Implementing Predictable Scheduling in RTSJ-Based Java Processor

Zhilei Chai[1,2], Wenbo Xu[1], Shiliang Tu[2], and Zhanglong Chen[2]

[1] Center of Intelligent and High Performance Computing, School of Information Engineering, Southern Yangtze University
214122 Wuxi, China
[2] Department of Computer Science and Engineering, Fudan University
200433 Shanghai, China
zlchai@fudan.edu.cn

Abstract. Due to the preeminent work of the RTSJ, Java is increasingly expected to become the leading programming language in embedded real-time systems. To provide an efficient real-time Java platform, a Real-Time Java Processor (HRTEJ) based on the RTSJ was designed. This Java Processor efficiently implements the scheduling mechanism proposed in the RTSJ, and offers a simpler programming model through meliorating the scoped memory. Special hardwares are provided in the processor to guarantee the Worst Case Execution Time (WCET) of scheduling. In this paper, the scheduling implementation of this Java Processor is discussed, and its WCET is analyzed as well.

1 Introduction

Currently, to provide an efficient Java platform suitable for real-time applications, many different implementations are proposed. These implementations can be generally classified as **Interpreter** (such as RJVM [1] and Mackinac [2]), **Ahead-of-Time Compiler** (Anders Nilsson et al [3]) and **Java Processor** (such as aJile-80/100 [4] and JOP [5]). Comparing with other implementing techniques, Java Processor is preferably being used in embedded systems because of its advantages in execution efficiency, memory footprint and power consumption.

The scheduling predictability is a basic requirement for real-time systems. The Real-Time Specification for Java (RTSJ) [6] makes some major improvements to Java's thread scheduling. Many of the current real-time Java platforms implement scheduling based on the RTSJ, such as Mackinac, RJVM and aJile etc, most of which allow threads to be allocated in heap memory. JOP implements a new and simpler real-time profile other than the RTSJ.

In this paper, the scheduling implementation in our RTSJ-based real-time Java Processor is introduced. None of the thread in this Processor is allocated in heap and the interference of Garbage Collector is totally avoided. Comparing with JOP, the profile of the HRTEJ Processor is more close to the RTSJ and dynamic thread creation and termination are supported.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part I, LNCS 3991, pp. 1043 – 1046, 2006.

2 Scheduling Implementation in the HRTEJ Processor

Based on the optimization method proposed in our previous work [7], to guarantee the real-time performance of the HRTEJ Processor, standard Java class files was processed by the *CConverter* (the program we designed to preprocess the Java class file) before being downloaded into the processor's memory. During this phase, all the process interfering predictability such as Class loading, verifying and resolution are handled. Some other optimizing operations are processed simultaneity. All the Classes needed in the application are loaded and linked before execution. The memory layout produced by the *CConverter* is displayed as a binary sequence. All of the strings, static fields and other data can be accessed by their addresses directly.

2.1 Thread Management Mechanism in the HRTEJ Processor

There are some thread related registers in the HRTEJ Processor to facilitate the predictability of the scheduling as follows:

Run_T, Ready_T, Block_T and Dead_T: *n*-bit (n is the width of the data path) registers to record the queues of threads which are running, ready, blocked and dead. A thread can be put into a queue by setting corresponding bit of that register to '1' according to its priority.

ThisThread: recording the object reference of current running thread.

Wait_Base: The base address of the static fields WaitObject0~n-1 in figure 1.

STK_base0~n-1: the stack base address of each thread.

LTMAddr0~n-1: the *LTMomory* base address of each thread.

The HRTEJ Processor can support *n* threads at most with unique priority from 0 to n-1 (0 is the highest priority).These threads can be created and terminated dynamically.

Creating a new thread just as creating a general object, but the object reference of this thread should be put into the corresponding static field '*Thread0~n-1*' according to its priority. The *Scheduler* terminates a thread by moving the corresponding '1' from other queues to the *Dead_T*. The *Scheduler* always chooses the thread corresponding to the leftmost '1' in *Ready_T* to dispatch and execute.

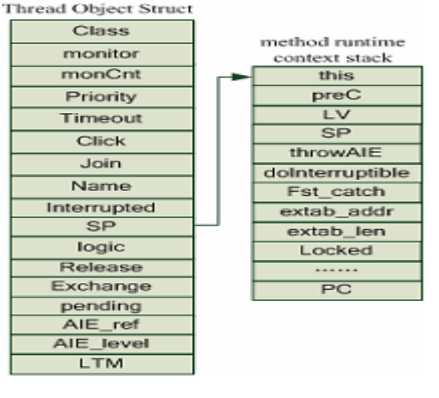

Fig. 1. Tread Object Structure of the HRTEJ Processor

When scheduling occurs, the context of the thread being preempted is saved at the top of its stack. The thread object and its corresponding context are shown in fig. 1.

