Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 57575, 8 pages
doi:10.1155/2007/57575

Research Article

Java Processor Optimized for RTSJ

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

[1] *School of Information Technology, Southern Yangtze University, Wuxi 214122, China*
[2] *Department of Computer Science and Engineering, Fudan University, Shanghai 200433, China*

Received 18 January 2006; Revised 25 September 2006; Accepted 23 April 2007

Recommended by Zoran Salcic

Due to the preeminent work of the real-time specification for Java (RTSJ), Java is increasingly expected to become the leading programming language in real-time systems. To provide a Java platform suitable for real-time applications, a Java processor which can execute Java bytecode is directly proposed in this paper. It provides efficient support in hardware for some mechanisms specified in the RTSJ and offers a simpler programming model through ameliorating the scoped memory of the RTSJ. The worst case execution time (WCET) of the bytecodes implemented in this processor is predictable by employing the optimization method proposed in our previous work, in which all the processing interfering predictability is handled before bytecode execution. Further advantage of this method is to make the implementation of the processor simpler and suited to a low-cost FPGA chip.

Copyright © 2007 Zhilei Chai et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

Real-time specification for Java RTSJ [1] is a real-time extension for the Java language specification [2] and the Java virtual machine specification [3] under the requirements for real-time extensions for the Java platform [4]. It provides an application programming interface that enables the creation, execution, and management of Java threads with predictable temporal behavior. The RTSJ 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. It holds predictable execution as the first priority in all trade-offs. With the advantages as an object-oriented and concurrent programming language, and with the real-time performance guaranteed by the RTSJ, Java is increasingly expected to become the leading programming language in embedded real-time systems.

Currently, some Java platforms supporting RTSJ or its variants have been implemented. These platforms include RI.()() [5], JTime [6], RJVM [7], Mackinac [8], and OVM [9], to name only few. All of these Java platforms are implemented as an interpreter or an ahead-of-time compiler. Just-in-time compilation (even with the hotspot technique) during the execution consumes too much memory and leads to WCET unpredictability. So it is not suitable for embedded real-time systems. Comparing with these platforms, a Java processor can execute Java bytecode directly in silicon and provide special support in hardware, which makes it an appealing execution platform for Java in embedded systems. Now, some excellent Java processors for real-time and embedded systems were implemented, such as aJile [10], FemtoJava [11], and JOP [12]. Nevertheless, few of them provide special support for mechanisms of the RTSJ now. aJile systems announces the RTSJ will be supported on top of the aJ-80 and aJ-100 chips.

In this paper, we propose a Java processor optimized for RTSJ (called JPOR for short) suitable for embedded real-time systems. This processor provides special support in hardware for mechanisms of the RTSJ such as asynchronous transfer of control (ATC), thread, synchronization, and memory management, as well as offers a simpler programming model through ameliorating the scoped memory of the RTSJ. Because all the processing interfering instruction predictability is handled by the *CConverter* (a class loader and preprocessor we designed to do preprocessing and optimization according to optimization method in [13]) before bytecode execution on JPOR, the WCET of the bytecodes implemented in this processor is predictable. At the same time, some of the complex Java bytecodes (e.g., *new*) are simplified dramatically by the CConverter and it makes implementation of JPOR straightforward and suited to a low-cost FPGA chip.

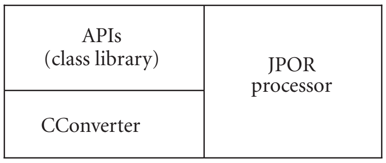

Figure 1: Java platform based on JPOR.

2. JPOR ARCHITECTURE OVERVIEW

2.1. The Java platform based on JPOR

A Java processor alone is not a complete Java platform. In this paper, the complete Java platform is composed of CConverter (class loader), APIs (class library), and JPOR processor (including execution engine, memory, and I/O), which is shown in Figure 1. The APIs provide a profile based on the RTSJ for Java application programmers. JPOR is our proposed processor to execute Java bytecode directly and provide support optimized for the RTSJ.

