Implementing RTSJ-optimized Java Processor for Real-Time Applications

Zhilei Chai[1], Wenke Zhao[2], Wenbo Xu[1]

[1] Center of Intelligent and High Performance Computing, School of Information Technology, Southern Yangtze University, 214122, Wuxi, China
zlchai@fudan.edu.cn
xwb@sytu.edu.cn
[2] Department of Computer Science and Engineering, Fudan University
200433, Shanghai, China
052021110@fudan.edu.cn

Abstract. Due to the preeminent work of the Real-Time Specification for Java (RTSJ), Java is increasingly expected to become the leading programming language in embedded real-time systems. To provide an efficient Java platform suitable for real-time applications, a hard real-time Java processor (HRTEJ) can execute Java bytecode directly is proposed in this paper. It efficiently supports mechanisms specified by the RTSJ and offers a simpler programming model through ameliorating the scoped memory of the RTSJ. The worst case execution time (WCET) of this processor is predictable by employing the optimization method proposed in our previous work[1], in which all the processing interfering predictability is processed before bytecode execution. Further advantage of this method is to make the implementation of the processor straightforward and suited to a low-cost FPGA chip.

1 Introduction

The complexity of embedded real-time systems keeps growing and novel methods and tools are required. With the advantages as an object-oriented and concurrent programming language, and with the occurrence of the real-time specification for Java (RTSJ)[2], Java is increasingly expected to become the leading programming language in embedded real-time systems.

Currently, to provide an efficient Java platform suitable for real-time applications, many different implementing methods are tried. real-time Java platforms can be generally classified as follows according to their implementing methods: 1). **Interpreter**, in this manner, Java platform is an interpreting program running on the operating system that can interpret and execute java bytecode under the assistance of the OS. RJVM[3], JTime[4], and Mackinac[5] all belong to this way. 2). **Ahead-of-Time** Compiler, in this manner, Java bytecode is compiled into native code or an intermediate language (e.g. C) in advance. Anders Nilsson et al[6] employed this method to insure real-time performance of the Java applications. In their implementation, C is taken as the intermediate language. 3). **Java Processor**, in this manner, the Java plat-

© L. Sánchez, O. Espinosa (Eds.)
Control, Virtual Instrumentation and Digital Systems.
Research in Computing Science 24, 2006, pp. 179-188

form is implemented directly on silicon. It not only avoids the overhead of translation of the bytecode to another processor's native language, but also provides special support for Java runtime features such as stack processing, multithreading etc. aJile[7], FemtoJava[8] and JOP[9] are all real-time Java processors implemented in the FPGA.

Comparing with other implementing techniques, Java processor is preferable in embedded systems for its high execution efficiency and low memory footprint. Therefore, it becomes popular in implementing embedded real-time Java platforms. Presently, FemtoJava and JOP do not put emphasis on supporting the RTSJ. aJile systems announces the RTSJ will be supported on top of the aJ-80 and aJ-100 chips.

In this paper, we propose a simple Java processor suitable for embedded real-time applications and it is dedicated to supporting RTSJ features more efficiently. This processor implements RTSJ mechanisms effectively and offers a simpler programming model through ameliorating the scoped memory of the RTSJ. According to the optimization method proposed in work[1], all the processing interfering predictability is completed before bytecode execution, thus, this processor is simple to implement and its worst case execution time (WCET) is more predictable.

2 RTSJ related concepts

RTSJ is a specification for extending the Java language specification[10] and the Java virtual machine specification[11], and providing an application programming interface(API) that will enable the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints. It contains some enhanced areas such as thread scheduling and dispatching, synchronization and resource sharing, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, memory management, and physical memory access. The details of the RTSJ can be found in [2]. In this paper, we put emphasis upon the special support for the RTSJ mechanisms in our processor, the methods to guarantee its real-time performance is discussed as well.

