VNU Journal of Science, Mathematics - Physics 27 (2011) 101-110

WCET analysis for multiprocessor based on real-time Java processor

Quang-Dung Vu*, Viet-Ha Nguyen

VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Received 11 Febuary 2011

Abstract. In this paper, we propose a solution for a worst-case execution time (WCET) analyzable Java system - a combination of a time predictable Java processor and a method WCET analysis at Java bytecode level. The execution time of bytecodes, the instructions of the Java virtual machine, is known cycle accurately for Java processor, which simplifies the low-level WCET analysis [1].

In hard real-time systems, the estimation of the WCET is essential. WCET analysis is in general an undecidable problem. As concerning above, we propose some of WCET analysis methods using control flow graph applied for high-level and low-level Java processor. Java bytecode generation has to follow stringent rules in order to pass the class file verification of the JVM. Those restrictions lead to an analysis friendly code; e.g. the stack size is known at each instruction. The control flow instructions are well defined. Branches are relative and the destination is within the same method. Detection of basic blocks in Java bytecode and construction of the control flow graph (CFG) is thus straight forward.

Keywork: Java processor, Java virtual machine, control flow graph, Java bytecode, worst case execution time, best case execution time, Integer linear programming (ILP).

1. Introduction

WCET plays an important role in verifying Java bytecode in real-time constraints. WCET analysis is a well established research area. However, there is still a gap between the theoretical findings and the practical usage of WCET analysis tools. WCET analysis is usually divided into high-level and low-level techniques. High-level WCET analysis considers the program structure by path analysis on the control flow graph (CFG). The low-level part is concerned with the execution time of machine instructions or instruction sequences. The main issue for WCET analysis method is the growing complexity of new processors. It is almost impossible to model them for the low-level analysis. We are using Java processor, called JOP [2], designed for time-predictable execution of real-time tasks instruction cache.

The paper is organized as follows. Related work is reviewed in Section 2. Section 3 presents our proposed real-time Java WCET analysis. An experimental case is presented in Section 4 together with results and discussion.

* Corresponding author. Tel.: +84915087345
E-mail: dungvq@vnu.edu.vn

2. Related work

In this section, we review several existing works that are related to our proposed method on WCET analysis in real-time Java systems. Sun introduced the first version of picoJava [3] in 1997. However, this processor was never released as a product by Sun. A redesign followed in 1999, known as picoJava-II, which was freely available. The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions. It is the most complex Java processor available. The processor can be implemented in 27.6K logic cells in an FPGA as shown in [4].

Shaw presents in [5] timing schemas to calculate minimum and maximum execution time for common language constructs. The rules allow to collapse the abstract syntax tree of a program until a final single value represents the WCET. However, with this approach it is not straight forward to incorporate global low-level attributes, such as pipelines or caches. The resulting bounds are not tight enough to be practically useful.

Computing the WCET with an integer linear programming (ILP) solver is proposed in [6] and [7]. We base our WCET analyzer on the ideas from these two groups. The WCET is calculated by transforming the calculation to an integer linear programming problem. Each basic block is represented by an edge in the T-graph (timing graph) with the weight of the execution time of the basic block. Vertices in the graph represent the split and join points in the control flow. Furthermore, each edge is also assigned an execution frequency. The constraints resulting from the T-graph and additional functional constraints (e.g., loop bounds) are solved by an ILP solver. The T-graph is similar to a CFG, where the execution time is modelled in the vertices. The motivation to model the execution time in the edges results from the observation that most basic blocks end with a conditional branch.

3. Real-time Java WCET analysis

In this section, we describe the methods of WCET analysis. There are 2 methods of WCET analysis - one is the basic method in WCET analysis- implicit path enumeration technique (IPET), which bases for high-level; the other is low-level WCET analysis, which involves to Java bytecode.

