A Multithreaded Java Microcontroller for Thread-Oriented Real-Time Event-Handling

U. Brinkschulte, C. Krakowski
Institute of Process Control, Automation and Robotics
University of Karlsruhe
D-76128 Karlsruhe, Germany

J. Kreuzinger, Th. Ungerer
Institute of Computer Design and Fault Tolerance
University of Karlsruhe
D-76128 Karlsruhe, Germany

Abstract

We propose a multithreaded Java microcontroller—called Komodo microcontroller—with a new hardware event handling mechanism that allows handling of simultaneous overlapping events with hard real-time requirements. Real-time Java threads are used as interrupt service threads (ISTs) instead of interrupt service routines (ISRs). Our proposed Komodo microcontroller supports multiple ISTs with zero-cycle context switching overhead. We evaluate the basic architectural attributes using real-time event parameters of an autonomous guided vehicle. When calculating the maximum vehicle speed without violating the real-time constraints, ISTs dominate ISRs by a speed increase of 28%.

1. Introduction

The market for embedded systems is rapidly spreading towards network-connected appliances. There will be millions of mobile phones, set-top-boxes, personal digital assistants, network terminals and other internet devices, which operate in a network environment and are optimized for specific applications. Moreover, the ability of remote maintenance and diagnosis becomes increasingly important for embedded systems in industrial applications.

The support for internet-based programming convinced us to base our project on the Java language. Moreover, the object-orientation of Java offers the advantages of easy programming, software reuse and robustness. A large, standardized class library is provided. Java bytecode [1] is the Java target format of the Java Virtual Machine (JVM). Java bytecode (not the entire class-file) is much more compact than code of RISC processors. Furthermore Java bytecode is platform-independent. It can be downloaded on any kind of processor type. This is an important property for remote maintenance and diagnosis. The bytecode verifier allows a secure use of Java applications. It prevents misuse on the executing processor by a deficient or manipulated application.

A special requirement for embedded systems in system automation is real-time behavior which, among other things, means that one or more events trigger the execution of event-handling routines that must be serviced within a given time period. Usually interrupt service routines (ISRs) are used to implement event-handling routines. However, ISRs may miss hard real-time deadlines, in particular, when several hard real-time events that emerge in an irregular time pattern must be serviced.

In our approach we propose interrupt service threads (ISTs) as a new hardware event handling mechanism which allows handling of simultaneous overlapping events with hard real-time requirements. Java threads are used as ISTs, execution is supported by a multithreaded processor core. A multithreaded processor [2] stores multiple thread contexts in different register sets on the processor chip. In general, multithreading is used to bridge latencies by a context switch to another thread.

Our design target, however, is a multithreaded Java microcontroller—called Komodo microcontroller—that uses the fast context switch for IST-based event handling. The Komodo microcontroller enhances a simple Java processor core with multithreading to meet the requirement of predictability even in case of simultaneous overlapping events with hard real-time requirements. A hardware unit called priority manager supports several priority schemes. In particular, we propose a proportional share scheme where each IST is assigned a percentage of the full processor power.

2. Java and Real-Time

In its original form, Java isn’t well suited for the field of real-time systems [3]. The language contains no elements to define real-time requirements for program execution. Fur-
thermore, the timing behavior of many Java implementa-
tions is very slow due to interpretation of Java bytecode
or unpredictable due to Just-in-Time Compiler and garbage
collection. To overcome these disadvantages, the common
solution is the hybrid combination of Java with a real-time
operating system, like JWorks [4] or OS-9 [5]. Here, the
advantages of Java are used only for the non time-critical
parts of the application. The time-critical parts are realized
in a conventional way, e.g. by the C language. Java itself
has no real-time capabilities.

