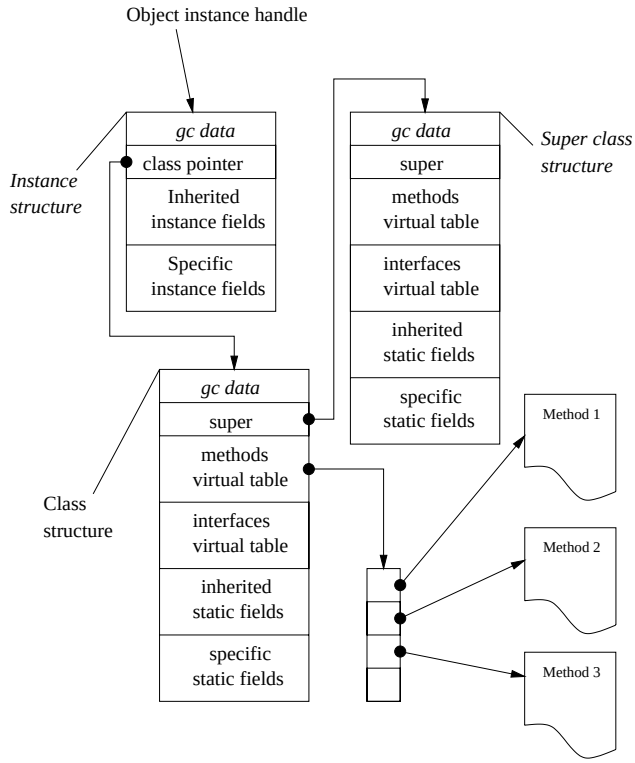

Figure 1. Run-time model of an object.

Consider a very simple Java class with one constructor and one method like:

```
class Dummy {
  int a,b;
  String name;

  Dummy() {
    a = 1;
    b = 2;
    name = new String("dummy");
  }

  String getName() {
    return name;
  }
}
```

This results in the following C code:

```
struct DummyClassStruct {

      // GC data

      // Super class ptr
   struct ObjectClassStruct *super;

      //methods virtual table ptr
   void* (**methodTblPtr)();

      //interfaces virtual table ptr
   void* (**interfaceTblPtr)();

};
typedef struct DummyClassStruct DummyClass;

struct DummyInstanceStruct {

      // GC data

      // Class ptr
   struct DummyClassStruct  *classPtr;

      // Primitive type fields
   int a;
   int b;

      // Field of type String
   REF(struct StringInstanceStruct) name;
};
typedef struct DummyInstanceStruct
                    DummyInstance;
```

The `REF(a)` macro is explained in section 5.

3. Real-Time Kernel

The kernel made is a preemptive multi-threaded kernel with a fixed priority based scheduler. Effort has been made in creating predictable lower and upper bounds on each function in the kernel. Worst case execution times of operations affecting context switch, interrupts, and initialization of threads are made to be affected by the number of priority levels and not the number of currently running threads to lower the jitter. Much of the kernel properties are standard, but the structure of the queues, with respect to priorities and execution time, may not be new but we have not seen it elsewhere.

3.1. Thread model

The kernel supports two different types of threads:

Ongoing threads Each ongoing thread has its own stack space and will upon completion be removed from the system. The thread will execute for a small period of time and then be preempted by another thread. These threads have no limitations on which kernel primitives to use, or how common resources are shared.

Periodic threads These threads have a fixed period and may not be preempted by threads that have the same priority, and hence they may not share synchronization primitives. These limitations makes it possible for all periodic threads of the same priority to share a common stack which improves memory consumption.

These types of threads, but not their implementations, are related to [4] which was inspired by [18].

3.2. Priority Queues

When a thread is in a suspended or in a ready state, it is placed in a queue. We used the data structure depicted in Figure 2. The *next pointer* references the next element in the queue. The *last pointer* references the last thread in a group with threads of the same priority. Within each priority level we use the last-in-first-out strategy. To insert an element we only have to go through all priority levels in the worst case, regardless how many threads there are in the queue. We can dequeue and enqueue all threads with the same priority in constant time.

3.3. Scheduling

A thread is assigned a time-slice to execute by the kernel, and control is transferred to that thread during a context switch. The thread can be preempted either by using a function in the kernel or when it has used its time slice. The scheduler is then invoked and chooses the next thread to execute.

