A Time Predictable Java Processor

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Abstract

This paper presents a Java processor, called JOP, designed for time-predictable execution of real-time tasks. JOP is the implementation of the Java virtual machine in hardware. We propose a processor architecture that favors low worst-case execution time (WCET) over average case performance. The resulting processor is an easy target for the low-level WCET analysis.

1 Introduction

This paper introduces the concept of a Java processor [9] for embedded real-time systems, in particular the design of a small processor for resource-constrained devices with time-predictable execution of Java programs. This Java processor is called JOP – which stands for Java Optimized Processor –, based on the assumption that a full native implementation of all Java bytecode instructions [3] is not a useful approach.

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. Static WCET analysis is necessary for hard real-time systems. In order to obtain a low WCET value, a good processor model is necessary. Traditionally, only simple processors can be analyzed using practical WCET boundaries. Architectural advancements in modern processor designs tend to abide by the rule: '*Make the average case as fast as possible*'. This is orthogonal to '*Minimize the worst case*' and has the effect of complicating WCET analysis. This paper tackles this problem from the architectural perspective – by introducing a processor architecture in which simpler and more accurate WCET analysis is more important than average case performance.

In the following section a brief overview of the architecture of JOP is given, followed by a more detailed description of the microcode. In Section 4 we will show that the execution time of Java bytecodes can be exactly predicted in terms of the number of clock cycles.

Comparison between JOP and a number of different solutions for embedded Java with respect to the resource usage and general performance can be found in [8]. This paper does not deal with the issues of Java for real-time applications (see [1] and [5]) or real-time garbage collection.

2 JOP Architecture

JOP is a stack computer with its own instruction set, called microcode in this paper. Java bytecodes are translated into microcode instructions or sequences of microcode. The difference between the Java virtual machine (JVM) and JOP is best described as the following:

> The JVM is a CISC stack architecture, whereas JOP is a RISC stack architecture.

Figure 1 shows JOP's major function units. A typical configuration of JOP contains the processor core, a memory interface and a number of IO devices. The module extension provides the link between the processor core, and the memory and IO modules.

The processor core contains the three microcode pipeline stages *microcode fetch*, *decode* and *execute* and an additional translation stage *bytecode fetch*. The ports to the other modules are the address and data bus for the bytecode instructions, the two top elements of the stack (A and B), input to the top-of-stack (Data) and a number of control signals. There is no direct connection between the processor core and the external world.

The memory interface provides a connection between the main memory and the processor core. It also contains the bytecode cache. The extension module controls data read and write. The *busy* signal is used by the microcode instruction `wait` to synchronize the processor core with the memory unit. The core reads bytecode instructions through dedicated buses (BC address and BC data) from the memory subsystem. The request for a method to be placed in the cache is performed through the extension module, but the cache hit detection and load is performed by the memory interface independently of the processor core (and therefore concurrently).

2.1 The Processor Pipeline

JOP is a pipelined architecture with single cycle execution of microcode instructions and a novel approach to mapping Java bytecode to these instructions. Three stages form the JOP core pipeline, executing microcode instructions. An additional stage in the front of the core pipeline fetches Java bytecodes – the instructions of the JVM – and translates these bytecodes into addresses in microcode. Bytecode branches