In more advanced solutions, real-time capabilities are
added to Java itself. Two basic types can be distinguished
for that kind of solution: The first type is based on a real-
time JVM, which is capable of executing Java bytecode in
real-time. A minimum requirement therefore is a real-time
garbage collection [6] and the real-time execution of Java
threads. This conception is followed e.g. by the realization
of Java real-time threads [7] by implementing a JVM on top
of the RT-Mach microkernel [8]. Another representative
of this type of solution is PERC [9], which allows the defi-
nition of timing constraints, resource negotiation and worst
case runtime analysis. All these solutions share the prop-
erty of bytecode interpretation, which leads to much slower
execution times in comparison to compiled C code.

The second type of real-time Java systems translates Java
bytecode to native machine code. A representative for that
type is JBed [10]. This technique can reach execution times,
which can compete with programs written in C++. But
by bypassing Java bytecode, these systems loose important
bytecode advantages like platform independence and secu-

The direct activation of an IST by the microcontroller
avoids latencies, which occur, if a thread is indirectly
activated by a ISR on a conventional processor, as is
done in concepts like AST [14].

ISTs allow flexible priority schemes. For example,
earliest deadline first or proportional share scheduling
can be used instead of fixed priority preemptive. Ear-
liest deadline first [13] schedules the processor to the
task closest to its deadline. Proportional share schedul-
ing [15] splits the processor execution cycles among
the concurrent tasks using a proportional ratio. Both
schemes allow processor charges of 100%.

Especially proportional share offers the possibility to as-
sign a guaranteed percentage of processor cycles to an
event. Herewith response times and data rates can be guar-
anteed even for several concurrent events independent of
other processor activities, as long as there is no overload
condition. Furthermore, overload conditions can be eas-
ily detected early, as soon as the total requested percentage
of processor cycles exceeds 100%. Scheduling strategies
like earliest deadline first detect overload conditions late by
missed deadlines.

A multithreaded microcontroller with zero-time context
switches [16] allows a very fine granular realization of the

3. Thread-based Real-time Event Handling

The conventional method of event handling on today’s
processors (including Sun’s Java processors) and microcon-
trollers is event handling by ISRs. Events of different prior-

A multithreaded microcontroller with zero-time context
switches [16] allows a very fine granular realization of the
proportional share scheduling scheme. The requested percentage can be guaranteed in a very short period of a few processor cycles. This offers e.g. the possibility of debug threads, which monitor the system at a low percentage without changing real-time behavior.

For that reasons, ISTs can simplify and improve the programming of concurrent overlapping real-time events. Section 5 will demonstrate this using a real industrial application example.

4. The Komodo Microcontroller

The Komodo microcontroller is a multithreaded Java microcontroller which supports multiple ISTs with several priority schemes. In particular we focus on the proportional share scheme. The microcontroller holds the contexts of up to four threads. If there are more then four threads to handle, our architecture allows up to three real-time threads, which are directly mapped to hardware threads. The remaining threads must be non real-time and are scheduled within the fourth hardware thread. To scale up for larger systems with more than three real-time threads, we propose a parallel execution on several microcontrollers connected by a middleware platform. This topic is covered in another paper [17].

Because of its application for embedded systems, the processor core of the Komodo microcontroller is kept at the hardware level of a simple scalar processor. As shown in Fig. 1, the microcontroller core consists of an instruction-fetch unit, a decode unit, a memory access unit (MEM) and an execution unit (ALU). Four stack register sets are provided on the processor chip. A signal unit triggers IST execution due to external signals.

![Block diagram of the Komodo microcontroller](image)

**Figure 1.** Block diagram of the Komodo microcontroller

The instruction fetch unit holds four program counters (PC) with dedicated status bits (e.g. thread active/suspended), each PC is assigned to another thread. Four byte portions are fetched over the memory interface and put in the according instruction window (IW). Several instructions may be contained in the fetch portion, because of the average bytecode length of 1.8 bytes. Instructions are fetched depending on the status bits and fill levels of the IWS.