3 Implementation of the HRTEJ processor

As described in [1], to guarantee the real-time performance of the processor, standard Java class files was processed by the *CConverter* (the program we designed to preprocess the Java class file) before being executed on the processor. During this phase, all the classes needed in the application are loaded, linked and transformed into the binary representation shown in Fig.1 that can be executed directly on the processor. Once power on, the HRTEJ processor starts to execute *Initial* code to do system initialization according to the *Initial* parameters. Then, it enters main thread creating code and executes bytecode from there step by step.

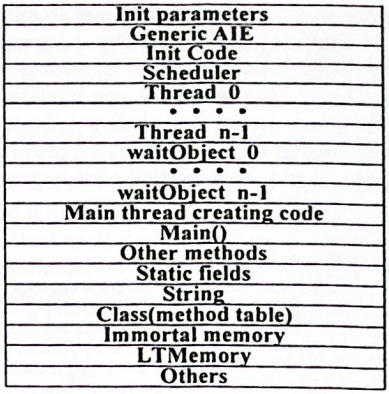

Fig.1. Memory layout of the binary representation.

3.1 Thread management in the HRTEJ processor

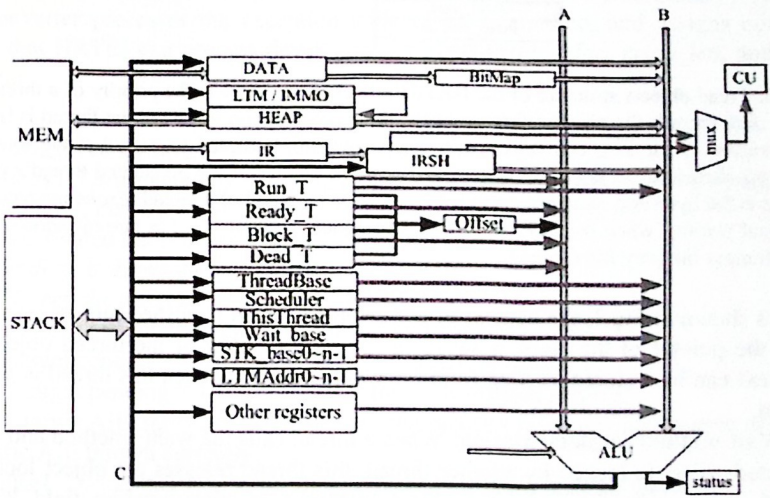

Fig.2. Thread related architecture of the HRTEJ processor. *Run_T*, *Ready_T*, *Block_T* and *Dead_T* are the n-bit (n is the width of the data path) registers to record the queues of threads which are running, ready, blocked and dead respectively. A thread can be put into a queue by setting corresponding bit of that register to '1' according to this thread's priority. *ThisThread* records the object reference of current running thread. *Wait_Base* is the base address of the static fields *WaitObject0~n-1* in Fig.1. *STK_base0*~n-1 is the base address of the stack for each thread. *LTMAddr0*~n-1 is the base address of the LTMemory memory for each thread.

The HRTEJ processor can support n (n is the width of the data path) threads at most with unique priority from 0 to n-1 (0 is the highest priority).These threads (several

threads are kept for processor itself) can be created and terminated dynamically during execution. Creating a new thread just like creating a general object, but the object reference of this thread should be put into the corresponding static field *Thread0~n-1* according to this thread's priority. The processor can terminate a thread by moving the corresponding '1' from other queues to the *Dead_T* according to its priority. When scheduling occurs, the context of the preempted thread is saved at the top of its own stack. The thread object and its corresponding context (registers in the processor) are shown in Fig.3.