The scheduler uses as many ready queues as there are priority-levels in the kernel. Each queue is of FIFO type. When a thread is preempted, after it has used its time slice, it is inserted last in the queue of its priority class. The scheduler then chooses the next thread to execute, by selecting the first element in the queue of highest priority. If the queue is empty, the queue of second highest priority is used and so forth.

3.4. Time

The kernel is interrupted at a given interval and at this time the tick counter is updated. This period is also used as a timeslice, so the kernel preempts the current running thread and reschedules at this time too. There are three primitives in the kernel for a user program to handle time. The tick counter can be read as well as a fine-grained timer. The kernel also provides a primitive for suspending a thread until a certain time.

When the tick counter is increased the kernel also moves any threads waiting for that tick from their suspended state to ready state. A time queue that consists of several queues,

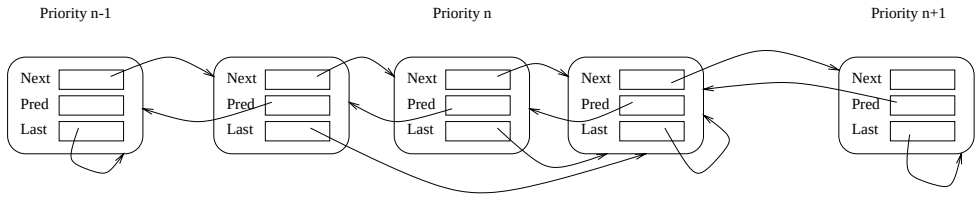

Figure 2. Queue data structure

like the one in Figure 2, are chained together to a long queue. Each subqueue is sorted by priority and the subqueues are chained together in a low-to-hi time order. This way threads of the same priority class that are waiting for the same tick can be handled as a unit. We have balanced the routine that moves threads from the time queue into the different ready queues to always take the worst case execution time. This is done to lower the jitter.

3.5. Synchronization

Synchronization is supported in the kernel by semaphore primitives. Other primitives can in turn be built from semaphores. There exist both binary and counting semaphores. Trying to take a semaphore which counter is zero suspends the current running thread and the kernel reschedules.

To solve the well known problem with priority inversion when using semaphores for mutual exclusion, a mutex form of semaphore is supplied. The most commonly used priority inheritance protocol to defeat priority inversion is the basic priority inheritance protocol. This protocol does, however, permit a thread to be blocked by several lower prioritized threads. Both priority ceiling and immediate inheritance protocol permit only one lower prioritized thread to block another thread and also prohibit deadlock [21]. The mutex primitive uses the immediate inheritance protocol as it is cheaper to implement and has the same worst case behavior as the more elegant priority ceiling protocol.

3.6. Interrupts

To each hardware interrupt a number of threads with different priorities can be attached. When an interrupt is triggered the kernel is invoked and the waiting threads are moved from an interrupt queue to different ready queues based on priority. The same type of queue as in Section 3.2 is used for interrupt queues. The kernel then reschedules, and the selected thread's context is restored.

4. Garbage Collector

The garbage collector (GC) in the run-time system needs to have short predictable worst-case execution times at each invocation to fulfill hard real-time requirements. A new fine-grained incremental mark-compact algorithm, which ensures no fragmentation as well as bounded worst case execution times of all operations, is proposed. An introduction to incremental mark-compact algorithms and GC in general is available in [27]. This GC is described in detail in [10].

4.1. Introduction to the GC

References to the heap stored in processor registers, on the program stack, or in global variables are called *roots*. All objects that are reachable directly through the roots or through a chain of pointers from the roots are considered to be live objects. The GC will trace the reachable objects and mark them. The rest of the objects on the heap are garbage and can be reclaimed.

Work done by the GC is interleaved with normal execution of the user program, often called the *mutator* [26]. We will use the tri-color abstraction introduced in [8] to describe synchronization between the mutator and collector. Each object on the heap is painted in one of three colors:

Black indicates that the object and its immediate descendants have been visited.

Grey indicates that the object must be visited by the collector. Either grey objects have been visited by the collector but not all pointers are scanned, or their connectivity to the rest of the graph has been changed.

White objects are unvisited and at the end of the marking phase considered being garbage.

To make sure that the collector has a coherent view of the heap during the marking phase the following invariant must hold:

No pointer in a black object references a white object.

Coherence is maintained by barriers between the mutator and the heap. These barriers can be either a *read-barrier*, trapping reads, or a *write-barrier*, trapping writes.