The instruction decode unit contains the above mentioned IWS, dedicated status bits (e.g. priority) and counters for the implementation of the proportional share scheme. A priority manager decides subject to the bits and counters from which IW the next instruction will be decoded. Besides the proportional share scheme also other priority schemes may be supported by the priority manager. The priority manager applies one of the implemented thread priority schemes for IW selection. A bytecode instruction is decoded either to a single micro-op, a sequence of micro-ops, or a trap routine is called. Each opcode is propagated through the pipeline with its thread id. Opcodes from multiple threads can be simultaneously present in the different pipeline stages.

The instructions for memory access are executed by the MEM unit. If the memory interface only permits one access each cycle, an arbiter is needed for instruction fetch and data access. All other instructions are executed by the ALU unit. Both units (MEM and ALU) can take several cycles to complete an instruction execution. After that, the result is written back to the stack register set of the according thread.

External signals are delivered to the signal unit from the peripheral components of the microcontroller core as e.g. timer, counter, or serial interface. By the occurrence of such a signal the corresponding IST is activated. As soon as an IST activation ends its assigned real-time thread is suspended and its status is stored. An external signal may activate the same thread again.

To avoid pipeline stalls, instructions from other threads can be fed into the pipeline. Idle times may result from branches or memory accesses. The decode unit predicts the latency after such an instruction, and proceeds with instructions from other IWSs. There is no overhead for such a context switch. No save/restore of registers or removal of instructions from the pipeline is needed, because each thread has its own stack register set.

Optimizations like folding [12] can be applied in the decode unit. A hardware mechanism to avoid stack overflow called dribbler can only be used for stack register sets assigned to non real-time threads due to the unpredictable occurrence of a fill or spill action.

Because of the unpredictability of cache accesses, a non-cached memory access is preferred for real-time microcontrollers. The emerging load latencies are bridged by scheduling instructions of other threads by the priority manager. Therefore a cache is omitted from our Komodo microcontroller.
5. Evaluation using an industrial application example

This section gives an evaluation of the IST technique and of the proportional share scheme, which are the basic architectural attributes of the Komodo microcontroller. The evaluation is done using a real industrial application example of autonomous guided vehicles (AGV). This is an appropriate example, because several concurrent time-critical events occur at the control of vehicles in an AGV system. Vehicles are guided by a reflective tape glued on the floor. A vehicle pursues its track by use of a CCD line camera producing periodic events at a rate of 10 milliseconds. This period gives the deadline for converting and reading the camera information and for executing the control loop, which keeps the vehicle on the track. A second time-critical event is produced asynchronously by transponder-based position marks, which notify the vehicle that some default position is reached (e.g. a docking station). If the vehicle notices a position mark, the corresponding transponder which is installed in the floor beside the track, must be read. The precision needed for position detection using these marks is 1 cm. This gives a vehicle speed dependent deadline for reading the transponder information. The vehicle speed can vary in the range of 0.5 to 1 meters per second, which results in a deadline range from 20 to 10 milliseconds.

To solve this job, the vehicle software is structured into three tasks. The first task (control task) performs the control loop based on the actual camera information. This task is triggered by a timer event with a period of 10 milliseconds. The second task (camera task) converts and reads the next camera information. This task is triggered by the same timer event. The third task (transponder task) is triggered by the position mark events. It reads the transponder information and calculates the current position and the action to be taken.

To evaluate the benefits of the IST architecture, we compare the real-time behavior of three different realizations of these tasks:

1. by conventional ISRs
2. by ISTs using proportional share scheduling
3. by ISTs using earliest deadline first scheduling

To allow a fair comparison, we assume an identical processor performance for all three techniques. This is based on the real performance of a 20 MHz M68302 microcontroller which is similar to the expected performance of the proposed Komodo microcontroller.

The evaluation itself is done as follows: first we examine the three techniques at a fixed vehicle speed of 0.65 meters per second. Then we calculate the maximum speed that can be reached for each technique without violating the real-time constraints.