3.1. IPET WCET analysis method

IPET method [6] is based for WCET analysis, which can find the exact WCET in these simple cases because the execution time of each section of code can be considered independently of all the others. That is a high-level WCET analysis. In IPET, the program is modeled as a flow network, and integer linear programming (ILP) is used to determine the execution path with the maximal execution time. The calculation of the WCET is transformed to an ILP problem. In the CFG, each vertex represents a basic block B_i with execution time c_i. With the basic block execution frequency e_i the WCET is:

$$WCET = max \sum_{i=1}^{N} c_i e_i$$

The sum is the objective function for the ILP problem. The maximum value of this expression results in the WCET of the program. Furthermore, each edge is also assigned an execution frequency f. These execution frequencies represent the control flow through the WCET path. Two primary constraints form the ILP problem: (i) For each vertex, the sum of f_j for the incoming edges has to be equal to

the sum of the f_k of the outgoing edges; (ii) the frequency of the back edges connecting the loop body with the loop header, is less than or equal to the frequency of the edges entering the loop multiplied by the loop bound.

From the CFG, which represents the program structure, we can extract the flow constraints.With the execution frequency f of the edges and the two sets I_i for the incoming edges to basic block B_i and O_i for the outgoing edges, the execution frequency e_i of B_i is:

$$e_i = \sum_{j \in I_i} f_i = \sum_{k \in O_i} f_k$$

The frequencies f_i are the integer variables calculated by solving the ILP problem. Additional constraints are needed to handle loops. The incoming edges to the entry point of the loop, the loop header h, are classified as follows:

1. The set of incoming edges E_h that enter the loop
2. The set of back edges C_h that close the loop

With the maximum loop count (the loop bound) n, we formulate the loop constraint as

$$\sum_{j \in C_h} f_i \leq n \sum_{k \in E_h} f_k$$

Without further global constraints, the problem can be solved locally for each method. We start at the leaves of the call tree and calculate the WCET for each method. The WCET value of a method is included in the invoke instruction of the caller method. To incorporate global constraints, such as cache constraints [8], a single ILP problem is built that models the whole program. The invoke instruction i is connected to the entry and exit node of the invoked method, adding edges c and r, respectively. To ensure that only valid paths are considered, one additional constraint is needed for each method invocation:

$$\sum_{j \in I_i} f_i = f_c = f_r$$

The example of IPET WCET analysis method could be shown in Figure ??. The basic block that contains the invoke instruction is split into three new blocks: The preceding instructions, the invoke instruction, and following instructions. Consider following basic block:

- iload 1
- iload 2
- aload 0
- invokevirtual foo
- istore 3

When different versions of foo() are possible receiver methods, we model the invocation of foo() as alternatives in the graph. Following the standard rules for the incoming and outgoing edges the resulting ILP constraint for this example is:

$$f_1 = f_2 + f_3$$

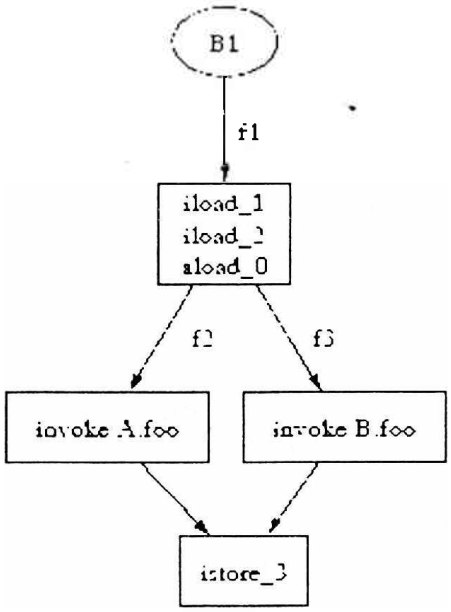

Fig. 1. Example on basic block for possibility of IPET WCET analysis method.

3.2. *Low-level WCET analysis method*

The low-level analysis concerns to executing Java bytecode inside Java processor. The Figure shows the process for execution bytecode on the real JOP systems [2].

