WCET Analysis for a Java Processor

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

Rasmus Pedersen
Department of Informatics
CBS, Denmark
rup.inf@cbs.dk

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 tool that performs WCET analysis of Java bytecode. We present a Java processor, called JOP, designed for time-predictable execution of real-time tasks. JOP is an implementation of the Java virtual machine (JVM) in hardware. The execution time of bytecodes, the instructions of the JVM, is known cycle accurate for JOP. Therefore, JOP simplifies the low-level WCET analysis. A method cache, that fills whole Java methods into the cache, is analyzable with respect to the WCET. The WCET analysis tool is based on integer linear programming. The tool performs the low-level analysis at the bytecode level and integrates the method cache analysis for a two block cache.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Run-time environments, Java*

Keywords

Real-time system, Java processor, Worst-case execution time

1. INTRODUCTION

The Java programming language is a simple and safe object-oriented language. Java is in wide use for general purpose applications. The safety aspect (with respect to programming errors) of Java makes it also a candidate as a language for (hard) real-time systems [4]. For real-time systems we need to know the worst-case execution time (WCET). In this paper we present a time predictable Java processor and a WCET analysis tool. This combination is an important step towards a real-time Java system.

Worst-case execution time (WCET) analysis is a well established research area. However, there is still a gap between the theoretical findings and the practical usage of WCET analyzer 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 tools is the growing complexity of new processors. It is almost impossible to model them for the low-level analysis.

We attack this problem by a processor architecture designed to be a feasible target for the low-level analysis. The Java processor JOP [25] provides time predictable execution of Java bytecodes and a real-time method cache [23]. JOP is open-source with all VHDL files available. Therefore, the architecture is fully documented. In this paper we also present a WCET analysis tool based on integer linear programming (ILP) [22, 13]. We analyze the WCET at the bytecode level. The two block variant of the method cache is integrated in the WCET analysis.

Other issues with Java for real-time systems [4], such as real-time garbage collection, are considered as a scheduling problem [26] (including a memory allocation analysis of the application threads) and not a WCET analysis problem. WCET analysis of the garbage collection thread itself is necessary to prove that the whole system will fulfill the expected temporal properties.

The paper is structured as follows: The remainder of this section describes related work in the field of Java processors, WCET analysis based on ILP and WCET analysis for Java. Section 2 gives a brief overview of the Java processor. Section 3 gives an introduction into the WCET analysis and presents our WCET analysis tool. Analysis of the instruction cache is given in Section 4. The combination of the real-time Java processor and the WCET analysis tool is evaluated in Section 5. Section 6 concludes the paper.

1.1 Java Processors

Sun introduced the first version of picoJava [17] in 1997. However, this processor was never released as a product by Sun. A redesign followed in 1999, known as picoJava-II that is now freely available. The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions and is the most complex Java processor. The processor can be implemented [7] in about 440K gates.

aJile's JEMCore is a Java processor that is available as both an IP core and a stand alone processor [10]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. The Lightfoot 32-bit core [6] is a hybrid 8/32-bit processor based on the Harvard architecture. The core is provided as an EDIF netlist for dedicated Xilinx devices and as an ASIC. The Cjip processor [9, 11] supports multiple instruction sets, allowing Java, C, C++ and assembler to coexist. The JVM is implemented largely in microcode (about 88% of the Java bytecodes). Microcode instructions execute in two or three cycles. A JVM bytecode requires several microcode instructions. Komodo [12] is a multithreaded Java processor with a four-stage pipeline. It is intended as a basis for research on real-time scheduling on a multithreaded microcontroller.

JTRES '06 October 11-13, 2006 Paris, France

For all, except the Cjip, described Java processors no timing information for the instructions are available[1]. Therefore, it is not possible, at least for us, to provide a safe low-level analysis for those processors. A detailed comparison of embedded Java systems and JOP can be found in [24].

1.2 WCET Analysis

Shaw presents in [28] timing schemas to calculate minimum and maximum execution time for common language constructs. The rules allow to collapse the control flow graph of a program till a final single value represents the WCET. However, with this approach it is not easy to incorporate global low-level attributes, such as pipelines or caches.

Computing the WCET with an integer linear programming solver is proposed by Puschner and Schedl [22] and Li and Malik [13]. The approach is named graph-based and implicit path enumeration respectively. We base our WCET analyzer on the ideas from these two groups.

Puschner and Schedl [22] calculate the WCET by transforming it to an integer linear programming (ILP) problem. Each basic block is represented by an edge e_i in the T-graph (timing graph) with the weight of the execution time of the basic block. Vertices v_i in the graph represent the split and join points in the control flow. Furthermore, each edge is also assigned an execution frequency f_i. 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 control flow graph (CFG), where the execution time is modeled 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. Those branches usually have different execution times when taken or not. This difference is represented by two edges with different weights.

Li and Malik [13] follow a similar approach with ILP. However, they use the CFG as the basis to build the ILP problem. In [14] they extend the approach to model the instruction cache with cache conflict graphs. The evaluation with an Intel i960 processor shows tight results for small programs. However, the conservative modeling of the register window (overflow/underflow on each function call/return) adds 50 cycles to each call and return. This observation is another argument for a WCET aware processor architecture. An overview of WCET related research can be found in [19].