Similar to other real time Java platforms, the execution of Java applications on this platform is also divided into two phases: *initialization* phase (nonreal time) and *mission* phase (real time). During the initialization phase, all of the class files including application code and class library referred to by the Java application are loaded, verified, linked, and then transformed into a binary representation before being executed on JPOR. This transformation is performed by the CConverter instead of JPOR. During the mission phase, the binary representation is downloaded and executed on JPOR with predictable WCET. The CConverter does some optimizations to guarantee the real-time performance of the Java processor and simplify its implementation at the same time. The *new* instruction is taken for an example. In the conventional JVM, the *new* instruction is quite complicated, which requires searching and loading the class and superclasses of this new object dynamically while allocating memory space for it. Furthermore, the *new* instruction needs containing a loop bounded by the size of the object to initialize the new object with zero values. Hence, the implementation of the *new* instruction is too complicated and its WCET cannot be predicted in the conventional JVM.

To solve this problem, every object's size is calculated and recorded by the CConverter in advance. And some bytecodes are inserted behind the *new* instruction into the binary representation to initialize the new object with zero values.

The original *new* instruction:

//indexbyte (16-bit) is the entry for constant pool.

new, indexbyte1, indexbyte2;

Stack: before: ......;

after: objectref;

The *new* instruction executed by JPOR:

//objectsize (16-bit) is size of this new object

//Classaddr (16-bit) is the address for the class' //reference of this object.

new, objectsize, Classaddr;

//initialize the new object with zero values.

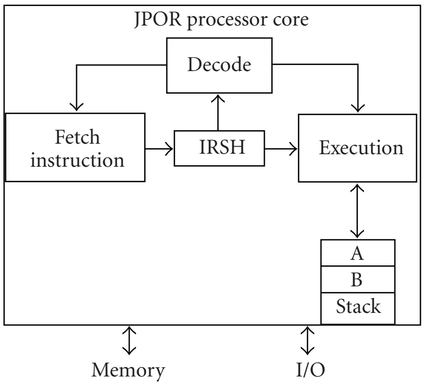

Figure 2: Architecture of JPOR processor.

Initialization bytecodes;

Stack: before: ...;

after: objectref;

As shown above, the *new* instruction in JPOR is only used to handle memory allocation. The initialization of the new object is processed by *initialization* bytecodes. This method makes *new* instruction simple and its WCET predictable. More details will be discussed together with memory management in Section 3.2. Other bytecodes are preprocessed and optimized by the CConverter similarly with *new* instruction.

2.2. Architecture of the JPOR processor

As shown in Figure 2, the JPOR processor core is simply divided into three pipeline stages: *fetch instruction*, *decode*, and *execution*.

Fetch instruction

In order to build a self-contained Java processor, direct access to the memory and I/O devices is necessary. However, there is no bytecode defined in Java instruction set for low-level access. Some extended instructions should be defined to solve this problem. In JPOR, the bytecodes from *0xcb* to *0xe4* that are used as *_quick* bytecodes in the conventional JVM are selected as extended instructions because *_quick* instructions are not used anymore in JPOR. Take *M2R(0xce), reg1, reg2*, for example, this extended instruction reads data from memory or I/O according to the address denoted by register *reg2*, and writes it into register *reg1*. The extended instructions in this way are in the uniform format with other bytecodes. Thus, the fetch unit can process them as a single instruction set conveniently. In order to reduce memory access frequency, a register IRSH (instruction register shifting to high 8-bit) having the same width with the memory interface (e.g., 16-bit) is used as an FIFO to fetch multiple instructions (e.g., 2 instructions) at a time. The fetched instruction, which is used in decode or execution stages, are located into register IRSH.

Decode

The decode unit of JPOR is implemented as a microprogramming model. With the IRSH shifting right 8-bit at a time, the decode unit always takes the highest 8-bit of IRSH as the entry to find the proper microcode.

Execution

JPOR is implemented as a stack-oriented machine to fit the JVM behavior. Since the stack is accessed frequently for *operands* and *locals*, it is placed into the same chip with the processor core. This stage performs ALU operations, load, store, and stack operations. Similar to [12], to avoid extra write-back stage and data forwarding, we use as the two topmost stack elements two explicit registers A and B providing operands for ALU. The execution unit can get operands from IRSH, stack, memory, and I/O.

Memory and I/O

The binary representation produced by the CConverter is downloaded into memory, whose layout is shown as Figure 3. The processor core can access I/O through interrupt or loop from a uniform addressing with the memory. Once power on, the JPOR processor starts to execute initial code and do system initialization according to the initial parameters. Then, it executes *main thread creating code* to create the main thread. Finally, the processor selects main thread and executes bytecode from there step by step. All of the string, static fields, and other data can be accessed by their addresses directly.