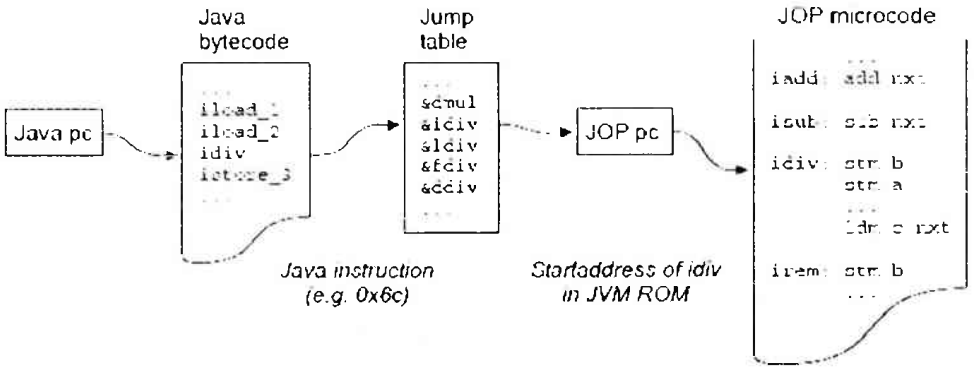

Fig. 2. Data flow for Java bytecode inside JOP.

For the low-level WCET analysis, a good model of the target architecture is needed. In our case the target architecture is simple with respect to the WCET and well documented. Following [9], the WCET analysis is performed in the microcode that implements the bytecode instructions. That means the bytecode instruction timing is derived by static analysis and no further measurements are necessary.

Most bytecode instructions that do not access memory have a constant execution time. They are executed by either one microcode instruction or a short sequence of microcode instructions. The execution time in clock cycles equals the number of microinstructions executed. As the stack is onchip, it can be accessed in a single cycle. We do not need to incorporate the main memory timing into the instruction timing of simple bytecodes. Table shows example instructions, their timing, and their

meaning (TOS is top-of-stack). Access to object, array, and class fields depend on the timing of the main memory.

Tabel 1. Execution time of simple bytecodes in cycles

Instruction	Cycle	Function
iconst 0	1	load constant 0 on TOS
bipush	2	load a byte constant on TOS
iload 0	1	load local variable 0 on TOS
iload	2	load a local variable on TOS
dup	1	duplicate TOS
iadd	1	integer addition
isub	1	integer subtraction
ifeq	4	conditional branch

Object oriented instructions, array access, and invoke instructions access the main memory. There are 2 types of memory that contain the executing bytecode inside Java processor - memory read and memory write. The following Figure shows the picture of caching memory in Java processor.

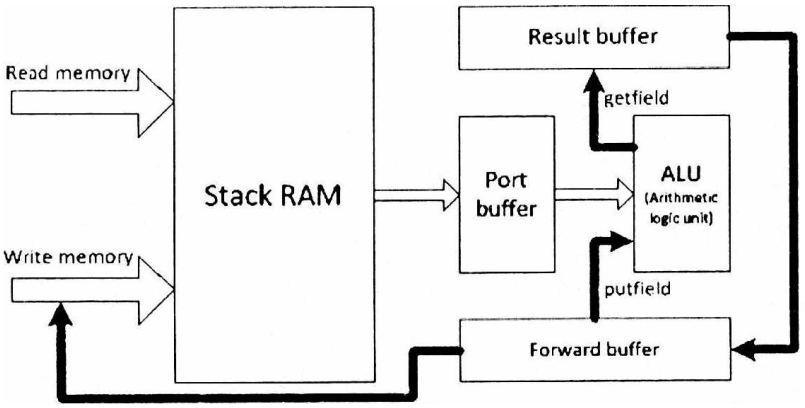

Fig. 3. Memory stack caching.