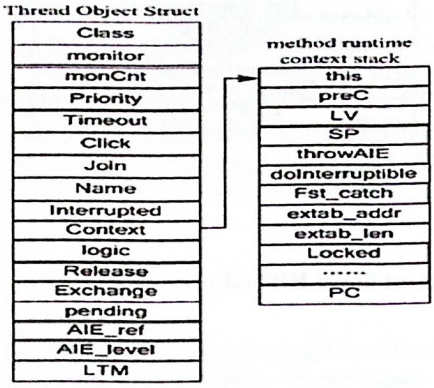

Fig.3. Tread objects structure of the HRTEJ processor. *Priority* is the priority of a thread (0 to n-1). *Join* records the object reference of the thread to wake up when current thread is finished. *Interrupted* is the flag compatible with the conventional thread to record if the *interrupt*() method of this thread is called. *Context* is the stack pointer of the preempted thread's context. *Logic* is the bytecode sequence executed by the *run*() method of a thread. *Exchange* records the original priority when priority inheritance avoidance is taken. *LTM* is current address of the LTMemory memory for a thread.

As shown in Fig.3, the context of a preempted thread is pushed into its own stack, and the pointer of the stack is saved in the field *Context* of the thread object. The context can be restored starting from the *Context* pointer when this thread is selected again.

Wait method implementation: When a thread calls the wait() method and the requested object is locked by another thread, this thread releases the object locked by itself and records the requested object reference in corresponding field *WaitObject0~n-1* according to its priority and blocks itself. This static field will be checked when another thread calls the *notify*() method.

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 synchronized block which another thread with a lower priority is in, then the priority inheritance must be taken. In HRTEJ, a simple method to implement the priority inheritance is adopted. The scheduler checks the field *Exchange* of the thread owning the shared object, if the priority inheritance has been taken (*Exchange* != -1), the higher priority

will be assigned to this thread directly. Otherwise, the original priority of this thread is saved in its *Exchange* field, then, the higher priority is assigned to it. When this thread releases the locked object, it takes the original priority back again from *Exchange*.

As discussed above, special hardware is 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.

Asynchronous thread termination is processed in the same way with asynchronous transfer of control. Physical memory access is not supported in HRTEJ to conceal the details of memory allocation from Java programmers. The special hardware in HRTEJ for thread scheduling, synchronization, asynchronous transfer of control and memory management will be introduced in following sections.

3.2 Implementing ATC in the HRTEJ processor

1 ATC preprocessing by the CConverter

CConverter reads standard Java class file and converts all of the methods into a binary format. Attributes of each method are stored in fields with a determinate location.

CConverter processes the exception table of every method, and assigns correct values that HRTEJ can process directly to the items *extab_addr*, *extab_len*, and *extab_*item.

2 ATC triggered by target.interrupt() or target.aie.fire()

The ATC is triggered by invoking method *interrupt()* or *aie.fire()* in a thread.

cur_level, *cur_AIE* is variants defined in *interrupt()* or *aie.fire()*.

ATC related fields of each thread shown in Fig.4.

Pending: it shows there is an AIE in action.

AIE_ref: the reference of the received AIE.

AIE_level: the method invocation level of the received AIE.

```
if(target.pending == 0){                    //no AIE in pending
    target.pending = 1;          //target thread is marked pending
    target.AIE_ref = cur_AIE; //record the object's reference for generic or specific AIE thrown currently
}else{       //other AIE in pending, replacement rules must be taken
    if(cur_AIE == GenericAIE){//generic AIE has a higher priority
            target.AIE_ref = cur_AIE;
    }elseif(cur_AIE!=GenericAIE&&target.AIE_ref!=GenericAIE){
        if(target.AIE_level>cur_level){//record the AIE with higher priority.
            target.AIE_ref = cur_AIE;
        }
    }
}
```

3 ATC processed by the scheduler

ATC related registers in HRTEJ:
throwAIE: it denotes whether the current method has a throws AIE clause.
doInterruptible: denote if there are catch AIE or its superclass, finally clauses in this method.
Fst_catch: record the first reference of method context be used to process the AIE.
Locked: denoting if this method is a synchronized method or not.