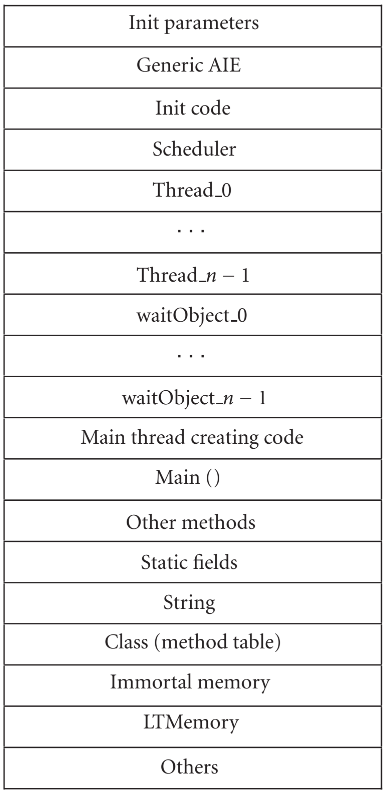

FIGURE 3: Memory layout of JPOR.

3. RTSJ-SPECIFIC SUPPORT IN JPOR

As mentioned in Section 1, the RTSJ contains some enhanced areas, among which the asynchronous event is handled by software but not extra hardware, and asynchronous thread termination is processed in the same way with asynchronous transfer of control. To conceal the details of memory allocation from Java programmers, physical memory access is not supported in JPOR. The special hardwares in JPOR for thread scheduling, synchronization, asynchronous transfer of control, and memory management will be introduced in the following sections.

3.1. Thread, scheduling, and synchronization in JPOR

The RTSJ defines two subclasses *RealtimeThread* and *NoHeapRealtimeThread* of *Java.lang.Thread* that own more precise scheduling semantics. The *NoHeapRealtimeThread* extending *RealtimeThread* is not allowed to allocate or even reference objects from the heap, and can safely execute in preference to the garbage collector.

The base scheduler required by the RTSJ is fixed-priority preemptive with 28 unique priority levels. The *PriorityInheritance* protocol is the default monitor control policy in the RTSJ, defined as follows: a thread with higher priority entering the monitor will boost the effective priority of another thread in the monitor to level of its own effective priority. When that thread exits the monitor, its effective priority will be restored to its previous value.

In JPOR, a fixed-priority preemptive scheduler is implemented. To schedule thread efficiently and predictably, some state registers are provided, namely *Run_T*, *Ready_T*, *Block_T*, and *Dead_T*. These registers are 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's state is changed by marking corresponding bit of that state register to "1" according to this thread's priority. For example, the 10th bit of the register Ready_T is set to "1" denoting the thread with a priority value 10 is ready now. Register *BitMap* is used to translate the thread's priority into corresponding bit "1" in a register. *Offset* is used to translate the corresponding bit "1" of a register into the thread's priority value. *STK_base* $0 \sim n-1$ is the base address of the stack for each thread. *LTMAddr* $0 \sim n-1$ is the base address of the *LTMemory* space for each thread.

The maximum thread number supported in the JPOR processor is n (n is the width of the data path). Every thread can be assigned a unique priority from 0 to $n-1$, and any two of them cannot be assigned to the same priority. 0 is the highest priority. These threads can be created and terminated dynamically during execution. Creating a new thread is just like creating a general object, but the reference of this thread should be kept into the corresponding static field *Thread* $0 \sim n-1$ shown in Figure 3 according to this thread's

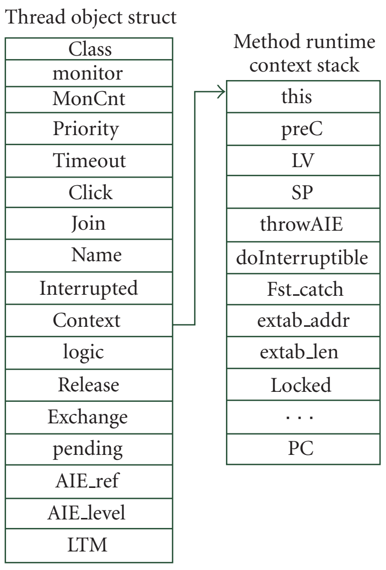

Figure 4: Thread object of JPOR.

priority. The scheduler can terminate a thread through moving the corresponding "1" from other queues to the *Dead_T* according to its priority. The scheduler always chooses the thread corresponding to the leftmost "1" in *Ready_T* queue to dispatch and execute it.