1.3 WCET Analysis for Java

WCET analysis at the bytecode level became a research topic, at the time Java was considered for future real-time systems. WCET analysis at the bytecode level was first considered in [3]. It is argued that the well formed intermediate representation of a program in Java bytecode, that can also be generated from compilers for other languages (e.g. Ada), is a viable path to a portable WCET analysis tool. In that paper, annotations for Java and Ada are presented to guide the WCET analysis at bytecode level. This work is extended by [2] to address the machine-dependent low-level timing analysis. Worst-case execution frequencies of Java bytecodes are introduced for a machine independent timing information. Pipeline effects (on the target machine) across bytecode boundaries are modeled by a *gain time* for bytecode pairs.

In [18] a portable WCET analysis is proposed. The abstract WCET analysis is performed on the developer site and generates abstract WCET information. The concrete WCET analysis is performed on the target machine by replacing abstract values within the WCET formulae by the machine dependent concrete values.

[1]We tried hard to get information for the aJile processor.

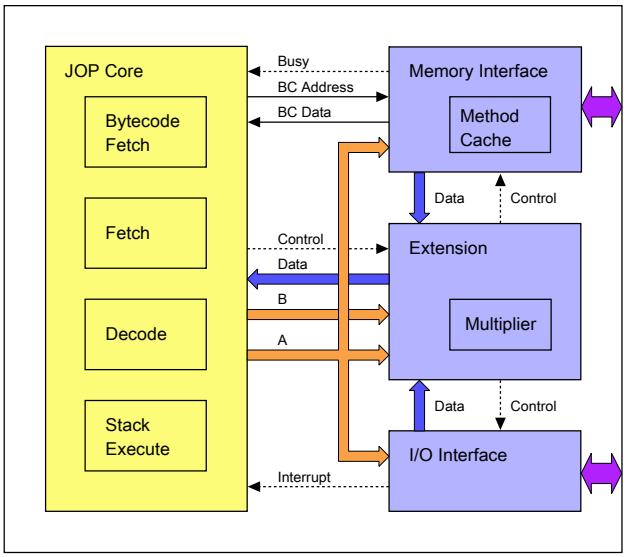

Figure 1: Block diagram of JOP

2. A TIME PREDICTABLE PROCESSOR

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.

JOP [25], the Java Optimized Processor, is an intended solution to this issue. The processor architecture is built from ground up to be WCET analyzable. Features, such as the real-time stack cache and method cache, provide top performance and are still analyzable. The execution time for Java bytecodes can be exactly predicted in terms of the number of clock cycles. JOP is the smallest and fastest Java processor available today [24].

2.1 JOP Architecture

JOP is a stack computer with its own instruction set, called microcode. Java bytecodes are translated into microcode instructions or sequences of microcode in hardware. The difference between the 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 processor core contains the three microcode pipeline stages: *microcode fetch*, *decode* and *execute* and an additional translation stage *bytecode fetch*. The module called extension provides the link between the processor core, and the memory and IO modules. The memory interface provides a connection between the main memory and the processor core. It also contains the method cache. The extension module controls data read and write. The *busy* signal is used by a microcode instruction to synchronize the processor core with the memory unit. The core executes microcode concurrently to memory access.

2.2 The Processor Pipeline

JOP is a fully 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 microcode 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. The second pipeline stage fetches JOP instructions from the internal microcode memory. Besides the usual decode function, the third pipeline stage also generates addresses for the stack cache.

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. The short pipeline results in short branch delays. Therefore, a hard to analyze branch prediction logic can be avoided.

In [27] we have shown that no processor resources are shared across bytecode boundaries. That means that there are no pipeline dependencies between two bytecodes that could generate an unbound timing effect. The processor is designed to avoid any timing anomalies as found in standard microprocessors [16]. We do not need to model the pipeline in the low-level WCET analysis.

2.3 Cache

In order to fill the gap between processor speed and the memory access time, caches are mandatory, even in embedded systems. 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.

2.3.1 Stack Cache

JOP contains no data cache in the traditional sense. However, the stack that contains method local variables and is used for the stack operations is a heavily accessed memory region. Therefore we place the stack – or part of it – in on-chip memory. This *stack cache* is not automatically exchanged with the main memory. That would result in a very hard to analyze feature. The exchange with the main memory can be done either at method invocation and return or at the thread switch.

The stack height is statically known for each instruction (a result from the verification restrictions of Java class files). The transfer time between the on-chip stack cache and the main memory can be integrated in the same way as the method cache load at the invoke and return instruction.

Keeping the stack in the cache for each thread results in faster invokes and returns, but limits the maximum stack hight for each thread. The local stack is now part of the threads context and has to be saved and restored on a thread switch. The additional time has to be added to the context switch time. As the maximum height of the stack is known (by analyzing the call graph for each thread) this time is bounded.

2.3.2 Method Cache

Typical Java programs consist of short methods (see [25]). There are no branches out of the method and all branches inside are relative. In the proposed architecture, the full code of a method is loaded into the cache before execution. The cache is filled on invocations and returns. This means that all cache misses are lumped together with a known miss penalty. The full loaded method and relative addressing inside a method also result in a simpler cache design. Tag memory and address translation are not necessary. As the cache stores whole methods it is named *method cache* [23].

The simplest version of a method cache can cache just a single method. Although less efficient than a conventional instruction cache, it can be incorporated very easily into the WCET analysis. The time needed for the memory transfer has to be added to the invoke and return instructions.