3-9810801-0-6/DATE06 © 2006 EDAA

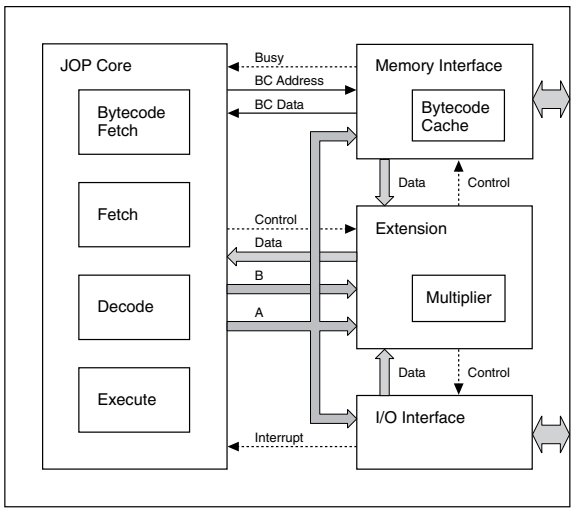

Figure 1. Block diagram of JOP

are also decoded and executed in this stage. The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches. Besides the usual decode function, the third pipeline stage also generates addresses for the stack RAM. The last pipeline stage performs ALU operations, load, store, and stack spill or fill. At the execution stage, operations are performed with the two topmost elements of the stack. A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill needs neither an extra write-back stage nor any data forwarding. Details of this two-level stack architecture are described in [7]. The short pipeline results in short branch delays. Therefore, a hard to analyze (with respect to WCET) branch prediction logic can be avoided.

2.2 Cache

A pipelined processor architecture calls for a high memory bandwidth. A standard technique to avoid processing bottlenecks due to the higher memory bandwidth is caching. However, standard cache organizations improve the average execution time but are difficult to predict for WCET analysis. Two time-predictable caches are proposed for JOP: a *stack cache* as a substitution for the data cache and a *method cache* to cache the instructions. As the stack is a heavily accessed memory region, the stack – or part of it – is placed in on-chip memory. This part of the stack is referred to as the *stack cache* and described in [7]. Fill and spill of the stack cache is subjected to microcode control and therefore time-predictable. In [6], a novel way to organize an instruction cache, as *method cache*, is given. The cache stores complete methods, and cache misses only occur on method invocation and return. Cache block replacement depends on the call tree, instead of instruction addresses. This *method cache* is easy to analyze with respect to worst-case behavior and still provides substantial performance gain.

3 Microcode

The following discussion concerns two different instruction sets: *bytecode* and *microcode*. Bytecodes are the instructions that make up a compiled Java program. These instructions are executed by a Java virtual machine (JVM). The JVM does not assume any particular implementation technology. Microcode is the native instruction set for JOP. Bytecodes are translated, during their execution, into JOP microcode. Both instruction sets are for a stack machine.

3.1 Translation of Bytecodes to Microcode

To date, no hardware implementation of the JVM exists that is capable of executing *all* bytecodes in hardware alone. This is due to the following: some bytecodes, such as `new`, which creates and initializes a new object, are too complex to implement in hardware. These bytecodes have to be emulated by software.

To build a self-contained JVM without an underlying operating system, direct access to the memory and I/O devices is necessary. There are no bytecodes defined for low-level access. These low-level services are usually implemented in *native* functions, which mean that another language (e.g. C) is native to the processor. However, for a Java processor, bytecode is the *native* language.

One way to solve this problem is to implement simple bytecodes in hardware and to emulate the more complex and *native* functions in software with a different instruction set (sometimes called microcode). However, a processor with two different instruction sets results in a complex design. Another common solution, used in Sun's picoJava [10], is to execute a subset of the bytecode native and to use a software trap to execute the remainder. This solution entails an overhead (min. 16 cycles in picoJava) for this software trap.

In JOP, this problem is solved in a much simpler way. JOP has a single *native* instruction set, the so-called microcode. During execution, every Java bytecode is translated to either one, or a sequence of microcode instructions. This translation merely adds one pipeline stage to the core processor and results in no execution overheads. With this solution, we are free to define the JOP instruction set to map smoothly to the stack architecture of the JVM, and to find an instruction coding that can be implemented with minimal hardware. The implementation of Java bytecodes by a sequence of single cycle microcode instructions also simplifies the calculation of the WCET for individual bytecodes (see Section 4).