When the target thread of the thrown AIE is scheduled again, the scheduler will process it based on three different conditions. **When** the target thread is in an AI-deferred section, the scheduler restores its context and executes it as a normal thread. **When** the target thread is in an AI section and it is the run() method of interface *doInterruptible*, the scheduler pops the stack frame of run() method and restores the context of method interruptAction() to handle the AIE. **When** the target thread is in an AI section and it is in other methods instead of run() method of interface *doInterruptible*, the scheduler will restore context from the register *fst_catch* to handle the AIE.

3.3 Memory management in the HRTEJ processor

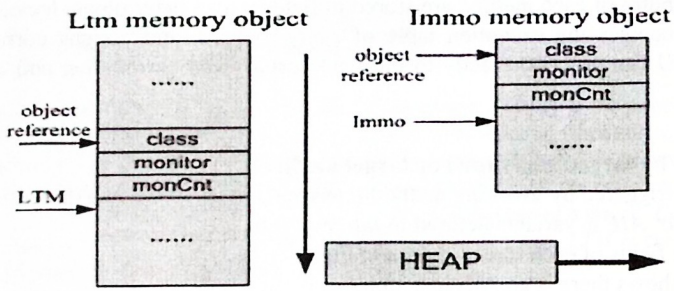

Fig.4. Memory management of the HRTEJ processor. *LTM* is current pointer to allocate space in the *LTMemory* of the running thread. *Immo* is current pointer to allocate space in the Immortal Memory. *HEAP* points to *LTM or IMMO* to concretely complete an allocation.

Memory management policy in the HRTEJ processor:

New LTMemory / ImmortalMemory:
class(classAddr)=>mem[LTM/Immo++];
monitor(0) => mem[LTM/Immo ++];
monCnt (0) => mem[LTM/Immo ++];
return the LTMemory/ImmortalMemory object reference(LTM/Immo-3);

LTMemory.enter• • //'1'denotes LTM; '0'denotes IMMO;
'1' => MEMType; LTM => HEAP;
LTMemory.exit()
this => LTM; //go back to last LTM scope

ImmortalMemory.enter()
'0' => MEMType; Immo => HEAP;

```
New GeneralObject:
        HEAP+ObjectSize => HEAP
        if(MEMType == '0')          HEAP => Immo;    //update Immo
        Else                        HEAP => LTM;     //update LTM
```

To avoid the interference of garbage collector and keep the efficiency about memory space, the LTMemory is provided in the HRTEJ processor. In standard RTSJ, the memory size must be specified by the programmer to use the LTMemory. It makes the programming model a little tricky to Java programmers. As described above, in the HRTEJ processor, an API *exit*() is offered to avoiding the programmer to specify the memory size for a LTMemory. This programming model is more maneuverable for the Java programmer. Moreover, without shared LTMemory between threads and based on the operation policy above, the memory management WCET of the HRTEJ processor is predictable.

4 Evaluation and discussion

The HRTEJ processor is implemented in experimental platform FD-MCES which provides a FPGA chip XC2S150-PQ208 and some debugging conveniences. Through the monitoring software FD-uniDbugger, the bytecode execution on top of HRTEJ can be traced single cycle. Due to the constraints of the experimental platform, the current version of HRTEJ is 16-bit and about 100 instructions implemented with several extended instructions. The memory used is 32Kx16 with 0.1us latency, so, read and write without cache by HRTEJ with 8M frequency can be completed in one cycle that simplifies the WCET analysis.