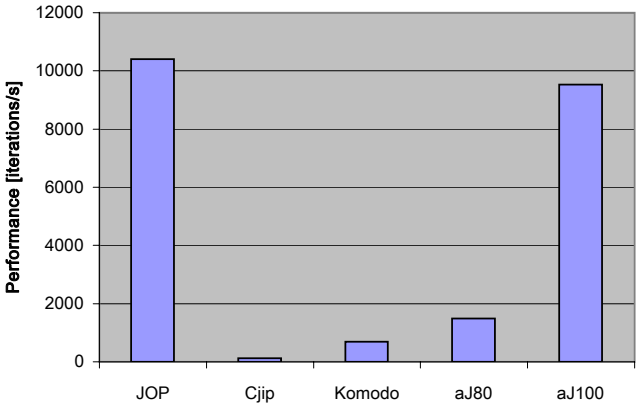

Figure 2: Performance comparison of different Java processors

An extension is a method cache that can cache two methods – the two-block cache. With two blocks/methods we have to decide which block is replaced on a cache miss. With only two blocks, least-recently used (LRU) is natural and trivial to implement. The WCET analysis is slightly more complex than with a single block. We can improve the hit rate by adding more blocks to the cache. However, the cache size increases with the number of blocks.

Several cache blocks, all of the size as the largest method, are a waste of cache memory. Using smaller block sizes and allowing a method to span over several blocks, the blocks become very similar to cache lines. The main difference from a conventional cache is that the blocks for a method are all loaded at once and need to be consecutive[2]. We name this organization of the cache the *variable block cache*. Choosing the block size is a major design decision. Smaller block sizes allow better memory usage, but hit detection needs either more hardware or more time.

2.4 Performance

Although JOP is intended as a processor for embedded real-time systems, whereas accurate WCET analysis is more important than average case performance, its general performance is still important.

To provide a realistic workload for embedded systems, a real-time application (the *Kippfahrleitung*) was adapted to create the benchmark. A second benchmark contains the generation of a request and response within an UDP/IP stack. Figure 2 shows the geometric means of the two benchmarks. The results are in iterations per second – a higher value means higher performance. Both benchmarks are also used in the evaluation of our WCET analysis tool in Section 5.

JOP is compared against some of the Java processors described in Section 1.1: the Cjip [9], Komodo [12], and two versions of the aJile processor [10] (aJ80 and aJ100). Only the aJ100 is almost as fast as JOP. This comparison also shows that a time-predictable processor architecture does not to be slow in the average case.

2.5 Size

One major design objective in the development of JOP is to create a small system that can be implemented in a low-cost FPGA.

[2]A method loaded over the cache end to the cache start is considered continuous as the cache is addressed with a modulo counting program counter.

Table 1: FPGA soft-core processors

Processor	Resources (LC)	Memory (KB)	fmax (MHz)
JOP Minimal	1,077	3.25	98
JOP Typical	2,049	3.25	100
Lightfoot	3,400	4	40
LEON3	7,978	10.9	35

Table 1 shows the resource usage for different configurations of JOP and different soft-core processors implemented in an Altera Cyclone low-cost FPGA. The size is given by the two basic structures in an FPGA: Logic Cells (LC) and embedded memory blocks.

The typical configuration contains some useful I/O devices such as an UART and a timer with interrupt logic for multi-threading. In the minimal configuration shift and multiply are implemented in microcode. Lightfoot [6] is a commercial Java processor available to be implemented in an FPGA. As a reference, LEON3 [8], the open-source implementation of the SPARC V8 architecture, is given in the last row.

3. WCET ANALYSIS FOR JOP

In hard real-time systems the estimation of the worst-case execution time is essential. In general WCET analysis is undecidable. In [21] Puschner and Koza provide program restrictions to make this problem decidable:

1. Programs must not contain any recursion
2. Absence of function pointers
3. The upper bound of each loop has to be known

Recursive algorithms have to be transformed to iterative ones. For our WCET analyzer the loop bounds have to be annotated in the program source. However, we want to relax the second restriction. Function pointers are very similar to inherited or overridden methods. Function pointers and methods are dispatched at runtime. For an object-oriented language this mechanism is fundamental. In contrast to function pointers (e.g. in C) we can statically analyze which methods can be invoked when the whole program is known. Therefore we allow dynamic dispatching of methods in Java in our analysis. We replace the function pointer restriction by the following restriction:

- Dynamic class loading is not allowed

As the full application has to be available for the WCET analysis dynamic class loading is of little use. For embedded real-time systems this is not a serious restriction.

3.1 High-Level WCET Analysis

From the class files that make up the application the relevant information is extracted. The control flow graph (CFG) of the basic blocks[3] is extracted from the bytecodes. Annotations for the loop counts are extracted from comments in the source. Furthermore, the class hierarchy is examined to find all possible targets for a method invoke.

Java bytecode generation has to follow stringent rules [15] 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. In the normal program there is no instruction that loads a branch destination in a local variable or onto the stack[4]. Thus detection of basic blocks in Java bytecode and construction of the CFG is straight forward.

In Java class files there is more information available than in compiled C/C++ executables. All links are symbolic and it is possible to reconstruct the class hierarchy from the class files. Therefore, we can statically determine all possible targets for a virtual method invoke.

3.1.1 WCET Annotations

For our WCET analyzer we use loop bound annotations in the Java source. The code in Figure 3 shows an example of the annotation. When compiling Java, the source line information is included in the class file. Therefore, when a loop is detected in the CFG, the relevant source line for the loop header is looked up in the source and the annotation is extracted.