3.2 Compact Microcode

For the JVM to be implemented efficiently, the microcode has to *fit* to the Java bytecode. Since the JVM is a stack machine, the microcode is also stack-oriented. However, the JVM is not a pure stack machine. Method parameters and local variables are defined as *locals*. These locals can reside

```
dup:    dup nxt    // 1 to 1 mapping

//  a and b are scratch variables at
//  the microcode level.
dup_x1: stm a      // save TOS
        stm b      // and TOS-1
        ldm a      // duplicate former TOS
        ldm b      // restore TOS-1
        ldm a nxt  // restore TOS and fetch
                   // the next bytecode
```

Figure 2. Implementation of `dup` and `dup_x1`

in a stack frame of the method and are accessed with an offset relative to the start of this *locals* area. Additional local variables (16) are available at the microcode level. These variables serve as scratch variables, like registers in a conventional CPU. However, arithmetic and logic operations are performed on the stack.

Some bytecodes, such as ALU operations and the short form access to *locals*, are directly implemented by an equivalent microcode instruction (with a different encoding). Additional instructions are available to access internal registers, main memory and I/O devices. A relative conditional branch (zero/non zero of TOS) performs control flow decisions at the microcode level. For optimum use of the available memory resources, all instructions are 8 bits long. There are no variable-length instructions and every instruction, with the exception of `wait`, is executed in a single cycle.

The example in Figure 2 shows the implementation of a single cycle bytecode and an infrequent bytecode as a sequence of JOP instructions. In this example, the `dup` bytecode is mapped to the equivalent `dup` microcode and executed in a single cycle, whereas `dup_x1` takes five cycles to execute, and after the last instruction (`ldm a nxt`), the first instruction for the next bytecode is executed.

3.3 Flexible Implementation of Bytecodes

As mentioned above, some Java bytecodes are very complex. One solution already described is to emulate them through a sequence of microcode instructions. However, some of the more complex bytecodes are very seldom used. To further reduce the resource implications for JOP, in this case local memory, bytecodes can even be implemented by *using* Java bytecodes. During the assembly of the microcoded JVM, all labels that represent an entry point for the bytecode implementation are used to generate the translation table. For all bytecodes for which no such label is found, i.e. there is no implementation in microcode, a *not-implemented* address is generated. The instruction sequence at this address invokes a static method from a system class. This class contains 256 static methods, one for each possible bytecode, ordered by the bytecode value. The bytecode is used as the index in the method table of this system class. This feature also allows to trade resource usage against performance.

4 Worst-Case Execution Time

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. WCET estimates can be obtained either by measurement or static analysis. The problem with using measurements is that the execution times of tasks tend to be sensitive to their inputs. As a rule, measurement does not guarantee safe WCET estimates. Instead, static analysis is necessary for hard real-time systems. Static analysis is usually divided into a number of different phases:

Path analysis generates the control flow graph (a directed graph of basic blocks) of the program and annotates (manual or automatic) loops with bounds.

Low-level analysis determines the execution time of basic blocks obtained by the path analysis. A model of the processor and the pipeline provides the execution time for the instruction sequence.

Global low-level analysis determines the influence of hardware features such as caches on program execution time. This analysis can use information from the path analysis to provide less pessimistic values.

WCET Calculation collapses the control flow graph to provide the final WCET estimate. Alternative paths in the graph are collapsed to a single value (the largest of the alternatives) and loops are collapsed once the loop bound is known.

For the low-level analysis, a good timing model of the processor is needed. The main problem for the low-level analysis is the execution time dependency of instructions in modern processors that are not designed for real-time systems. JOP is designed to be an easy target for WCET analysis. The WCET of each bytecode can be predicted in terms of number of cycles it requires. There are no dependencies between bytecodes.

Each bytecode is implemented by microcode. We can obtain the WCET of a single bytecode by performing WCET analysis at the microcode level. To prove that there are no time dependencies between bytecodes, we have to show that no processor states are *shared* between different bytecodes.