A method cache, with cache fills only on invoke and return, does not interfere with data access to the main memory. Data in the main memory is accessed with getfield and putfield, instructions that never overlap with invoke and return. In traditional caches, data access and instruction cache fill requests can compete for the main memory bus. For example, a load or store at the end of the processor pipeline competes with an instruction fetch that results in a cache miss. One of the two instructions is stalled for additional cycles by the other instruction. With a data cache, this situation can be even worse. The worst-case scenario for the memory stall time for an instruction fetch or a data load is two miss penalties when both cache reads are a miss. This unpredictable behavior leads to very pessimistic WCET bounds. Access time that exceeds a single cycle includes additional wait states (r_{ws} for a memory read and w_{ws} for a memory write). With a memory with r_{ws} wait states, the execution time for, e.g., getfield is:

$$t_{getfield} = 11 + 2r_{ws}$$

The meaning of formula above could be explained as following, the time of getfield command is taken about two times of memory read and about 11s on idle for system data transfer. A memory read in JOP microcode is split into two phases: (1) start the read transaction and (2) read the result. To avoid timing dependencies within the memory subsystem over bytecode boundaries, memory store instructions are also split into a start write and wait for completion instruction. Between those two microcode instructions the memory subsystem performs the memory transaction in parallel to the the core pipeline executing microcode instructions. Filling this slot with useful microcode instructions can hide some of the access latency. The microcode sequence in this load/store slot is straight line code and the number of hidden cycles is constant. The following example gives the exact execution time of bytecode ldc2w in clock cycles (the clock of processor is about 100MHz and Stack RAM speed for cycle - 15ns):

$$t_{ldc2_w} = 17 + max(r_{ws} - 2, 0) + max(w_{ws} - 1, 0)$$

Memory access time also determines the cache load time on a miss. The cache load time is calculated as follows - the wait state f_{ws} for a single word cache fill is:

$$f_{ws} = max(r_{ws}, 1)$$

The real system interaction between Java application, schedule, JVM and hardware is shown on the following Figure .