Annotations given as source comments are simple and less intrusive than using a predefined dummy class [3]. Two variants of the loop bounding annotation are supported: one with an exact bound[5] (=) and one that places an upper bound on the iterations (<=). The extension to more elaborate annotations, as suggested in [21, 3], can provide tighter WCET bounds.

3.1.2 ILP Formulation

The calculation of the WCET is transformed to an integer linear programming (ILP) problem [22]. 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 the sum of the f_k of the outgoing edges; (ii) The loop constraints are formulated with the loop bound and edges from the loop header.

From the CFG, that 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 ingoing edges to basic block B_i and O_i for the outgoing edges the execution frequency e_i of B_i is:

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

The f are the decision variables found by the solution of the ILP problem. Furthermore, we add two special vertices to the graph: The start node S and the termination node T. The start node S has only one outgoing edge that points to the first basic block of the program. The execution frequency f_s of this edge is set to 1.

[3] A basic block is a sequence of instructions without any jumps or jump targets within this sequence.

[4] The exception are bytecodes `jsr` and `ret` that use the stack and a local variable for the return address of a method local subroutine. This construct can be used to implement the `finally` clauses of the Java programming language. However, this problematic subroutine can be easily inlined [1]. Furthermore, Sun's Java compilers version 1.4.2 and later compile `finally` blocks without subroutines.

[5] The exact bound has been used to find best-case values for some test settings.

```
public static int loop(boolean b, int val) {

   for (int i=0; i<10; ++i) {      //@WCA loop=10
      if (b) {
         for (int j=0; j<3; ++j) { //@WCA loop=3
            val *= val;
         }
      } else {
         for (int j=0; j<7; ++j) { //@WCA loop=7
            val += val;
         }
      }
   }
   return val;
}
```

Figure 3: The example used for the WCET analysis with ILP

The termination node T has only incoming edges with the sum of the frequencies also set to 1. When the method contains multiple return statements, all are connected to the node T. That means that the program is executed once and can only terminate through one path.

Loop bounds are functional constraints for the ILP problem. A special vertex, the loop header, is connected by following edges:

1. Incoming edges that enter the loop with frequency f_h
2. One outgoing edge entering the loop body with frequency f_l
3. One incoming edge that closes the loop
4. One loop exit edge

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

$$f_l \leq n \sum f_h$$

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 [14], a single CFG is built that contains the whole program by inserting the CFG of a method at the invoke instruction. The effect is the same as when we inline each method.

In Section 4 we will show that the cache constraints for a two block method cache can be resolved using only method local information. Therefore, we can stay with the method local solving of the ILP problem and avoid scaling issues from building a whole program CFG.

3.1.3 An Example

To illustrate the WCET analysis flow we provide a small example. Figure 3 shows the Java source code that contains nested loops with a condition. The loops are annotated with the maximum loop counts in a comment. In our target hardware the multiplication takes longer than the addition. Therefore, in this example it is not obvious which branch will result in the WCET path.

Table 2 shows the bytecodes and basic blocks of the example as generated by our WCET analysis tool. The fourth column gives the execution time of the bytecodes in clock cycles. The fifth column gives the execution time of the basic blocks. These are the values used for the ILP equations.

From the basic blocks we can construct the CFG as shown in Figure 4. The vertices represent the basic blocks and include the

Table 2: Java bytecode and basic blocks

Block	Addr.	Bytecode	Cycles	BB Cycles
B1	0:	iconst_0	1	
	1:	istore_2	1	2
B2	2:	iload_2	1	
	3:	bipush	2	
	5:	if_icmpge 53	4	7
B3	8:	iload_0	1	
	9:	ifeq 29	4	5
B4	12:	iconst_0	1	
	13:	istore_3	1	2
B5	14:	iload_3	1	
	15:	iconst_3	1	
	16:	if_icmpge 47	4	6
B6	19:	iload_1	1	
	20:	iload_1	1	
	21:	imul	35	
	22:	istore_1	1	
	23:	iinc	8	
	26:	goto 14	4	50
B7	29:	iconst_0	1	
	30:	istore_3	1	2
B8	31:	iload_3	1	
	32:	bipush	2	
	34:	if_icmpge 47	4	7
B9	37:	iload_1	1	
	38:	iload_1	1	
	39:	iadd	1	
	40:	istore_1	1	
	41:	iinc	8	
	44:	goto 31	4	16
B10	47:	iinc	8	
	50:	goto 2	4	12
B11	53:	iload_1	1	
	54:	ireturn	19	20