Table 1. Clock cycles of bytecode execution time

	HRTEJ	HRTEJ	JOP	JOP
iload	2	2+r	2	2
iadd	1	1	1	1
iinc	8	7+r	11	11
ldc	8	6+2*r	4	3+r
if_icmplt taken	10	9+r	6	6
if_icmplt not taken	10	9+r	6	6
getfield	7	5+2*r	12	10+2*r
putfield	7	5+w+r	15	13+r+w
getstatic	8	6+2*r	6	4+2*r
putstatic	8	6+w+r	7	5+r+w
iaload	2	2+r	21	19+2*r
invoke	43	34+9*r	82	78+4*r+b
invoke static	39	29+10*r	61	58+3*r+b
invoke interface	n/a	n/a	90	84+6*r+b
dup	2	2	1	1
new	12	12	Java	Java
iconst_x	2	2	1	1
aconst_null	2	2	1	1
astore_x	3	3	1	1
aload_x	3	3	1	1
return	20	20	14	13+r+b
ireturn	22	22	16	15+r+b
goto	5	4+r	4	4
bipush	4	4+r	2	2
pop	1	1	1	1
istore	2	2+r	2	2
istore_x	3	3	1	1

Table 1 shows some bytecode execution time in cycles implemented by the JOP and the HRTEJ processor. r denotes the time to complete a memory read and w denotes a memory write when the bytecode needs access the memory. b is the bytecode loading time when a cache miss happened in JOP. There is no instruction/method cache implemented in HRTEJ till now, so b is not used for denoting its bytecode execution cycles. The column 1 and 3 show the bytecode execution time assuming that r=w=1 and b=0. Obviously, the JOP with 100M frequency has a much higher performance than HRTEJ. Currently, the HRTEJ processor puts more emphases on the predictability than the performance. The performance will be considered carefully at the next step.

Estimating the WCET of tasks is essential for designing and verifying a real-time system. Because the measured execution times of tasks is sensitive to their inputs. As a rule, static analysis is a necessary method for hard real-time systems. Therefore, the WCET of the example shown in appendix is static analyzed to demonstrate the real-time performance of the HRTEJ processor.

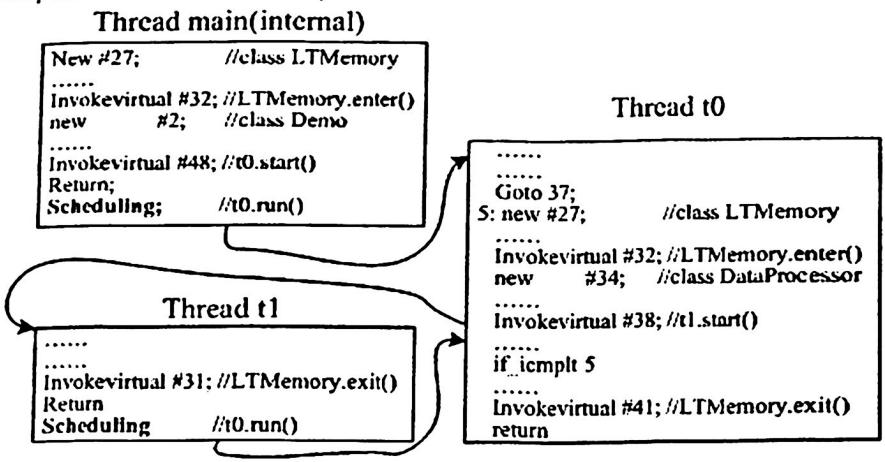

Fig.5. Thread scheduling of example *Demo.java* in appendix. The bytecode compiled from *Demo.java* can be mainly partitioned as 3 parts. In each part, the WCET of the general bytecode listed in Table 2 is predictable. For thread *t0*, there is a finite loop. Its WCET can be calculated as *100*WCET (general code + LTMemory + start() + Scheduling)*. Method *start()* sets the created thread into queue *Ready_T* and wait scheduling. *Scheduling* denotes the WCET of the scheduler execution when a thread scheduling happens. The WCET of the *LTMemory* operation is predictable as shown in Fig.4. Thus, just the WCET of the method *start()* and *Scheduling* are analyzed in detail below.

