Multicore Support for Realtime Java

Andy Walter
aicas GmbH
Haid-und-Neu-Str. 18
76131 Karlsruhe, Germany

September 9, 2008

1 Abstract

Real-Time Java is getting more and more popular in Embedded Systems. The object-oriented approach significantly increases the productivity of developers. The open standard RTSJ [2] (Real-Time Specification for Java) extends the Java paradigm "write once, run everywhere" to real-time Java applications.

At the same time, multicore technology is finding it's way into Embedded Systems as well: The number of CPUs and CPU cores per Embedded board is steadily increasing. In Desktops and Servers, SMP and multicore CPUs have been state-of-the-art for many years, because increasing the clock cycle is getting more difficult and consumes too much power and space. As usual, Embedded Systems follow the trends that were set by Desktops and Servers with a delay of 5-10 years.

However, the software still has to catch up with the new situation. Most Embedded Operating Systems and Software Development Tools are not ready for multicore yet. E.g., the Real-Time Specification for Java, RTSJ, does not currently allow to achieve maximum advantage of such new devices. The Jeopard project makes real-time Java suitable for multicore applications: The Garbage Collector needs to be extended, the existing Standard classes and RTSJ API needs to be checked for possible problems in multicore environments. Development of complex, safe, real-time multicore applications needs to be made easier. Even ordinary Java applications using several threads should benefit when running on a multicore device.

2 Java goes Multicore

Just extending the tools is not enough: Also, a new kind of programming paradigm is necessary for the development of Embedded Systems. Generally, simply running an application on a multicore hardware is not sufficient. Applications with just one main thread need to be split up in order to benefit

from multiple cores. If this is not possible (or would be too much effort), at least VM-internal activities such as Garbage Collection and Finalizers will be executed in parallel and thus increase the performance of the application. But without parallelising the application itself, hardly any advantage can be taken of more than one additional CPU core. Luckily, many Java applications make use of multi-threading already: E.g., graphical applications usually are split up at least into a painting thread, which (re-)paints a certain area of the screen whenever it becomes necessary, a communication thread for gathering I/O, and some kind of event handling. Other activities which can easily be put on additional CPU cores are precomputation of data to fill a cache (e.g., parts of a graphical application which are currently not visible) or a JIT compiler. Making a real-time application to benefit from a multicore environment is usually more difficult. For such applications, worst-case execution time (WCET) is more important than mere performance. Quite often, real-time applications have very little activity in some high-priority threads and the main activity in a single main thread. Since cache or JIT don't improve the WCET, such applications have to be modified in order to take advantage of multicore technology.

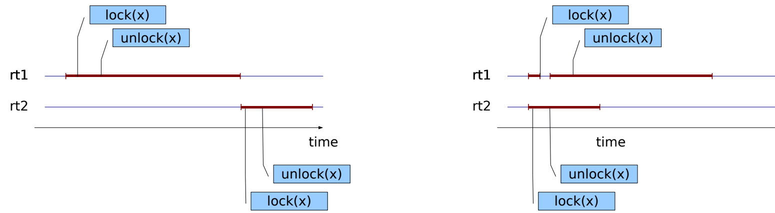
3 Extended API

The Jeopard team is working on an extended API to support developers who want to write applications for a multicore device. A common design pattern for multi-core devices is a parallel apply: A method should be invoked on a set of values, regardless of the invocation order. An automated load balancing can most efficiently use all available cores for this task, and still ensure scalability.

In many cases, the load balancing can be done automatically, but in some cases, developers may have to control it explicitly. E.g., developers might want to set the affinity of a thread or an event handler to a certain CPU or set of CPUs, thus specify that not all threads may be executed on all CPUs. This can be used to ensure, e.g., that a thread still finds it's data in the CPU cache next time it is executed. In some cases, multicore systems may even block high-priority threads by synchronising on the same lock (see figure 1). If two threads synchronise on the same monitor several times, the higher priority thread *rt1* may block each time, if the monitor is held by the lower priority thread *rt2*. On a single-core system, *rt1* would block at most once in this case. This problem can be avoided by setting the affinity: If all threads that synchronise on the same locks are forced to run on the same CPU, this problem cannot occur. But manually setting the affinity reduces the efficiency of the automated load balancing. If possible, the load balancer should consider this automatically.