The following table summarizes the basic values for the first evaluation:

<table>
<thead>
<tr>
<th></th>
<th>vehicle speed</th>
<th>0.65 m/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>camera period</td>
<td>10 msec</td>
<td></td>
</tr>
<tr>
<td>position mark precision</td>
<td>1 cm</td>
<td></td>
</tr>
<tr>
<td>control task calculation time</td>
<td>5 msec</td>
<td></td>
</tr>
<tr>
<td>camera task execution time</td>
<td>1 msec</td>
<td></td>
</tr>
<tr>
<td>transponder task execution time</td>
<td>5.5 msec</td>
<td></td>
</tr>
</tbody>
</table>

This gives the following execution-time / deadline ratio:

<table>
<thead>
<tr>
<th></th>
<th>5 msec / 10 msec</th>
<th>= 50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>camera task</td>
<td>1 msec / 10 msec</td>
<td>= 10%</td>
</tr>
<tr>
<td>transponder task</td>
<td>5.5 msec / 15.4 msec</td>
<td>= 35%</td>
</tr>
</tbody>
</table>

1) 15.4 msec = 1 cm / 0.65 m/sec

1. Realization with ISRs

At first we demonstrate the solution with a conventional ISR realization of the tasks. The priority scheme of ISRs is a fixed priority scheme. Therefore problems can be expected, because fixed priority scheduling can only guarantee a processor utilization of 78% for three events [13], which is less than the needed 95%. This can be proved with the above scenario. The first priority constraint is as follows:

$$P_{control} > P_{transponder}$$

If this is not fulfilled, the deadline of the control task can be missed even without regarding the camera task, as shown in figure 2.

Figure 2. ISRs with $P_{control} < P_{transponder}$

This implies further that the control task must be able to preempt the transponder task, so interrupts must be enabled during ISR execution (fixed priority preemptive).

The second priority constraint is the following:

$P_{camera}$ must not be the lowest priority

If this is not fulfilled, the deadline of the camera task can be missed, as shown in figure 3.

Regarding the constraints, two out of six priority assignments remain:

1. $P_{control} > P_{camera} > P_{transponder}$
2. $P_{camera} > P_{control} > P_{transponder}$
The proportional share scheduling assigns a scheduling weight to each task, and the ISR solution cannot guarantee the specified deadlines. In assignment 2, the control task and camera task are just flipped. The result is that earliest deadline first scheduling meet all deadlines as well as proportional share scheduling, with a percentage of 95%, which means all deadlines will be met in any case. But these remaining assignments also can't guarantee the deadlines as shown for assignment 1 in figure 4. Assignment 2 produces the same deadline miss, the order of control task and camera task are just flipped. The result is that the ISR solution cannot guarantee the specified deadlines.

Figure 3. ISRs with $P_{camera}$ is lowest priority

Figure 4. ISRs with $P_{control} > P_{camera} > P_{transp}$

2. Realization with ISTs using proportional share scheduling. The proportional share scheduling assigns a guaranteed percentage of processor cycles to a thread. On the proposed Komodo microcontroller with its zero time context switches, this percentage is guaranteed in a very short period of a few processor cycles by the hardware priority manager. So the realization of the three tasks is simple: each task is assigned to a thread (IST) with the execution-time/deadline ratio as requested percentage of processor cycles: 50% for the control task, 10% for the camera task and 35% for the transponder task. This gives a total requested percentage of 95%, which means, all deadlines will be met in any case. Furthermore there is an additional spare of 5%, which e.g. could be used for a debug thread.

Figure 5 shows the worst case scenario, where all events occur at the same time.

3. Realization with ISTs using earliest deadline first scheduling. Earliest deadline first scheduling is very popular in real-time programming. It assigns the processor to the task with the nearest deadline. Like proportional share scheduling, earliest deadline first scheduling allows a processor utilization of 100% [13]. This means, ISTs with earliest deadline first scheduling meet all deadlines as well in any case. Figure 6 shows the above scenario using earliest deadline first scheduling.