The process of method *t.start()* and its WCET:

Disable the interrupt;	(1	cycle)
Save the PC for current thread;	(1	cycle)
Put current thread into Ready_T;	(2	cycles)
Jump to the scheduler;	(1	cycle)

The process of *Scheduling* and its WCET:

Disable the interrupt;	(1	cycle)
Save the context of the preempted thread;	(17	cycles)
Move corresponding '1' from *Run_T* to *Ready_T*;	(2	cycles)
Move the leftmost '1' to the corresponding bit in *Run_T*;	(2	cycles)
Save the thread reference to *ThisThread* register;	(3	cycles)
Restore the context for the thread corresponding to the leftmost '1';	(16	cycles)

From discussed above, the WCET of the method start() and scheduling is also predicted. So, the real-time performance of the whole application can be guaranteed. Furthermore, in appendix, the efficiency of the LTMemory is illustrated clearly. The maximal allocation of the LTMemory space in this example is S(t0)+S(t1) instead of S(t0)+100*S(t1). S(t) denotes the space of thread t.

Another advantage learned from this example is that the programming model of the LTMemory is very simple. Java programmers just need creating and entering a LTMemory space to use it instead of denoting its memory size.

5 Conclusions

In order to employ Java programming language to improve development efficiency, security and robustness of real-time systems, many different real-time Java platforms were proposed. Due to high execution efficiency, low memory footprint and low power consumption, Java processor is preferable for embedded real-time systems. In this paper, a hard real-time Java processor based on the RTSJ is implemented. This processor provides some special supports for the mechanisms specified in the RTSJ such as ATC, scoped memory etc. Because most of the operations are preprocessed by the CConverter, this processor is simple to implement and suitable for low cost real-time systems.

References

1. 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.
2. G. Bollela, J. Gosling. B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Trunbull, "The Real-Time Specification for Java," Addison Wesley, vol. 1st edition, 2000.
3. H. Cai and A. Wellings, "Supporting mixed criticality applications in a ravenscar-java environment," ON THE MOVE TO MEANINGFUL INTERNET SYSTEMS 2004: OTM 2004 WORKSHOPS, PROCEEDINGS, vol. 3292, pp. 278-291, 2004.
4. "TimeSys JTime® 1.0 RTSJ Extensions Reference Pages," available at http://students.cs.tamu.edu/jtime_guide.pdf.
5. 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, 2005. ISORC 2005., 2005.

6. 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, 2005. ISORC 2005., 2005.
7. D. S. Hardin, "aJile Systems: Low-Power Direct-Execution JavaTM Microprocessors for Real-Time and Networked Embedded Applications," Available at http://www.ajile.com/Documents/aJile-white-paper.pdf.
8. S. A. Ito, L. Carro, and R. P. Jacobi, "Making Java work for microcontroller applications," Design & Test of Computers, IEEE, vol. 18, pp. 100 - 110, 2001.
9. M. Schoeberl., "JOP: A Java Optimized Processor for Embedded Real-Time Systems," Phd dissertation, http://www.jopdesign.com, 2005.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha, "The Java Language Specification Second Edition," 2000.
11. T. Lindholm and F. Yellin, "The Java Virtual Machine Specification, 2nd edition," Addison Wesley, 1999.

Appendix

Demo.java

```
import javax.realtime.*;
class DataProcessor extends NoHeapRealtimeThread{
        int data=0;
        public DataProcessor(int priority){
                super(priority);
        }
        public void run(){
                System.out.println("Processing in DataProcessor");
                Demo.ltm1.exit();
        }
}
public class Demo extends NoHeapRealtimeThread{
        public static LTMemory ltm = null;
        public static LTMemory ltm1 = null;
        public Demo(int priority){
                super(priority);
        }
        public void run(){
                for(int i=0; i<100; i++){
                        ltm1 = new LTMemory();
                        ltm1.enter();
                        DataProcessor t1 = new DataProcessor(3);
                        t1.start();
                } ltm.exit();
        }
        public static void main(String[] args){
                ltm = new LTMemory();
                ltm.enter();
                Demo t0 = new Demo(5);
                t0.start();
        }
}
```