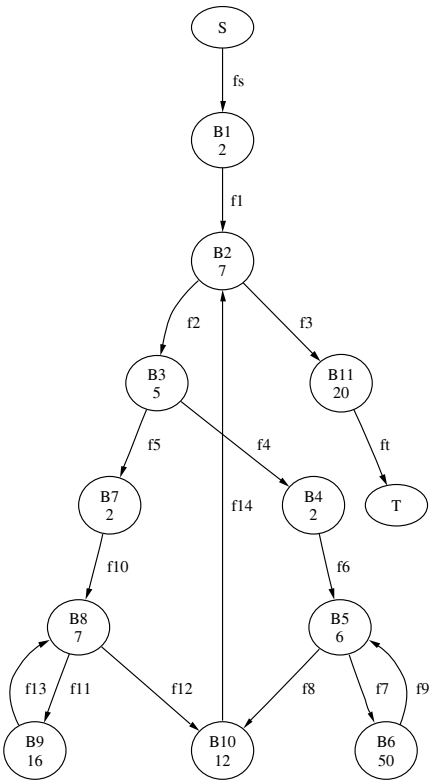

Figure 4: CFG of the example

```
/* Objective function */
max: t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11;
/* flow constraints */
S:  fs = 1;
B1: fs = f1;
B2: f14 + f1 = f2 + f3;
B3: f2 = f4 + f5;
B4: f4 = f6;
B5: f9 + f6 = f7 + f8;
B6: f7 = f9;
B7: f5 = f10;
B8: f10 + f13 = f11 + f12;
B9: f11 = f13;
B10: f12 + f8 = f14;
B11: f3 = ft;
T: ft = 1;
/* loop bounds */
f2 = 10 f1;
f7 = 3 f6;
f11 = 7 f10;
/* execution time (with incoming edges) */
t1 = 2 fs;
t2 = 7 f14 + 7 f1;
t3 = 5 f2;
t4 = 2 f4;
t5 = 6 f9 + 6 f6;
t6 = 50 f7;
t7 = 2 f5;
t8 = 7 f10 + 7 f13;
t9 = 16 f11;
t10 = 12 f12 + 12 f8;
t11 = 20 f3;
```

Figure 5: ILP equations

execution time in clock cycles. We can identify block B2 as the loop header for the outer loop. B3 is the branch node. B5 and B8 are the loop headers for the inner loops.

From the CFG we can extract the flow constraints by following fact: The execution frequency of a basic block is equal to the execution frequency of all incoming edges and equal to the execution frequency of all outgoing edges. E.g. for block B2 the execution frequency e_2 is:

$$e_2 = f_1 + f_{14} = f_2 + f_3$$

The loop constraints are formulated by multiplying the frequencies of the edges that enter the loop via the header by the loop bound. The loop bounds are automatically extracted from the source annotations. For the outer loop in the example this is:

$$f_2 = 10 f_1$$

In Figure 5 the resulting equations, as generated by our tool, for the integer linear programming problem are listed. We use the open-source ILP solver `lp_solve`.

The tool `lp_solve` gives a result of 2029 cycles. We run this example on the Java processor for verification. As the execution time depends only on a single boolean variable, `b` (see Figure 3), it is trivial to measure the actual WCET. We measure the execution time with a cycle counter from the start of block B1 until the exit of the outer loop at block B2. The measured result is 2009 cycles. The last block B11, that contains the return statement, is not part of the measurement. When we add the 20 cycles for the block B11 to our measured WCET we get 2029 cycles. This measured result is exactly the same as the analytical result.

3.1.4 Dynamic Method Dispatch

Dynamic dispatching at runtime of inherited or overridden instance methods is a key feature of object-oriented programming. Therefore, we allow dynamic methods as a *better controlled* form of function pointers. The full class hierarchy can be extracted from the class files of the application. From the class hierarchy we can extract all possible receiver methods for an invoke. We include all possible receivers as alternatives in the ILP constraints. It has to be noted that, without further analysis or annotations, this can lead to pessimistic WCET estimates.

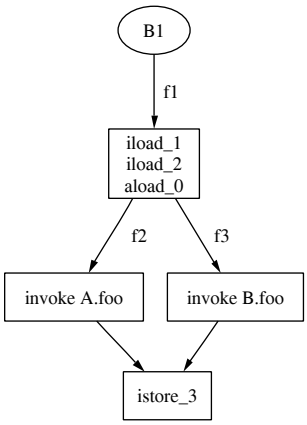

Figure 6: Split of the basic block for instance methods

We split the basic block that contains the invoke instruction into following 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 available in the class tree we model the invocation of `foo()` as alternatives in the graph. The example for two classes `A` and `B` that are part of the same hierarchy is shown in Figure 6. Following the standard rules for the ingoing and outgoing edges the resulting ILP constraint for this example is:

$$f_1 = f_2 + f_3$$

3.2 Low-Level WCET Analysis

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 known. In [27] we have performed the WCET analysis of the microcode that *implements* the bytecode instructions. That means the bytecode instruction timing is derived by static analysis and no measurements are necessary. We have shown that there are no dependencies, neither through pipeline effects nor shared processor resources, between individual bytecodes. The detailed bytecode instruction timing can be found in [25]. The latest revision of the timing information is available at `http://www.jopdesign.com/`.

3.2.1 Basic Bytecodes

Simple bytecode instructions are executed by either one microinstruction or a short sequence of microinstructions. The execution time in cycles equals the number of microinstructions executed. As the stack is on-chip it can be accessed in a single cycle. We do not need to incorporate main memory timing into the instruction timing.

3.2.2 Memory Access

Object oriented instructions, array access, and invoke instructions access the main memory. Therefore we have to model the memory access time. We assume a simple SRAM with a constant access time. Access time that exceeds a single cycle includes additional wait states (r_{ws} for a memory read and w_{ws} for a memory write). However, the memory subsystem performs read and write parallel to the execution of microcode. Therefore, some access cycles can be hidden.

The following example gives the exact execution time of bytecode `ldc2_w` in clock cycles:

$$t_{ldc2\_w} = 17 + \begin{cases} r_{ws}-2 & : \quad r_{ws} > 2 \\ 0 & : \quad r_{ws} \leq 2 \end{cases} + \begin{cases} r_{ws}-1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

Thus, for a memory with two cycles access time ($r_{ws} = 1$), as we use it for a 100MHz version of JOP with a 15ns SRAM, the wait state is completely hidden by microcode instructions for this bytecode.