4.2. The marking phase

During the marking phase we start by coloring all objects referenced by the root pointers grey. We have a stack of root pointers for each thread which is processed root by root. Each object referenced by a root is inserted into a marking list. The elements in the marking list are the grey objects. We then visit all grey objects and mark the objects referenced by them grey. All pointers in the object are processed and then the visited object is colored black. The procedure is repeated for all grey objects. This way all objects reachable from the roots are visited. In the end of the marking phase all objects are either black or white. As we don't want to risk overflowing the stack by recursively traversing the graph we use Cheney's, [6], non recursive marking strategy with a marking list where the reference to the next object to mark is placed in each object.

If the mutator, during the marking phase, tries to set a pointer in a black object to reference a white object we immediately color the referenced object grey to make the invariant hold. To ensure that this is done a *write barrier* is used that colors the white object grey by inserting it last in the marking list.

4.3. The sweeping phase

During the sweeping phase all black objects are moved into one continuous block at the top of the heap. A *scan pointer* is used, which is initially set to reference the last allocated memory location, from the marking phase of the previous cycle. All objects are processed in a top-down order by decreasing the *scan pointer* with the size of the currently scanned object. Black objects are moved to the top of the heap, and white objects are reclaimed. To be able to traverse the heap in either direction the object size is placed in both the beginning and the end of the objects. When the *scan pointer* reaches the bottom of the heap all objects are processed, the current cycle completed, and the next initiated. In the next cycle the heap is traversed in the opposite direction.

4.4. Allocating new objects

New objects are allocated at the top of the heap. At the end of the current cycle, these objects and the object moved during the sweeping phase, are placed in one continuous block at the top of the heap. These objects will be colored and marked during the following collection cycle. When an object is allocated we need to initialize all pointers in the object to reference *null* through a table. The use of a table is discussed in Section 4.5.

4.5. Moving objects

When an object is moved to a new location we must ensure that all references to that object are changed to reference the new location. Moving the object and changing the references must be done as an atomic operation, to make sure that all accesses to an object are made to the correct copy of the object. As the number of references to an object is not known, we would not get a tight upper bound for worst-case execution time of that operation. To get an upper bound on moving an object we access referenced objects through a table. All pointers reference the target object indirectly through the table. This way only the reference in the table needs to be changed. This is the *read barrier*.

As the entire object needs to be moved all at once, the kernel may be suspended for a too long time to meet hard real-time requirements. We allow preemption during copying of an object, and if we are preempted we restart copying the object when we resume. This is to ensure that the copy of the object contains the most recent data.

4.6. Scheduling of GC work

To be sure that the collector will reclaim garbage at a rate necessary for the mutator never to run out of memory, we a priori calculate a minimum collection rate. As long as the current collection rate is above the a priori calculated worst case, we are ensured never to run out of memory. The operation that can exhaust the heap is memory allocation, and therefore we perform an increment of GC at each allocation. We can in this way make sure that there is space on the heap for the new object. We also want to make sure not to do more collection than necessary to interfere as little as possible with the mutator.

To simplify the discussion we assume that all objects are of the same size. This can quite easily be extended to different sized objects as in our actual implementation. Throughout this discussion we will use the following notation:

S	total number of objects that the heap can hold
W	the amount of GC work done so far during this cycle
W_{max}	the amount of GC work necessary during one cycle in the worst case
E_{max}	maximum number of live objects
A_{max}	maximum number of new objects allocated during one cycle
H_{max}	maximum number of new objects allocated by high-priority threads
A	number of objects allocated so far during this cycle
R_{max}	maximum number of root pointers

We define the minimum GC rate, GCR_{min} as

$$GCR_{min} = \frac{W_{max}}{A_{max}}$$

and current garbage collection rate, GCR as

$$GCR = \frac{W}{A}$$

As long as $GCR > GCR_{min}$ we are ensured not to run out of memory. The work that has to be done during one cycle is divided in three parts: processing the roots, marking all live objects, evacuating the marked objects. The maximum work that has to be done can be written:

$$W_{max} = \alpha \cdot R_{max} + \beta \cdot E_{max} + E_{max}$$

where the coefficients α and β compensate for different costs in processing a root or marking an object compared to evacuating an object. For a given hardware, α and β are constant.