Figure 5. ISTs with Proportional share

Figure 6. ISTs with earliest deadline first

But figure 6 reveals a disadvantage of earliest deadline first scheduling. The camera task and the transponder task must not only meet a deadline, they deal as well with data rates. The transponder task must read the transponder information, which comes on a 19200 Baud serial link. Information transfer (8 bytes) starts at the position mark event and lasts 4 milliseconds. In case of proportional share scheduling this is no problem, because the transponder task starts at the position mark event as well with 35% of processor cycles, which is enough to read the information. In case of earliest deadline first scheduling, the transponder task starts 6 milliseconds after the position mark event. This means, if the serial link doesn’t contain a byte hardware buffer, the information will be lost.

The camera task has to read the CCD line camera information, which is 64 pixels of 8 bytes. Information transfer is synchronized by a clock cycle produced by the camera task. The maximum time to read this information is 10 milliseconds. After that, the information is overwritten in the camera by the next line. In case of proportional share scheduling, this maximum available amount of time is used. This means, the AD converter connected to the camera must offer a conversion time of 10 msec / 64 = 156 μsec. In case of earliest deadline first scheduling, the camera is read in 1 millisecond. So we need a 10 times faster AD converter with a conversion time of 1 msec / 64 = 15.6 μsec.

As a conclusion the ISR solution isn’t able to guarantee the specified deadlines, in contrast to the IST solution, even offers a spare of 5%. Moreover, proportional share scheduling offers an advantage over earliest deadline first scheduling.
scheduling, if not only deadlines, but also data rates must be met.

**Maximum vehicle speed for IST and ISR**  In a second step, the maximum vehicle speed without violating the real-time constraints can be calculated. For ISTs using proportional share scheduling, the maximum vehicle speed can be reached, if the transponder task uses the remaining spare of 5%. This leads to a transponder task of 40% and a total processor utilization of 100%. In this case, the transponder task can guarantee a deadline of (5.5 msec * 100%) / 40% = 13.75 msec. With this deadline, the position resolution of 1 cm can be retained for a vehicle speed of:

\[
V - IST_{max} = 0.01m / 0.01375sec = 0.73m/sec
\]

This calculation is valid for ISTs using earliest deadline first scheduling as well, because the same processor utilization of 100% is reached.

In case of ISRs, the reachable deadline for the transponder task can be taken from Fig. 4. It calculates to 5 msec + 1 msec + 4 msec + 5 msec + 1 msec + 1.5 msec = 17.5 msec. So the maximum vehicle speed can be calculated to:

\[
V - ISR_{max} = 0.01m / 0.0175sec = 0.57m/sec
\]

As result, the vehicle speedup for the IST architecture compared to ISR is:

\[
Speedup_{IST/ISR} = \frac{(0.73 - 0.57)}{0.57} = 28%
\]

In summary, this section evaluates the benefit of the multithreaded architecture for realizing ISTs. Avoiding latencies by multithreading is not regarded, which can bring an additional benefit [16]. So the calculated values are worst-case values.

### 6. Conclusions

We base our real-time project upon the Java language because of its internet support, its wide spread use, its object-orientation and its platform independence. However, our analysis shows that the use of Java in a real-time environment causes several problems which are solved by our approach. We combine a multithreaded Java microcontroller with interrupt service threads (ISTs) as a new hardware event handling mechanism which allows handling of simultaneous overlapping events with hard real-time requirements. In particular, we propose a proportional share scheme where each real-time thread is assigned a percentage of the full processor power. The overhead for thread based even handling is the overhead of multithreading with multiple register sets.

The evaluation of our approach using a real industrial application example shows, that the conventional ISR solution isn’t able to guarantee the specified deadlines, in contrast to the IST solution, that even offers a spare of 5%. Moreover, proportional share scheduling offers an advantage over deadline first scheduling, if not only deadlines, but also data rates must be met. When calculating the maximum vehicle speed without violating the real-time constraints our example shows that using ISTs instead of ISRs allows a 28% higher speed of the autonomous vehicle.

### References