Memory access time also determines the cache load time on a miss. For the current implementation the cache load time is calculated as follows: the wait state c_{ws} for a single word cache load is:

$$c_{ws} = \begin{cases} r_{ws}-1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

On a method invoke or return the bytecode has to be loaded into the cache on a cache miss. The load time b is:

$$b = \begin{cases} 6+(n+1)(2+c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cach hit} \end{cases}$$

where n is the length of the method in number of 32-bit words. For short methods the load time of the method on a cache miss, or part of it, is hidden by microcode execution. As an example the exact execution time for the bytecode `invokestatic` is:

$$\begin{aligned} t \quad = \quad & 74 + r + \begin{cases} r_{ws}-3 & : \quad r_{ws} > 3 \\ 0 & : \quad r_{ws} \leq 3 \end{cases} \\ + \quad & \begin{cases} r_{ws}-2 & : \quad r_{ws} > 2 \\ 4 & : \quad r_{ws} \leq 2 \end{cases} + \begin{cases} b-37 & : \quad b > 37 \\ 0 & : \quad b \leq 37 \end{cases} \end{aligned}$$

For `invokestatic` a cache load time b of up to 37 cycles is completely hidden.

3.2.3 Bytecodes in Java

Bytecode can even be implemented in Java on JOP. In this case a static method from a JVM internal class gets invoked when such a bytecode is executed. For the WCET analysis we have to substitute this bytecode by an invoke instruction to this method. The influence on the cache (the bytecode execution results in a method load) is analyzed in the same way as for *ordinary* static methods (see Section 4).

3.2.4 Native Methods

In JOP most of the JVM internal functionality, such as input, output, and thread scheduling, is implemented in Java (i.e. implemented in software). However, Java and the JVM do not allow access to memory, peripheral devices or processor registers. For this low-level access to system resources we need *native* methods. However, for a Java processor the *native* language is still Java bytecode. We solve this issue by substituting the native method invocation by a special bytecode instruction on class loading. Those special bytecodes are implemented in JOP microcode in the same way as regular bytecodes. With this translation *trick* we get a link from standard Java code to microcode without the overhead of a native method call. The execution time of the native methods (or in other words special bytecodes) is given in the same way as the execution time for standard bytecodes.

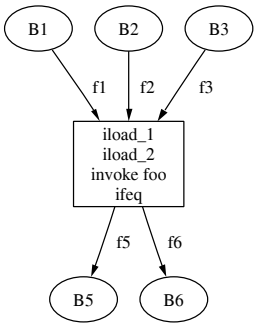

Figure 7: Basic block with an invoke instruction

3.3 Implementation

The WCET analyzer (WCA) is an open source Java program based in the ILP approach, as described in Section 3.1.2. To access the class files we use the Byte Code Engineering Library (BCEL) [5]. BCEL allows inspection and manipulation of class files and the bytecodes of the methods. WCA extracts the basic blocks from the methods and builds the CFG. Within the CFG the WCA detects loops and the loop head. From the source line attribute of the loop head the annotation of the loop count is extracted. WCA uses the open-source ILP solver `lp_solve`. `lp_solve` is integrated into WCA by directly invoking it via the Java library binding.

Besides generating a single number (the WCET in clock cycles) WCA gives a more detailed feedback on the program structure. Each individual method is listed with basic blocks and execution time on bytecodes, basic blocks, and cache miss times. This output is similar to Table 2, but with more detailed information. WCA also generates a graphical representation of the CFG for each method and for the whole program. The WCET path through the CFG is highlighted in the graph. This form of feedback for the programmer can help to optimize the WCET path in the application.

4. CACHE ANALYSIS

From the properties of the Java language — usually small methods and relative branches — we derived the novel idea of a *method cache* [23], i.e. an instruction cache organization in which whole methods are loaded into the cache on method invocation and on the return from a method.

The method cache is designed to simplify the WCET analysis. Due to the fact that all cache misses are included in two instructions (*invoke* and *return*) only, the instruction cache can be ignored on all other instructions. The time needed to load a complete method is calculated using the memory properties (latency and bandwidth) and the length of the method. On an invoke, the length of the invoked method is used, and on a return, the method length of the caller.

Integration of the method cache into the WCET analysis is straight forward. As the cache hits or misses can only happen at method invocation or return from a method we can model the miss times as extra vertices in the graph. Figure 7 shows an example with 6 connected basic blocks. Basic block B4 is shown as a box and has three incoming edges (f_1,f_2,f_3) and two outgoing edges (f_5,f_6). B4 contains the invocation of method `foo()` surrounded by other instructions. The execution frequency e_4 of block B4 in the example

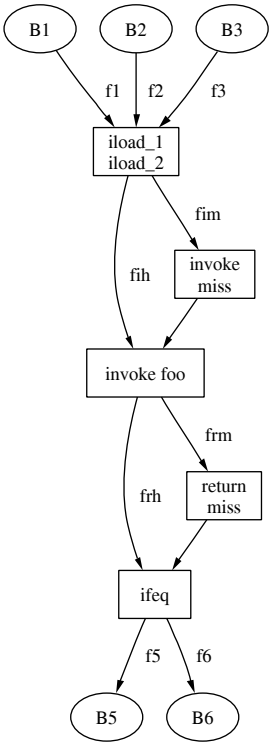

Figure 8: Split of the basic block and cache miss blocks

is

$$e_4 = f_1 + f_2 + f_3 = f_5 + f_6$$

We split a basic block that contains a method invoke (B4 in our example) into several blocks so one block contains just the invoke instruction. The miss on invoke and return are modeled as extra blocks with the miss penalty as execution time.