The work W_{max} has to be done during one GC cycle, and how long this cycle is depends on how much memory we have. At each allocation we let the collector perform an increment of GC work. We will start by finding out how long a cycle is. When finishing a collection cycle the maximum number of objects on the heap is the number of live objects from last cycle, E_{max}, plus the maximum number of new objects allocated during that cycle, A_{max}. During the following cycle the mutator may allocate as many as A_{max} new objects. The maximum total number of objects on the heap is therefore $E_{max} + 2 \cdot A_{max}$. As we know how big the heap is, S, we can easily calculate how many allocations we can do in one cycle, without exhausting the heap.

$$A_{max} = \frac{S - E_{max}}{2}$$

We now have an expression for the minimum GC ratio necessary, GCR_{min}:

$$GCR_{min} = 2 \cdot \frac{\alpha \cdot R_{max} + \beta \cdot E_{max} + E_{max}}{S - E_{max}}$$

The current GC ratio, GCR, can be expressed as,

$$GCR = 2 \cdot \frac{\alpha \cdot i + \beta \cdot j + k}{S - E_{max}}$$

where i is the number of processed roots, j the number of marked objects, and k the number of evacuated objects. As long as $GCR \geq GCR_{min}$ we are ensured not to run out of memory.

If we divide the maximum total work that has to be done, W_{max}, by the number of allocations we will do during a cycle, A_{max}, we know how much work that will be done at each allocation. The worst execution time for this work can be calculated and added to the cost for allocating one object.

Even if the amount of work that has to be done during an allocation is small and bounded it can still be too long for us to meet all deadlines. To improve the real-time capabilities of the collector we use the technique proposed by Henriksson, [12], and create a semi-concurrent GC. The threads are divided into groups of high- and low-priority threads. GC work is done interleaved with allocation for the low-priority threads. To improve response time for the high-priority threads we suspend the collector until after the high prioritized threads have executed. We can view the collector as a mid priority thread, that gets to execute after the high prioritized threads. We need to make sure that there is enough free space on the heap for the high-priority threads to allocate new objects without exhausting the heap. We denote the maximum number of objects allocated by high priority threads, assuming they are all released at the same time, H_{max}. If we always have this much space free on the heap we are ensured not to exhaust the heap, even if we don't perform any collection work during high-priority thread allocations. The maximum total number of objects on the heap is now $E_{max} + 2 \cdot A_{max} + H_{max}$. The new minimum GC ratio is:

$$GCR_{min} = 2 \cdot \frac{\alpha \cdot R_{max} + \beta \cdot E_{max} + E_{max}}{S - E_{max} - H_{max}}$$

and the the current GC ratio is changed accordingly.

5. Integration

So far we have designed the kernel, the compiler and the GC. But to make predictable garbage collection possible, we need a tight coupling between the compiled application and the garbage collector.

When using a preemptive mark-compact garbage collector, great care must be taken when handling object references as the currently running thread could be preempted at virtually any time. We must assert that there are no dereferenced object handles whenever the GC starts moving objects around. To fulfill these demands, we must consider all reference manipulations as atomic actions.

We must also inform the GC when new roots of object trees are created, and when they are dismissed. This happens whenever a method is called. Consider the example code below:

```
class Dummy {
  public void aMethod(String s) {
    String aString;
        ...
  }
}
```

To register and unregister these two objects as roots in the GC, we introduce the calls `GC_PUSH(ref)` and `GC_POP(nbr)` which, respectively, pushes a reference *ref* onto the GC stack and pops *nbr* references from the stack. The code example above then results in the following generated code:

```
void Dummy_aMethod(REF(DummyInstance) this,
                   REF(StringInstance) s){
    REF(StringInstance) aString;
    GC_PUSH(this);
    GC_PUSH(s);
    GC_PUSH(aString);
        ...
    GC_POP(3);
}
```

To implement the read-barrier we use a macro `REF(a)`, and for the write-barrier we use a macro `GC_SET(a,b)`.

6. Experimental verification and experiences

We have run two types of tests to verify that our solution is feasible. First, we have measured the predictable timing for a multi-threaded application on a small microprocessor typically found in small embedded systems. Second, we have run some benchmarks on a normal desktop workstation to measure the efficiency of our generated code compared to Java and C++.

6.1. Predictable timing

To verify our techniques, we have implemented a prototype for a very limited target platform; the 8-bit Atmel AVR RISC microcontroller [3]. This platform is very typical for tiny embedded systems, with only 128 KB ROM and 64 KB RAM. The microcontroller used runs at 4 MHz and approaches 1 MIPS per MHz. The kernel and garbage collector allocates 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. If we can predict worst-case execution times and worst-case memory demands of the different threads, in combination with execution times of the kernel operations, we can use generalized scheduling theory [22], to check if the system is schedulable or not.