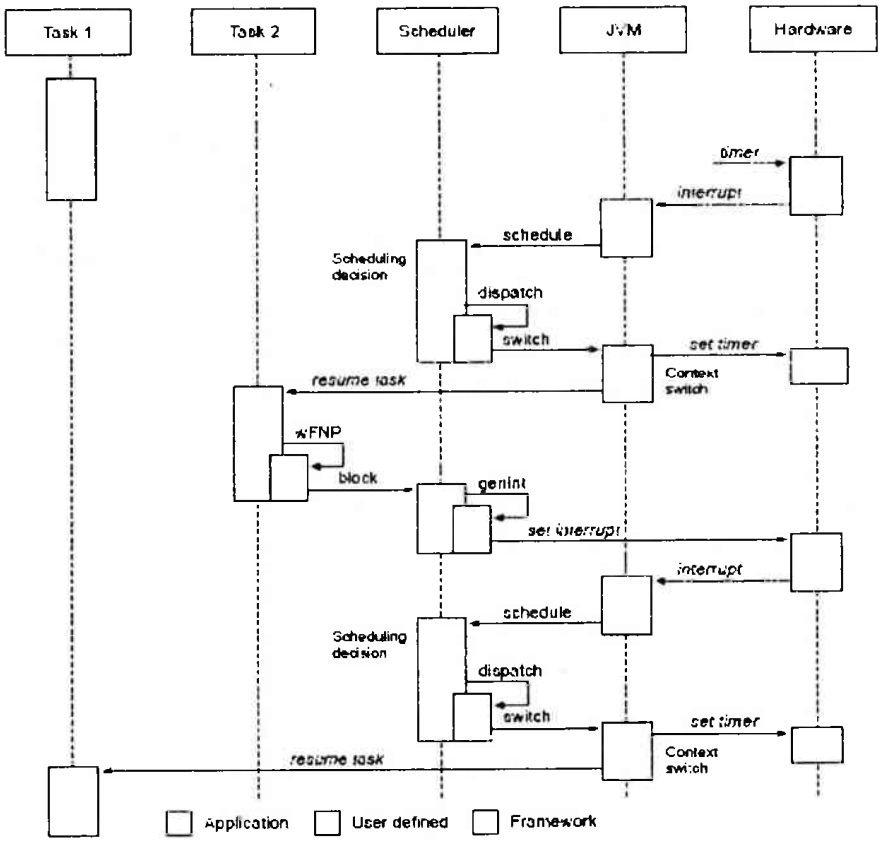

Fig. 4. Interact between Java application with schedule and hardware.

4. Experiment

In this section, we conclude with a worst and best case analysis of a classic example, the Bubble Sort algorithm. The Java source code is showing in below:

```
public class Bubble
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;
}
}
}
}
}
```

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 is 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. The disassembler of bytecode is showing in the Figure . The bytecode is divided into some of code blocks as shown in the Figure .

The annotated control flow graph (CFG) of the example, which is shown in Figure , is a result after analysing. 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. With using a formula in section 3.2 we can also calculate the execution time on each edge.

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 cycle counter in Java processor. 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. 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. The result is shown in Table . The execution frequency value is depended on input data. The result is

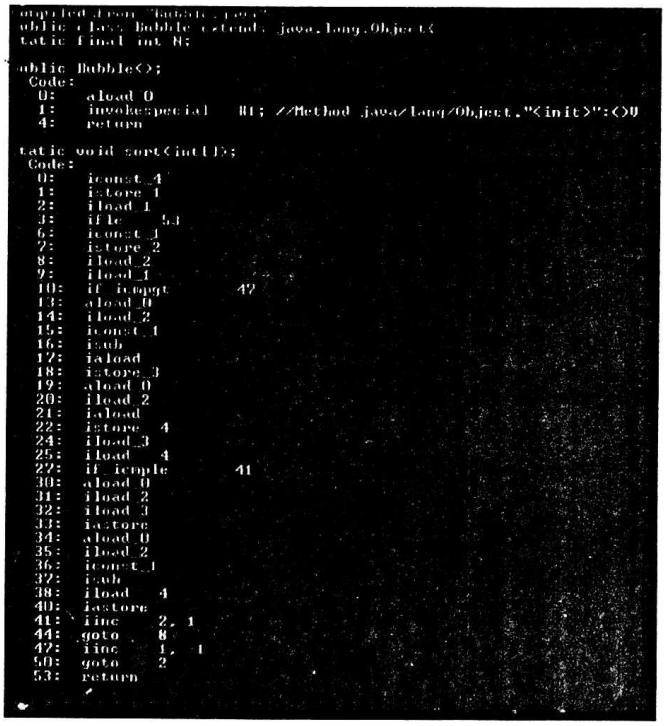

Fig. 5. Java bubble disassembler.

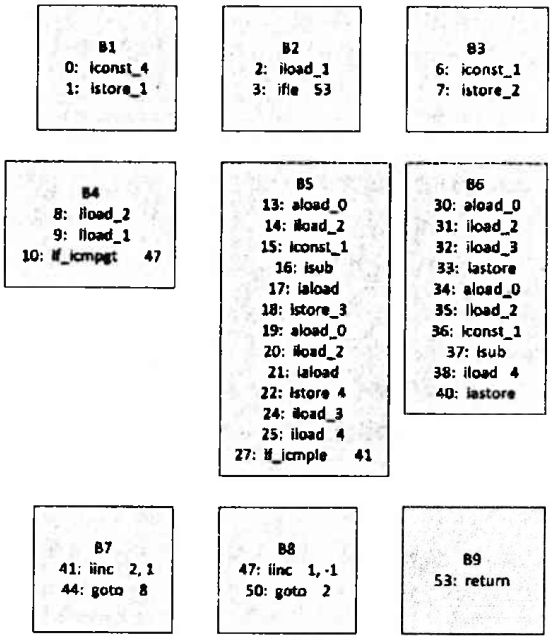

Fig. 6. Java bubble disassembler block.

based on Altera board with 48MHz Cyclone processor, and 64KB memory cache by installing Java processor.