4.1 Microcode Path Analysis

To obtain the WCET values for the individual bytecodes we perform the path analysis at the microcode level. First, we have to ensure that a number of restrictions (from [4]) of the code are fulfilled:

- Programs must not contain unbounded recursion. This property is satisfied by the fact that there exists no call instruction in microcode.
- Function pointers and computed `gotos` complicate the path analysis and should therefore be avoided. Only simple conditional branches are available at the microcode level.

- The upper bound of each loop has to be known. This is the only point that has to be verified by inspection of the microcode.

To detect loops in the microcode we have to find all backward branches (e.g. with a negative branch offset). The branch offsets can be found in a VHDL file (`offtbl.vhd`) that is generated during microcode assembly. In the current implementation of the JVM there are ten different negative offsets. However, not each offset represents a loop. Most of these branches are used to share common code. All backward branches found in `jvm.asm` are summarized below:

- Three branches are found in the initialization code of the JVM. They are not part of a bytecode implementation and can be ignored.
- Five branches are used by exceptions, the interrupt bytecode, and for the call of Java implemented bytecodes. The target of these branches is found in the implementation of `invoke` to share part of the microcode sequence. These branches are therefore not part of a loop.
- One branch is found in the implementation of `imul` to perform a fixed delay. The iteration count for this loop is constant.
- Two backward branches share the same offset and are used in loops to move data between the stack memory and main memory. This loop is not part of a regular bytecode. It is contained in a system function used by the scheduler for the task switch. The bound for this loop has to be determined in the scheduler code.

A few bytecodes are implemented in Java. The implementation can be found in the class `com.jopdesign.sys.JVM` and can be analyzed in the same way as application code. The bytecodes `idiv` and `irem` contain a constant loop. The bytecodes `new` and `anewarray` contain loops to initialize (with zero values) new objects or arrays. The loop is bound by the size of the object or array. The bytecode `lookupswitch`[1] performs a linear search through a table of branch offsets. The WCET depends on the table size that can be found as part of the instruction.

As the microcode sequences are very short, the calculation of the control flow graph for each bytecode is done manually.

4.2 Microcode Low-level Analysis

To calculate the execution time of basic blocks in the microcode, we need to establish the timing of microcode instructions on JOP. All microcode instructions except `wait` execute in a single cycle, reducing the low-level analysis to a case of merely counting the instructions.

The `wait` instruction is used to stall the processor and wait for the memory subsystem to finish a memory transaction. The execution time of the `wait` instruction depends on the memory system and, if the memory system is predictable, has a known WCET. A main memory consisting of SRAM chips can provide this predictability and this solution is therefore advised. The predictable handling of DMA, which is used for the instruction cache fill, is explained in [6]. The `wait` instruction is the only way to stall the processor.

Microcode is stored in on-chip memory with single cycle access. Each microcode instruction is a single word long and there is no need for either caching or prefetching at this stage. We can therefore omit performing a low-level analysis. No pipeline analysis [2], with its possible unbound timing effects, is necessary.

4.3 Bytecode Independency

We have seen that all microcode instructions except `wait` take one cycle to execute and are therefore independent of other instructions. This property directly translates to independency of bytecode instructions.

The `wait` microcode instruction provides a convenient way to hide memory access time. A memory read or write can be triggered in microcode and the processor can continue with microcode instructions. When the data from a memory read is needed, the processor explicitly waits until it becomes available.

For a memory store, this wait can be deferred until the memory system is used next. It is possible to initiate the store in a bytecode such as `putfield` and continue with the execution of the next bytecode, even when the store has not been completed. In this case, we introduce a dependency over bytecode boundaries, as the state of the memory system is *shared*. To avoid these dependencies that are difficult to analyze, each bytecode that accesses memory waits (preferably at the end of the microcode sequence) for the memory system.

Furthermore, the deferring of `wait` in a store operation results in an additional `wait` in every read operation. Since read operations are more frequent than write operations (15% vs. 2.5%, see [9]), the performance gain from the hidden memory store is lost.