	Execution time in CPU cycles	
Operation	*Worst*	*Best*
Context switch due to timer interrupt	$963 + 358 \cdot k$	$889 + 346 \cdot k$
Context switch due to voluntary suspension	$740 + 12 \cdot k$	728
Take a mutex	113	113
Give a mutex	$1024 + 12 \cdot k$	1014
Create an object	$234+78 \cdot i+54 \cdot n$	$234+78 \cdot i+54 \cdot n$
Push a root	24	24
Pop j roots	12	12
Reference $null$	8	8
Read barrier	4	4
Write barrier	51	5
Process a root	83	64
Mark an object	$110+78 \cdot i+118 \cdot n$	$110+78 \cdot i+60 \cdot n$
Sweep an object	$169 + 9 \cdot s$	104

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 μs.

6.2. Comparison benchmarks

In order to get some kind of performance measurements of the translated Java code, we have run some benchmark tests on Java2C generated code as well as on a JVM and compiled C++. The benchmarks presented are limited to speed measurements on object allocation and methods calling. So called "number crunching" is not that interesting as Java2C translated arithmetic expressions code will perform very close to the speed of hand-written C-code.

All execution time measurements were performed on a Sun Ultra 5 workstation with 256MB of RAM memory running Solaris 7. The Java compiler used was jikes from IBM and the virtual Java machine was Java HotSpot version 1.3.0. G++ and GCC were of version 2.91.66.

The code generated with Java2C does not contain a garbage collector. This because the GC developed is too tightly coupled to the ARM microprocessor to be easily ported to the SPARC. Adding our predictable garbage collector would give some penalty on the measured execution times, but the magnitude would not change.

The benchmarks were as follows:

Allocation. Allocate 2500 objects. The constructor of each objects also allocates a byte array of size $0 - 5000$.

Virtual methods. Make 1000000 calls to a small virtual method.

Final methods. Make 1000000 calls to a small final method.

Final methods. Make 1000000 calls to a small static method.

The measured execution times are shown in Table 3.

From the results one can see that the performance of the code from our translator is almost as fast as natively compiled C++ code, especially if we could allow some compiler optimizations.

Compiler/Run-time	*Execution time (ms)*			
	Allocation	*Virtual methods*	*Final methods*	*Static methods*
java -Xint	350	13000	13450	11980
java	360	1180	1080	670
g++	90	1720	960	1340
g++ -O2	90	970	70	310
Java2C / gcc	100	2280	2260	1380
Java2C / gcc -O2	100	1040	1000	140

Table 2. Measured execution times for some benchmarks.

What may seem astonishing is that the Java HotSpot performs so well performing method calls. The explanation is that because it is the same method that is called 1000000 times, HotSpot will soon perform JIT compilation.

7. Problems and Future Work

Problems concerning C as the intermediate language is not so much about the C language— which is very allowing, to say the least—but how the C compilers generate machine code. In the current implementation, the mandatory atomicity of object reference manipulations is obtained by using pre-emption points in the code, and by turning off all compiler optimizations. A very interesting problem is to be able to utilize compiler optimization techniques , but that is outside the scope of this paper.

Another problem is to calculate a good upper bound on the maximum number of live objects in the system. A guaranteed upper bound tends to be very pessimistic and will degrade the performance of the system. There is, however, recent work done [19] which provides a much better estimate.

There is still some work to be done in the Java to C translator. Some of the more important features of the Java language that is being implemented are interfaces and exceptions. The implemented object model is also, at the time of writing, a somewhat simplified version of the one depicted in Figure 2.4.

8. Related Work

There has been quite some work done on natively compiling Java, but not much on hard real-time Java for small systems. Sun Microsystems Inc. has published a white paper[24] on using the Java 2 Platform Micro Edition (J2ME) for mobile devices. The J2ME is centered around a small JVM called KVM and aimed at devices with a total memory amount in the range of 128 - 512 KB. A J2ME application can also be compiled to native code and linked to the KVM for better performance. J2ME is, however, not yet suited for use in systems with hard real-time demands.