The thread object and its corresponding context are shown in Figure 4. The context of a preempted thread is pushed into its own stack (on the same chip with the processor) when a scheduling occurs, and the pointer of the stack is kept in the field *Context* of the thread object. The context can be restored from the *Context* pointer when this thread is selected again.

wait() method implementation: when a thread calls the *wait()* method and the object being requested is locked by another thread, it releases the object already locked by itself. Then, it records the reference of the object being requested in corresponding static field *WaitObject* $0 \sim n-1$ according to its priority and blocks itself. This static field will be checked to decide whether the thread is waiting for a released object when another thread calls the *notify()* method.

notify() method implementation: when a thread calls the *notify()* method, it checks the object reference recorded in *WaitObject* $0 \sim n-1$. If this reference is equal to the object reference released by current thread, then, the thread specified by this *WaitObject* is notified and put into *Ready_T* queue.

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 where another thread with a lower priority locates, then the priority inheritance must be taken. In JPOR, 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*.

3.2. Asynchronous transfer of control in JPOR

Asynchronous transfer of control is a crucial mechanism for real-time applications that enables one thread to throw an exception into another. It is useful for several purposes such as timeout expressing, thread termination, and so on. Some *ATC-related* terms are described as follows.

AIE: *Javax.realtime.AsynchronouslyInterruptedExcept-ion* is a subclass of *Java.Lang.InterruptedEx- ception*. ATC is triggered by throwing an AIE into another thread.

AI-method: a method is Asynchronously Interruptible if it includes AIE in its throws clause.

ATC-deferred section: a synchronized method, a synchronized statement, or any methods without AIE in its throws clause.

Because the ATC mechanism needs processing operation rules, propagation rules, replacement rules, and so on, processing these rules during execution will impede the predictability of the WCET. To solve this problem, some special supports are provided in JPOR and the ATC is processed from three aspects as follows.

(1) ATC preprocessing by the CConverte

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 determined location.

CConverter processes the exception table of every method, and assigns correct values that JPOR 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* are variants defined in method interrupt() or aie.fire().

ATC related fields of each thread are shown in Figure 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.

When an ATC is triggered, if there is no AIE in pending, the target thread is marked pending and the object's reference of the AIE thrown currently is kept into the target thread's field *AIE_ref*. Else, replacement rules must be taken. The AIE with higher priority (generic AIE has a higher priority than specific AIE) will be recorded into the target thread's field AIE_ref. If the ATC is triggered by method interrupt(), the target thread's field interrupted is marked for compatibility.

(3) ATC processing by the scheduler

ATC related registers in JPOR:

throwAIE: it denotes whether the current method has a throws AIE clause.

doInterruptible: it denotes whether catch AIE (or its superclass) or finally clauses exist in this method.

Fst_catch: always records the first reference of method context 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.

With many processing completed before execution and with the special hardware designed for ATC (e.g., Fst_catch always records the reference of method used to process the AIE), the ATC process in JPOR is predictable.

3.3. Memory management in JPOR

The unpredictability caused by the interference of garbage collector is intolerable for the real-time systems. The RTSJ proposes two kinds of memory classes *ScopedMemory* and *ImmortalMemory* to allow the definition of memory regions outside of the traditional Java heap. ImmortalMemory is a memory resource that is shared among all threads. Objects allocated in the immortal memory live until the end of the application. ScopedMemory is the abstract base class of all classes dealing with representations of memory regions with a limited lifetime and its reclamation is predictable. The ScopedMemory area is valid as long as there are real-time threads with access to it. ScopedMemory has four subclasses defined in the RTSJ. Considering the factor of operation predictability and program portability, only *LTMemory* is selected to be used in JPOR.

Some goals of the LTMemory in JPOR

(i) In order to check the object assignment rules in advance by the CConverter to guarantee the predictability of WCET, the LTMemory in JPOR cannot be shared with multiple threads.

(ii) To improve the efficiency of memory space, the LTMemory in JPOR can be nested.

(iii) It offers simpler API in JPOR to simplify the programming model.

Related registers are used in Figure 5:

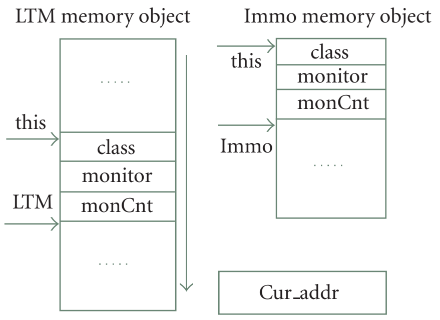

Figure 5: Memory management of the JPOR.