Wait Method Implementation: When a thread calls the wait method and blocks itself, it records the reference of the waited object in corresponding static field *WaitObject0~n-1*. This static field will be checked when notifying a thread. *WaitObject0~n-1* is used to record the locked object waited by each thread.

Join Method Implementation: Using the instance field '*join*' to record the object reference of the thread to wake up when current thread is finished.

Priority Inheritance Implementation: If a thread wants to enter a synchronization block which another lower priority thread is in, then the Priority Inheritance must be taken. In the HRTEJ Processor, a simple method to implement the Priority Inheritance is adopted. Two threads sharing the same synchronization object exchange their priority, and record the old priority in the *Exchange* field. When the thread exits the synchronization area, it takes the original priority back again.

As discussed above, special hardwares are used in the HRTEJ Processor to ensure the predictability of WCET. The clock cycles of thread scheduling, dispatching, and other thread related mechanisms are all predictable in the HRTEJ Processor. The implementations of other scheduling related mechanisms described in [7] and [8], will not be discussed anymore.

3 Evaluation and Discussion

The HRTEJ differs from JOP in supporting dynamic thread creation and termination, ATC, nested scoped memory, and dynamic allocation of shared objects, which provides a more flexible programming model. Table 1 shows the comparison of some bytecodes execution cycles between the JOP and the HRTEJ. It is displayed that the average execution cycles of the HRTEJ is smaller than that of JOP.

Table 1. Clock Cycles of Bytecode Execution Time

	HRTEJ(min)	HRTEJ(max)	JOP
iload iadd	3	3	2
iinc	8	8	11
ldc	7	8	10
if_icmplt	8	10	6
getfield	6	7	25
getstatic	7	8	17
iaload	3	3	30
invoke	42	43	128
invoke static	39	39	101
dup	2	2	x
new	10	12	x
iconst_x	2	2	x
astore_x / aload_x	3	5	x
return	20	20	x
goto	3	5	x

Estimating the WCET of tasks is essential for designing and verifying real-time systems. As a rule, static analysis is a necessary method for hard real-time systems. Hence, the WCET of an application (*Demo.java*) is statically analyzed in this paper to demonstrate the real-time performance of the HRTEJ Processor.

The bytecodes compiled from *Demo.java* can be mainly partitioned into 3 parts (one part a thread). In each part, the WCET of the general bytecode is known according to table 1. For the finite loop in thread t0, its WCET can be calculated as *100*WCET (general code + LTMemory + start() + Scheduling)*. The LTMemory operation is predictable as mentioned in [8], and the WCET of the scheduling and method start() is also predictable. So, the real-time performance of the whole application can be guaranteed.

Furthermore, the maximal allocation of the LTMemory space in this application is *S(t0)+S(t1)* instead of *S(t0)+100*S(t1)*. S(t) denotes the space of thread t. Another advantage of this processor is that Java programmers just need creating and entering a LTMemory space to use instead of denoting the memory size.

4 Conclusions

The multithreading mechanism is vital for real-time systems to handle the concurrent events in the real world. The RTSJ defines more accurate semantics for the predictable scheduling. It makes Java become popular in embedded real-time systems. In this paper, the RTSJ based scheduling mechanism implemented in our Java Processor is introduced. With special architectural supporting, all the WCET of the thread related mechanisms are predictable. Because heap memory is not used, this Processor is suitable for hard real-time applications.

References

1. http://www.cs.york.ac.uk/rts/
2. G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain, "Mackinac: making HotSpot/spl trade/ real-time," presented at Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2005, 45 – 54.
3. A. Nilsson and S. G. Robertz, "On real-time performance of ahead-of-time compiled Java," presented at Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2005, 372 – 381.
4. http://www.ajile.com/
5. M. Schoeberl, "JOP: A Java Optimized Processor for Embedded Real-Time Systems", http://www.jopdesign.com/thesis/thesis.pdf, 2005
6. G. Bollela, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin and M. Trunbull, "The Real-Time Specification for Java", Addison Wesley, 1st edition,2000.
7. Z. L. Chai, Z. Q. Tang, L. M. Wang, and S. L. Tu, "An Effective Instruction Optimization Method for Embedded Real-Time Java Processor," 2005 International Conference Parallel Processing Workshops, Oslo, Norway, pp. 225-231, 2005.
8. Z. L. Chai, Z. L. Chen, and S. L. Tu, "Framework of Scoped Memory in RTSJ-Compliant Java Processor", Mini-Micro Systems, accepted.