4.4 WCET of Bytecodes

The control flow of the individual bytecodes together with the basic block length (that directly corresponds with the execution time) and the time for memory access result in the WCET (and BCET) values of the bytecodes. The exact values for each bytecode can be found in [9].

[1] `lookupswitch` is one way of implementing the Java `switch` statement. The other bytecode, `tableswitch`, uses an index in the table of branch offsets and has therefore a constant execution time.

```
final static int N = 5;

static void sort(int[] a) {
    int i, j, v1, v2;
    // loop count = N-1
    for (i=N-1; i>0; --i) {
        // loop count = (N-1)*N/2
        for (j=1; j<=i; ++j) {
            v1 = a[j-1];
            v2 = a[j];
            if (v1 > v2) {
                a[j] = v1;
                a[j-1] = v2;
            }
        }
    }
}
```

Figure 3. Bubble Sort test program for the WCET analysis

4.5 Evaluation

We conclude this section with a worst and best case analysis of a classic example, the Bubble Sort algorithm. The values calculated are compared with the measurements of the execution time on JOP on all permutations of the input data. Figure 3 shows the test program in Java. The algorithm contains two nested loops and one condition. We use an array of five elements to perform the measurements for all permutations (i.e. $5! = 120$) of the input data. The number of iterations of the outer loop is one less than the array size: $c_1 = N - 1$, in this case four. The inner loop is executed $c_2 = \sum_{i=1}^{c_1} i = c_1(c_1+1)/2$ times, i.e. ten times in our example.

The annotated control flow graph (CFG) of the example is shown in Figure 4. The edges contain labels showing how often the path between two nodes is taken. We can identify the outer loop, containing the blocks B2, B3, B4 and B8. The inner loop consists of blocks B4, B5, B6 and B7. Block B6 is executed when the condition of the `if` statement is true. The path from B5 to B7 is the only path that depends on the input data.

The compiled version, i.e. the bytecodes of the test program, split into basic blocks, is given in Table 1. The fourth column contains the execution time of the bytecodes. In Table 2 the basic blocks with their execution time in clock cycles and the worst and best case execution frequency is given. The values in the third and fifth columns (Count) of Table 2 are derived from the CFG and show how often the basic blocks are executed in the worst and best cases. The WCET and BCET value for each block is calculated by multiplying the clock cycles by the execution frequency. The overall WCET and BCET values are calculated by summing the values of the individual blocks B1 to B8. The last block (B9) is omitted, as the measurement does not contain the return statement.

The execution time of the program is measured using the

Table 1. Bytecode listing of the Bubble Sort with basic blocks

Block	Addr.	Bytecode	Cycles
B1	0:	iconst_4	1
	1:	istore_1	1
B2	2:	iload_1	1
	3:	ifle 53	4
B3	6:	iconst_1	1
	7:	istore_2	1
B4	8:	iload_2	1
	9:	iload_1	1
	10:	if_icmpgt 47	4
B5	13:	aload_0	1
	14:	iload_2	1
	15:	iconst_1	1
	16:	isub	1
	17:	iaload	29
	18:	istore_3	1
	19:	aload_0	1
	20:	iload_2	1
	21:	iaload	29
	22:	istore 4	2
	24:	iload_3	1
	25:	iload 4	2
	27:	if_icmple 41	4
B6	30:	aload_0	1
	31:	iload_2	1
	32:	iload_3	1
	33:	iastore	32
	34:	aload_0	1
	35:	iload_2	1
	36:	iconst_1	1
	37:	isub	1
	38:	iload 4	2
	40:	iastore	32
B7	41:	iinc 2, 1	11
	44:	goto 8	4
B8	47:	iinc 1, -1	11
	50:	goto 2	4
B9	53:	return	