LTM: current pointer to allocate space in the LTMemory associated with the running thread. Each thread has its own LTMemory space and its LTM value for allocating. Because there will be several threads creating and using several scopes at the same time, the separate LTMemory space for per thread is used to avoid memory collision.

Immo: current pointer to allocate space in the shared immortal memory.

Cur_addr: pointing to LTM or IMMO to concretely make an allocation.

As described in Section 2.1, the *new* instruction preprocessed by the CConverter has a format like *new, objectsize, Classaddr*. *objectsize* is used to denote the type of the object to be created (objectsize = −1, LTMemory object; objectsize = −2, ImmortalMemory object; others, general object and its size) and the object size. JPOR can create proper object according to the *new* instruction.

```
//create a memory space
New LTMemory/ImmortalMemory:
//save some common fields for this object
//class address of this object
class(class address)
=>mem[LTM++/Immo++];
//thread locking this object
monitor(0) => mem[LTM++/Immo ++];
//locked count
monCnt (0) => mem[LTM++/Immo ++];
//return the reference of this object for use later
//LTMemory/ImmortalMemory object reference
return this (LTM-3/Immo-3);
//create a general object
New general object:
//save some common fields for this object
//class address of this object
class(class address)=>mem[Cur_addr +1];
//thread locking this object
monitor(0) => mem[Cur_addr + 2];
//locked count
monCnt (0) => mem[Cur_addr + 3];
//allocate space for the object
Cur_addr +objectsize => Cur_addr;
if(MEMType == "0")
//update Immo if allocating in the Immortal memory
```

```
Cur_addr => Immo;
Else
//update LTM if allocating in the LTMemory
Cur_addr => LTM;
//return the object reference
Return this (Cur_addr - objectsize);
```

As shown above, the general *new* instruction in JPOR is only used to process memory allocation. The initialization of the new object is processed by *initialization* bytecodes described in Section 2.1. Thus, the *new* instruction is simple and its WCET is predictable.

```
//"1"denotes LTM; "0"denotes Immo;
LTMemory.enter()
//current memory space is LTMemory
"1" => MEMType;
//current LTMemory address to be allocated of the running thread
LTM => Cur_addr;
LTMemory.exit()
//go back to last LTMemory scope
this => LTM;
ImmortalMemory.enter()
//current memory space is immortal memory
"0" => MEMType;
//current immortal memory address to be allocated
Immo => Cur_addr;
```

Because the immortal memory exists for ever, the *ImmortalMemory.exit()* is not needed for JPOR.

In standard RTSJ, the memory size must be specified by the programmer to create an LTMemory object, such as *LTMemory(long initialSizeInBytes, long maxSizeInBytes)* or *LTMemory(SizeEstimator initial, SizeEstimator maximum)*, and so forth. It makes the programming model a little tricky to Java programmers. Another disadvantage of the memory object with fixed size is that there will be many memory fragments existing. As described above, an API *exit()* is provided in JPOR to avoid the programmer to specify the memory size for an LTMemory and avoid memory fragments occurring. This programming model is more maneuverable for a Java programmer. Moreover, without shared LTMemory between threads and based on the operation policy above, the WCET of memory management in JPOR is predictable.

4. EVALUATION AND DISCUSSION

The JPOR processor is implemented in experimental platform FD-MCES (the computer architecture experimental platform designed by Fudan university), which provides an FPGA chip XC2S150-PQ208 and some debugging conveniences. Through the monitoring software FD-uniDbugger designed by our laboratory, the bytecode execution on top of JPOR can be traced single cycle. Due to the constraints of the experimental platform, the current version of JPOR is 16-bit and about 100 instructions implemented including several extended instructions with the resource usage 1933 LCs + 2 KB RAM. The memory provided by FD-MCES is 32 Kx16 with 0.1 microseconds latency, so, read and write without cache by JPOR with 8 MHz frequency can be completed in one cycle that simplifies the WCET analysis.

Table 1: Clock cycles of bytecode execution time.

	JPOR	JPOR	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 JPOR 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 bytecodes loading time when a cache miss happened in JOP. There is no instruction/method cache implemented in JPOR till now, so b is not used for denoting its bytecode execution cycles. The columns 1 and 3 show the bytecode execution time assuming that $r = w = 1$ and $b = 0$. Obviously, the JOP with 100 MHz frequency has a much higher performance than JPOR. Currently, the JPOR processor puts more emphases on the predictability and optimization for the RTSJ than the performance. The performance will be considered carefully at the next step.

```
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();
        }
}
```

FIGURE 6: An example designed with the APIs of the JPOR processor.