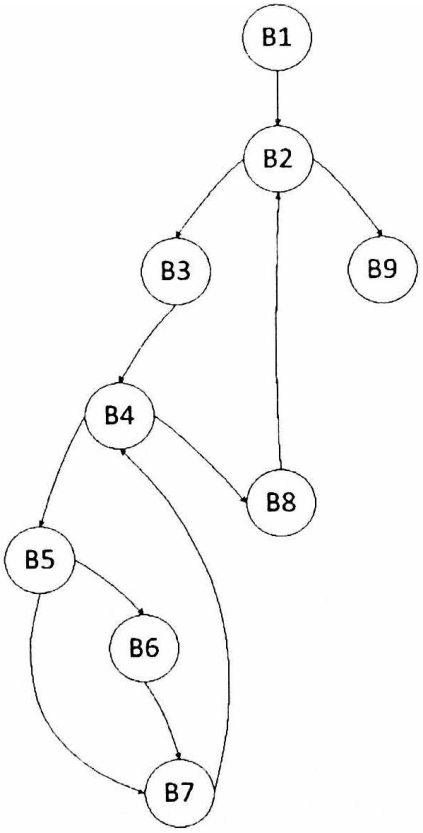

Fig. 7. Control flow graph of bytecode block.

Table 2. The result of experiment

Node	Execution time (c)	WCET = c*e (ms)
B1B2	1	B1=2
B2B3	4	B2=25
B3B4	4	B3=8
B4B5	10	B4=80
B5B6	7	B5=740 (Depending on loop)
B6B7	6	B6=730 (Depending on loop)
B7B4 c_2	10	B7=150
B5B7	5	Depending on loop
B4B8	4	
B8B2 c_1	4	B8=60
B2B9	1	return

5. Conclusion and Future work

In this paper, we have proposed methods for WCET analysis in Java processor. Some of them is still in developing, and we will come to an optimizing WCET method that applied for Java real-time embedded systems. In the future, we focus on more complex Java bytecode, that can be implemented in Java processor. We've been developing and will complete an WCET analysis tool, that supports for hight-level as well as low-level WCET analysis.

Acknowledgement. This work is partly supported by the research project No. QC.08.05 granted by Vietnam National University, Hanoi.

References

[1] Vu Quang Dung, N.V.H., Real time garbage collection for java microprocessor, *ATC 2008 conference* (2008)
[2] M. Schoeberl, Jop: A java optimized processor for embedded real-time systems, *In: PhD thesis, Vienna University of Technology* (2005).
[3] J.M. O'Connor, M. Tremblay, Picojava-i: The java virtual machine in hardware, *In: IEEE Micro*, No 17 in 2 (1997) 45.
[4] W. Puffitsch, Picojava-ii in an fpga, *In: Master's thesis*, Vienna University of Technology (2007).
[5] A.C. Shaw, Reasoning about time in higher-level language software, *In: IEEE Trans. Softw. Eng.*, No 15 (7) (1989) 875.
[6] P. Puschner, A, Schedl, Computing maximum task execution times of a graph-based approach, *Journal of Real-Time Systems*, No 13 (July 1997) 67.
[7] Y.T.S. Li, S. Malik, Performance analysis of embedded software using implicit path enumeration, *In LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on languages, compilers, and tools for real-time systems*, (1995) 88.
[8] Y.-T. S. Li, S.M., A. Wolfe, Efficient microarchitecture modeling and path analysis for real-time software, *In RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 298, Washington, DC, USA, IEEE Computer Society (1995)
[9] M. Schoeberl, A time predictable java processor, *In Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, (2006) 800.