Table 2. WCET and BCET in clock cycles of the basic blocks

		WCET		BCET	
Block	Cycles	Count	Total	Count	Total
B1	2	1	2	1	2
B2	5	5	25	5	25
B3	2	4	8	4	8
B4	6	14	84	14	84
B5	74	10	740	10	740
B6	73	10	730	0	0
B7	15	10	150	10	150
B8	15	4	60	4	60
B9		1		1	
Execution time calculated			1,799		1,069
Execution time measured			1,799		1,069

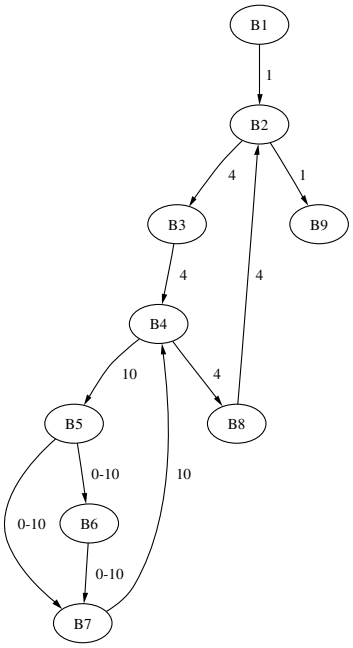

Figure 4. The control flow graph of the Bubble Sort example

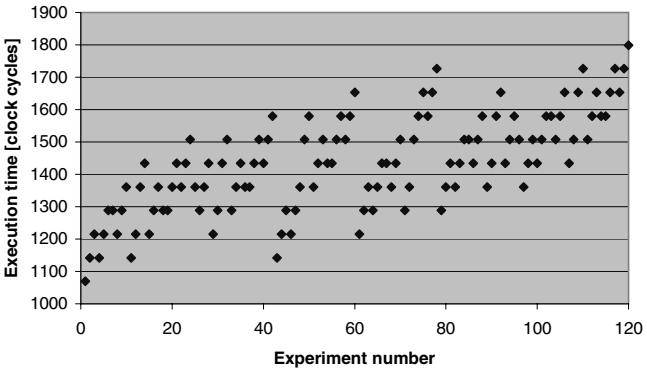

Figure 5. Execution time in clock cycles of the Bubble Sort program for all 120 permutations

cycle counter in JOP. The current time is taken at both the entry of the method and at the end, resulting in a measurement spanning from block B1 to the beginning of block B9. The last statement, the `return`, is not part of the measurement. The difference between these two values (less the additional 8 cycles introduced by the measurement itself) is given as the execution time in clock cycles (the last row in Table 2). The measured WCET and BCET values are exactly the same as the calculated values.

In Figure 5, the measured execution times for all 120 permutations of the input data are shown. The vertical axis shows the execution time in clock cycles and the horizontal axis the number of the test run. The first input sample is an already sorted array and results in the lowest execution time. The last sample is the worst-case value resulting from the reversely ordered input data. We can also see the 11 different execution times that result from executing basic block B6 (which performs the element exchange and takes 73 clock cycles) between 0 and 10 times.

This example has demonstrated that JOP is a simple target for the WCET analysis. Most bytecodes have a single execution time (WCET = BCET), and the WCET of a task depends only on the control flow. No pipeline or data dependencies complicate the low-level part of the WCET analysis.

5 Conclusion

In this paper, we presented a brief overview of the concepts for a real-time Java processor, called JOP, and the evaluation of this architecture. We performed the WCET analysis of the implemented JVM at the microcode level. This analysis provides the WCET and BCET values for the individual bytecodes. We have also shown that there are no dependencies between individual bytecodes. This feature, in combination with the method cache [6], makes JOP an easy target for low-level WCET analysis of Java applications.

References

[1] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[2] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[3] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[4] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[5] M. Schoeberl. Restrictions of Java for embedded real-time systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004.

[6] M. Schoeberl. A time predictable instruction cache for a java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[7] M. Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[8] M. Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

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

[10] Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.