4 Multicore Pitfalls

Some programming techniques which are considered bad style in a single core device can cause severe problems in a multicore environment. While in a single



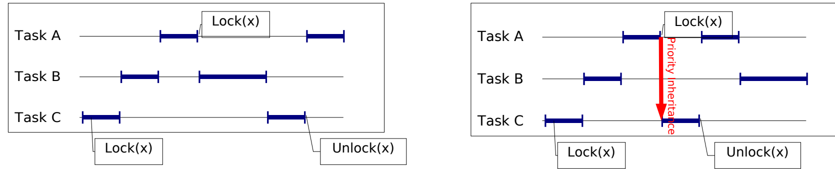
Two threads competing for a lock in a single-core . . .   . . . and in a multi-core environment

Figure 1: *Running an application in a multi-core system may block high-priority threads more often than a single core system.*

core system, wrong synchronisation usually causes race conditions - which are hard to find, but possibly never occur - a missing synchronisation most definitely causes a multi core system to either crash or at least to return the wrong results. Therefore, the Jeopard project is also working on style guides for multicore developers. E.g., in a single-core environment, the thread with highest priority can always rely on the fact that it will not be interrupted by any other thread. This does not only affect timing (as in the previous example in figure 1), but could also lead to broken data structures. Even though this "synchronising by priority" is considered bad style even in a single-core environment—extending the application with a higher-priority thread or code-reuse for other applications become dangerous with this technique—it could be used there, but such an application will fail in a multi-core environment.

An example for this behaviour can be found in the RTSJ library: To solve the problem of Priority Inversion, RTSJ offers two protocols, Priority Inheritance (PI) and Priority Ceiling Emulation (PCE). PI raises the priority of a thread which is holding a lock to the priority of the thread which is acquiring the lock (see figure 2), while PCE raises the priority of a thread which enters a lock to the highest priority of which might ever acquire this particular lock (see figure 3). The most important advantage of PCE compared to the default PI is, that PCE not only avoids Priority Inversion, but is also guaranteed to avoid deadlocks in a single-core environment. This is based on the assumption that entering a PCE lock can never block at all—which is not true in a multi-core environment, as shown in figure 3. While PCE could still be used in order to prevent Priority Inversion in a multi-core environment, it loses it's biggest benefit over PI.

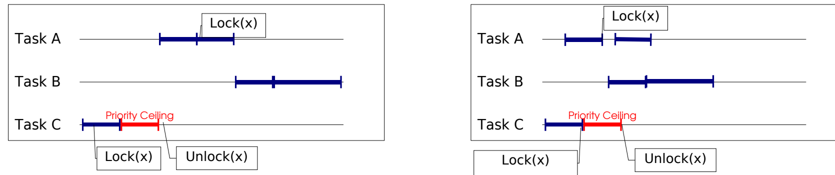
For the development of future embedded applications, it is recommended not to count on a particular number of cores or CPUs in the target system. The current project might use a dual-core CPU, while the next project might run on a hardware with more or even with less cores. Future development tools should abstract as much as possible the number of cores, just like Embedded Java abstracts the CPU architecture and Operating System.



Priority Inversion can cause high-priority threads to block.

Priority Inheritance raises the priority of threads that holds a semaphore to the priority of the thread which is waiting for this semaphore.

Figure 2: *A Standard Problem and a protocol to avoid it: Priority Inversion and Priority Inheritance.*



Entering a Priority Ceiling monitor never blocks in single-core system . . .

. . . but it does in multi-core systems.

Figure 3: *Priority Ceiling Emulation loses it's advantages on multi-core.*

5 Conclusion

Multicore systems are becoming mainstream in embedded devices, but current software development tools, as well as Embedded developers still need to catch up with the new situation. Multicore has been established in desktop systems and servers, where performance is important, but realtime is not an issue. Those big systems usually run several applications at the same time, such that they can easier benefit from more than one CPU than an embedded system can, which often just runs one single application. The JEOPARD project is an important mile stone for this new technology. It provides additional libraries and guidance to embedded developers who have to go multicore. This can help saving hardware costs and allows for more complex applications in embedded systems. The partners are aicas GmbH (Germany), EADS Deutschland GmbH (Germany), FZI Karlsruhe (Germany), RadioLabs (Italy), SkySoft (Portugal), Sysgo (France), Technical University Cluj-Napoca (Romania), Technical University of Vienna (Austria), and the University of York (UK).

References

[1] M. Aldea and et al. Fsf: A real-time scheduling architecture framework. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 113 – 124, 2006.

[2] Greg Bollela. *Real-Time Specification for Java*. Addison-Wesley, 2001.

[3] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70, 2007.

[4] Peter Dibble. JSR 282: RTSJ version 1.1. *http://jcp.org/en/jsr/detail?id=282*.

[5] HIJA, High-Integrity Java, Project Number IST-511718 of the sixth framework programme of the European Commission. www.hija.info, 2004-2006.

[6] C. Douglass Locke. JSR 302: Safety Critical Java Technology. *http://jcp.org/en/jsr/detail?id=302*.

[7] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[8] Fridtjof Siebert. Limits of parallel marking garbage collection. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 21–29, New York, NY, USA, 2008. ACM.

[9] A.J. Wellings. Multiprocessors and the real-time specification for java. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing ISORC-2008*, pages 255–261. Computer Society, IEEE, IEEE, May 2008.

[10] A.J. Wellings. Multiprocessors and the real-time specification for Java. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing ISORC-2008*, pages 255–261. Computer Society, IEEE, IEEE, May 2008.