Various issues concerning real-time behavior in Java are dealt with in *The Real-Time Specification for Java*[5]. There are significant drawbacks in this specification from our point of view, specifically concerning memory management. Instead of adopting a predictable run-time system, it extends Java with a new memory organization. In addition to the normal *HeapMemory*, it adds *ImmortalMemory*, *ImmortalPhysicalMemory* and *ScopedMemory* memory areas which are all treated differently by the automatic memory management system. These additions place responsibility on the programmer to always do the right thing, since a wrongly placed memory allocation type in an application could totally void the real-time behavior of that application.

There has also been some work done on implementing very small and memory efficient Java virtual machines which can be deployed in systems with hard real-time demands [14] but that requires more time and memory.

9. Conclusions

We have shown that it is possible to use Java as a programming language for developing small embedded systems with very limited resources of CPU power and memory. Given a few assumptions on the memory usage of an application, we can also show that hard real-time timing demands are met.

By choosing C as an intermediate language—and choosing a suitable object representation model—we can achieve the efficiency needed for running applications on very small CPUs while still maintaining some platform independency.

By combining natively compiled Java with a very small and efficient RT kernel and a pre-emtive garbage collector we can write and test multi-threaded programs in a normal Java runtime environment which can later be compiled for small hard real-time systems.

10. Acknowledgments

VINNOVA (formerly NUTEK, the Swedish Board for Tech. R&D) is acknowledged for financial support. We are grateful for input and feedback from Anders Blomdell (@control.lth.se), and we thank Klas Nilsson (formerly @control.lth.se and @abb.com, now @cs.lth.se) for inspiring suggestions to work in this direction. We also thank our GC-expert Roger Henriksson for many valuable comments.

References

[1] *IEC 1131-3, Programmable controllers, Part 3: Programming Languages*. International Electrotechnical Commission, 1992.

[2] K. J. Åström and B. Wittenmark. *Computer Controlled Systems: Theory and Design*. Prentice Hall, 3rd edition, January 1997.

[3] *ATmega103(L) Preliminary (Complete)*, Jan. 2000. *http://www.atmel.com/acrobat/doc0945.ps*.

[4] L. A. Bigagli. Real-time java, - a pragmatic approach. Master's thesis, Department of Computer Science, Lund Institute of Technology, October 1998.

[5] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[6] C. J. Cheney. A non recursive list compacting algorithm. *Commmunications of the ACM*, 13(11), 1970.

[7] Cygnus. The GNU compiler for the java language. *http://sourceware.cygnus.com*.

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11), 1978.

[9] J. Eker. A tool for interactive development of embedded control systems. In *Preprints 14th World Congress of IFAC*, Beijing, P.R. China, 1999.

[10] T. Ekman. A real-time kernel with automatic memory management for tiny embedded devices. Master's thesis, Department of Computer Science, Lund Institute of Technology, November 2000.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1st edition, August 1996.

[12] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology / Lund University, 1998.

[13] Jove, super optimizing deployment environment for java, July 1998. Instantiations Inc. *http://www.instantiations.com*.

[14] A. Ive. Implementation of an embedded real-time java virtual machine prototype. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, 2001. In preparation.

[15] P. Kouznetsov. Jad - the fast java decompiler, 2000. http://www.geocities.com/SiliconValley /Bridge/8617/jad.html.

[16] Java-cc parser generator. Metamata Inc. *http://www.metamata.com*.

[17] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. *http://www.irisa.fr/accueil/index_uk.htm*.

[18] K. Nilsen and S. Lee. Perc real-time api, July 1998.

[19] P. Persson. Predicting time and memory demands of object-oriented programs. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, April 2000.

[20] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and C. A. Watterson. Toba: Java for applications, a *Way ahead of Time* (wat) compiler. *http://www.cs.arizona.edu*.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), 1990.

[22] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. In *Proceedings of the IEEE*, volume 82, 1994.

[23] N. Shaylor. Jcc - a java to c converter. *http://www.geocities.com/ CapeCanaveral/Hangar/4040/jcc.html*.

[24] Java 2 platform micro edition (j2me) technology for creating mobile devices. http://www.java.sun.com, May 2000. Sun Microsystems Inc. White Paper. *http://www.java.sun.com*.

[25] Deploying high-performance and flexible server-side applications, an introduction to towerj3. Tower Technology Corporation. *http://www.towerj.com*.

[26] P. L. Wadler. Analysis of an algorithm for real time garbage collection. *Communications of the ACM*, 19(9), 1976.

[27] R. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of IWMM'92*. Springer-Verlag, 1992.

[28] Turboj. WindRiver Inc. *http://www.windriver.com*.