The miss for the return happens during the return instruction. On a miss the caller method has to be loaded into the cache. Therefore the miss penalty depends on the caller method size. However, as the return instruction is the last instruction executed in the called method we can model the return miss time at the caller side after the invoke instruction instead of the called side. This approach simplifies the analysis as both methods, the caller and the called, with their respective length are known at the occurrence of the invoke instruction.

Figure 8 shows the resulting graph after the split of block B4 and inserting vertices for the cache misses. The miss penalty is handled in the same way as execution time of basic blocks for the ILP objective function. The additional constraints for the control flow in our example are

$$e_4 = f_{ih} + f_{im}$$

$$e_4 = f_{rh} + f_{rm}$$

with the invocation hit f_{ih} and miss f_{im} frequency and the return hit f_{rh} and miss f_{rm} frequency.

It has to be noted that misses are always more expensive than hits. A conservative bound on the hit frequency is a safe approximation when the exact information is missing. As the hit or miss time is contained within a single bytecode execution there are no issues with timing anomalies [16].

As a next step we have to formulate the relation between the hit and the miss frequency. In [23] several variants of the method cache are described:

1. A single block that can only cache a single method
2. Several blocks that can each cache a single method
3. A variable block cache where a method can span several blocks

4.1 Single Block Cache

The *single block cache* can store only a single method. Therefore it is very simple to analyze: Each invoke and each return results in a miss. We can include both miss times in the invoke execution time. It has to be noted that this single method cache still is a caching solution. The actual fetch of the bytecodes is from the cache. It provides a performance enhancement compared to a non-caching architecture.

4.2 Dual Block Cache

A natural extension to the single block cache is usage of several cache blocks, each one storing exactly one method. With more than one method in the cache, cache hit detection has to be performed as part of the WCET analysis. Considering the minimal variant of two blocks the analysis can be performed locally. We do not need to consider the whole program flow. We use a least recently used (LRU) replace strategy. LRU is quite natural for two blocks as we fill the block which we are currently not using.

A cache hit on invoke or return can only happen when the invoked method is a leaf in the call tree. In that case the cache contains the caller method and the called method. If we would invoke another method the former caller method would be replaced in the cache.

As a conservative estimate we only consider methods that are statically known to be leafs, i.e. methods that do not contain any invoke statement. Furthermore, we restrict the analysis to methods invoked within a loop. In that case the hit detection is as follows:

Invoke A hit on invoke is only possible if the method is the same as the last invoked. That means a single method invoked in a loop[6]. In this case the first invocation is probably a miss and all following invokes are guaranteed hits. With the loop count n and the execution frequency f_h entering the loop head the hit and miss frequencies are:

$$\begin{aligned} f_{im} &= f_h \\ f_{ih} &= (n-1) f_h \end{aligned}$$

Return A return is always a hit on the leaf as the caller is still in the other block. In this case we can remove the miss block from the graph.

Figure 9 illustrates the invocation of a method in a loop. Basic block B2 is the loop head. With loop bound n the resulting loop and cache constraints for this example are:

$$\begin{aligned} f_2 &\leq n f_1 \\ f_2 &= f_{im} + f_{ih} \\ f_{im} &= f_1 \\ f_{ih} &\leq (n-1) f_1 \end{aligned}$$

[6] A second invocation of the same method in straight line code, without an invoke of a different method in between, will also result in a hit.

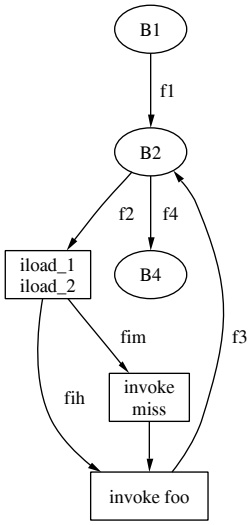

Figure 9: A method invocation in a loop

It has to be noted that the cache constraints are conservative as there could be another surrounding loop without a method invocation in the control flow.

An extension of the two block cache to several blocks needs the whole control flow to model. Furthermore, reserving blocks for single methods is a waste of cache capacity. A better solution is described in the next section.

4.3 Variable Block Cache

The *variable block cache* divides the cache in several blocks similar to cache lines in a conventional cache. However, a single method has to be loaded in a continuous region of the cache.

The variable block cache needs the whole program call graph and the whole program CFG for the analysis and is considered as future work. However, every hit in the two-block cache will also be a hit in a variable block cache (of the same size). Consider, as an example, two blocks each 1KB and a variable block cache of 2KB. In that case the method size is restricted to 1KB due to the dual block cache size. The variable block cache can at least cache two methods. At method invocation the blocks that are currently used are not replaced in the variable block cache. Therefore, we use the other blocks for the called method. The sum of those other blocks is at least 1KB. On the return from a leaf method we find the caller method still in the cache. This configuration is stable for the invoke hit case in the loop. A tradeoff is to analyze the program by assuming a two block cache but using a variable block cache.

With the variable block cache, it could be argued that the WCET analysis becomes too complex, but it is nevertheless simpler than that with the direct-mapped cache. We only have to consider invoke and return instructions and not all instructions in a cache line for a cache analysis.