In JPOR, some complex instructions in conventional JVM are simplified. Take iaload as an example, to reduce the complexity and guarantee the predictability of its WCET, the null and bound checking are processed by the CConverter and the programmer. Then, this instruction is implemented as follows, its WCET is predictable.

arrayref(B) + index(A) => ADDR;
mem[ADDR] => A; stack[SP] => B;
(SP)−1 => SP;

Estimating the WCET of tasks is essential for designing and verifying a real-time system. In general, static analysis is a necessary method for hard real-time systems. Therefore, the WCET of the simple example shown in Figure 6 is static analyzed to demonstrate the real-time performance of the JPOR processor. There are 3 threads in this example namely main(0), $t1(3)$, $t0(5)$. The smaller priority value denotes a higher priority.

Because many hardware features such as data forwarding, branch prediction and data/instruction cache are not implemented in JPOR, the global low-level analysis can be omitted. The bytecodes compiled from Demo.java can be mainly partitioned as 3 parts. In each part, the WCET of the general bytecode is listed in Table 1. For thread t0, there is a finite loop. Its WCET can be calculated as *100* WCET (general codes + 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 discussed in Section 3.3. Thus, just the WCET of the method start() and scheduling are described 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 this thread register; (3 cycles)
Restore the context for the thread corresponding to the left most "1;" (16 cycles)

From the described 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, from Figure 6, the efficiency of the nested 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 allocated for thread t. It is notable that although some hardware is implemented in JPOR to prevent the memory collapse, the programmer should also take some measures to avoid a program collapse, such as do not exceed maximum thread number, do not assign the same priority to more than one thread, and so on.

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

5. CONCLUSIONS

In this paper, a real-time Java processor optimized for RTSJ is implemented. This processor provides efficient supports for the mechanisms specified in the RTSJ such as thread management, synchronization, ATC, scoped memory, and so on. Because most of the operations are completed by the preprocessor CConverter, this processor is simple to implement with a predictable bytecode execution time. Presently, the JPOR processor puts more emphases on the predictability than the performance. It will be considered carefully at the next step.

REFERENCES

[1] G. Bollela, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Trunbull, *The Real-Time Specification for Java*, Addison Wesley, Reading, Mass, USA, 1st edition, 2000.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, Addison-Wesley, Boston, Mass, USA, 2nd edition, 2000.

[3] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Boston, Mass, USA, 2nd edition, 1999.

[4] L. Carnahan and M. Ruark, "Requirements for Real-time Extensions for the Java^{TM} Platform," September 1999, http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf.

[5] "Java Reference Implementation (RI) and Technology Compatibility Kit (TCK)," http://www.timesys.com/java/.

[6] G. Bollella, K. Loh, G. McKendry, and T. Wozenilek, "Experiences and Benchmarking with JTime," in *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)*, vol. 2889 of *Lecture Notes in Computer Science*, pp. 534–549, Catania, Sicily, Italy, November 2003.

[7] H. Cai and A. J. Wellings, "Towards a high integrity real-time Java virtual machine," in *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '03)*, vol. 2889 of *Lecture Notes in Computer Science*, pp. 319–334, Catania, Sicily, Italy, November 2003.

[8] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain, "Mackinac: making hotspot^{TM} real-time," in *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, pp. 45–54, Seattle, Mass, USA, May 2005.

[9] http://www.cs.purdue.edu/homes/jv/soft/ovm/documents.htm.

[10] D. S. Hardin, "aJile Systems: Low-Power Direct-Execution Java^{TM} Microprocessors for Real-Time and Networked Embedded Applications," http://www.jempower.com/ajile/downloads/aJile-white-paper.pdf.

[11] S. A. Ito, L. Carro, and R. P. Jacobi, "Making Java work for microcontroller applications," *IEEE Design & Test of Computers*, vol. 18, no. 5, pp. 100–110, 2001.

[12] M. Schoeberl, "JOP: a Java optimized processor for embedded real-time systems," Phd dissertation, Vienna University of Technology, Vienna, Austria, 2005, http://www.jopdesign.com/.

[13] Z. Chai, Z. Q. Tang, L. M. Wang, and S. Tu, "An effective instruction optimization method for embedded real-time Java processor," in *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW '05)*, pp. 225–231, Oslo, Norway, June 2005.