5. EVALUATION

For the evaluation of our tool we analyze and measure various benchmarks (see Table 3). It has to be noted that we actually cannot measure the real WCET. If we could measure it, we would not need to perform the WCET analysis at all. The measurement gives us confidence that we have no serious bugs in the analysis and an idea of the pessimism of the analyzed WCET. The benchmarks `Lift` and `Kfl` are real-world examples that are in industrial use. `Kfl` and `UdpIp` are also part of an embedded Java benchmark suit that is used in Section 2.4.

Table 3: Benchmark examples

Program	Description	LOC
`crc`	CRC calculation for short messages	8
`robot`	A simple line follower robot	111
`Lift`	A lift controler	635
`Kfl`	*Kippfahrleitung* application	1,366
`UdpIp`	UDP/IP benchmark	1,297

Table 4: Measured and estimated WCET with result in clock cycles

Program	Measured (cycle)	Estimated (cycle)	Pessimism (ratio)
`crc`	1,552	1,620	1.04
`robot`	736	775	1.05
`Lift`	7,214	11,249	1.56
`Kfl`	13,334	28,763	2.16
`UdpIp`	11,823	219,569	18.57

Table 4 shows the measured execution time and the analyzed WCET. The last column gives an idea of the pessimism of the WCET analysis. For very simple programs, such as `crc` and `robot`, the pessimism is quite low. For the `Lift` example it is in an acceptable range. The difference between the measurement and the analysis in the `Kfl` example results from the fact that our measurement does not cover the WCET path. We only simulate input values and commands for the mission phase. However, the main loop of `Kfl` also handles service functions. Those functions are not part of the mission phase, but make up the WCET path.

The large conservatism in `UdpIp` results from the loop bound in the IP and UDP checksum calculation. It is set for a 1500 byte packet buffer that can be handled by our UDP/IP stack. However, in the benchmark the UDP payload is only 8 bytes. When setting this loop bound according to the benchmark, the WCET drops to 25,792 cycles. This example provides a strong argument to add dataflow analysis to the WCET analysis tool.

The last two examples also show the issue when a real-time application is developed without a WCET analysis tool available. Getting the feedback from the analysis earlier in the design phase can help the programmer to adapt to a WCET aware programming style. In one extreme this can end up in the single-path programming style [20]. A less radical approach can use some heuristics for a WCET aware programming style. E.g. for the `UdpIp` example a *special* version of the UDP and IP checksum calculation for short messages can be added to the UDP/IP stack.

6. CONCLUSION

In this paper we have presented the combination of a time predictable Java processor and a WCET analysis tool based on the integer linear programming approach. The architecture of the Java processor greatly simplifies the low-level part of the WCET analysis. An instruction cache, named *method cache*, stores complete methods and is easy to integrate into the WCET analysis.

The WCET analysis tool, with the help of loop annotations, provides WCET values for the schedulability analysis. We have also integrated a two block method cache into the analysis. This cache configuration can be analyzed at the method level and does not need the full program CFG. As a future extension we will also integrate the variable block cache. Besides the calculation of the

WCET the tool provides user feedback by generating bytecode listings with timing information and a graphical representation of the CFG with timing and frequency information. This representation of the WCET path through the code can guide the developer to write WCET aware real-time code.

As future work we consider integrating the WCET analysis into the Eclipse IDE. The worst-case program path will be colored in the Java source. Furthermore, methods can be annotated with their WCET values and a graphical representation of the CFG. This form of immediate feedback can guide the programmer to optimize the worst-case path. More types of annotations can lead to tighter WCET estimates. Automatic detection of loop bounds (for simple cases) and a more elaborate data flow analysis will simplify the usage of the WCET analyzer.

Acknowledgment

The authors thank Peter Puschner for the discussion on ILP based WCET analysis and his constructive comments on early versions of the paper.

7. REFERENCES

[1] C. Artho and A. Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. *Electronic Notes in Theoretical Computer Science*, 141(1):109–128, December 2005.

[2] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level analysis of a portable Java byte code WCET analysis framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[3] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using java byte code. In *Proc. 12th EUROMICRO Conference on Real-time Systems*, Jun 2000.

[4] 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.

[5] M. Dahm. Byte code engineering with the BCEL API. Technical report, Freie Universitat Berlin, April 2001.

[6] DCT. Lightfoot 32-bit Java processor core. data sheet, September 2001.

[7] S. Dey, P. Sanchez, D. Panigrahi, L. Chen, C. Taylor, and K. Sekar. Using a soft core in a SOC design: Experiences with picoJava. *IEEE Design and Test of Computers*, 17(3):60–71, July 2000.

[8] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[9] T. R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.

[10] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Javatm virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.

[11] Imsys. Im1101c (the cjip) technical reference manual / v0.25, 2004.

[12] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[13] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[14] Y.-T. S. Li, S. Malik, and 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, 1995. IEEE Computer Society.

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

[16] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[17] J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[18] P. Puschner and G. Bernat. Wcet analysis of reusable portable code. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[19] P. Puschner and A. Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.

[20] P. Puschner and A. Burns. Writing temporally predictable code. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 85, Washington, DC, USA, 2002. IEEE Computer Society.

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

[22] P. Puschner and A. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

[23] 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.

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

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

[26] M. Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

[27] M. Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, Munich, Germany, March 2006.

[28] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, 1989.